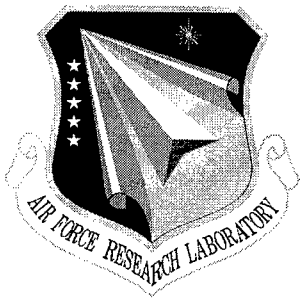
AFRL-IF-RS-TR-2001-3
Final Technical Report
January 2001



ON WEB-BASED MODELS AND REPOSITORIES

University of Florida

Paul A. Fishwick and John F. Hopkins

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-3 has been reviewed and is approved for publication.

APPROVED: 

ALEX F. SISTI
Project Engineer

FOR THE DIRECTOR: 

ROBERT E. MARMELSTEIN, Maj, Deputy Chief
Information Systems Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFSB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JANUARY 2001	Final Sep 98 - Aug 00

4. TITLE AND SUBTITLE
ON WEB-BASED MODELS AND REPOSITORIES

5. FUNDING NUMBERS
C - F30602-98-C-0269
PE - 62702F
PR - 4600
TA - II
WU - D7

6. AUTHOR(S)
Paul A. Fishwick and John F. Hopkins

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Florida
Department of Computers and Information Science Engineering
Gainesville Florida 32611-6120

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFSB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2001-3

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Alex F. Sisti/IFSB/(315) 330-3983

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document contains a description of research performed at the University of Florida on the concept and implementation of web-based models and repositories. The document is divided into three sections. The first section briefly presents an overview of the contract...what was accomplished, why certain areas were pursued, and how those efforts contribute to the field of modeling, in particular, and computer simulation, more generally. The second section represents papers published by the research group; while the final section is comprised of classic readings that bear a relation to this project and to the philosophical foundations behind it.

14. SUBJECT TERMS
Web-Based, 3-D Modeling, Repositories, Digital Objects, Aesthetic Programming, Multimodeling

15. NUMBER OF PAGES
172

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Introduction

This document contains a description of research performed at the University of Florida on the concept and implementation of web-based models and repositories. The document is divided into three sections. The first section briefly presents an overview of contract F30602-98-C-0269; what was accomplished, why certain areas were pursued, and how those efforts contribute to the field of modeling and simulation. The second section consists of papers published by the research team, while the final section is comprised of classic readings in the literature that bear a relation to this project, and to the philosophical foundations behind it.

Alex F. Sisti

On Web-Based Models and Repositories

Paul A. Fishwick

Department of Computer & Information Science & Engineering
University of Florida
Gainesville, Florida 32611, U.S.A.
fishwick@cise.ufl.edu

October 18, 2000

1 CONTRACT

This document refers to work performed under contract F30602-98-C-0269, "A Web-Based Model Repository for Sharing and Reusing Model Components" under the direction of Program Manager Alex Sisti and Contracting Officer Gary Slopka, Rome Lab. Paul Fishwick, University of Florida was the Principal Investigator.

2 OVERVIEW

This document contains a description of the research that we performed at the University of Florida on the concept and implementation of web-based models and repositories. The document is divided into three sections. We are inside the first section, where I overview what we did during the period of the contract, what was accomplished, why we did it, and how I see our efforts contributing to the field of modeling, in particular, and computer simulation, more generally. The second section represents papers published by our research group, and the third section is a set of reading material that bears a relation to the rube Project in its application of metaphor to model construction. My students deserve a tremendous amount of credit for not only implementations, but also for their philosophies and contributions to methodology. They are the fuel that drives the research, whereas their own fuel is their assistantship status and their university credit. Together, with my students, I worked on basic research elements, some of which will ideally have an impact on the way that models are constructed using a time horizon extending out 10 years hence. The students involved in this project were Robert Cubert, Kangsun Lee, John Hopkins, Linda Dance, and Taewoo Kim.

3 GOAL

When I wrote the proposal a year and a half ago, I promised to deliver a methodology and system for web-based modeling and its associated repository mechanism. This has been accomplished with the creation of two sorts of end products: refereed papers published in the literature, and tested software. What does it mean to have a "web based model repository?" A reductionist would suggest that one must first define the constraints on what it means to be a "model" and then the machinery for enabling the World Wide Web with models in the form of a repository. Both of these meanings involve inherent difficulties, since model-making is vast in scope. If one asks a random selection of 10 people what defines "modeling," one easily gets 10 different answers, but it may come as somewhat of a surprise to get similar variance from 10 people who think of themselves as experts in modeling and simulation (M&S). Instead of this situation being indicative of paranoid schizophrenia, it suggests that the field of modeling is rich indeed, beyond any of our wildest dreams. To wit, a quick look at the Webster's dictionary reveals modeling to be applicable to every human endeavor from logic and the arts to science. In contrast, model execution is better understood and more limited when we speak of execution on sequential machines. The execution of a model is accomplished through writing a computer program. Since computer scientists have been writing computer programs for half a century, albeit with a panoply of available languages, model execution methods limit the scope of inquiry to program execution.

The Department of Defense is abuzz with the High Level Architecture (HLA), and with good reason. HLA, along with its Distributed Interactive Simulation (DIS) ancestor, promise to increase the efficiency of training in a virtual theatre of armed conflict. The research goals of HLA are primarily those of executing programs, and not model design. I recall being at a conference where Judith Dahmann suggested that solving the modeling problem was more difficult, and that as a first step it was necessary for HLA to focus on the immediate problem of designing federates and their accompanying protocols. Dahmann was correct, and the decision to build HLA, along with its execution methods and communication procotols, served as the logical first step. HLA represents a significant evolution from the DIS fledging, and will help the military to link disparate simulators and simulations for training.

Because of its execution bias, HLA, however, has not addressed longer-term issues regarding model design, even though preliminary object templates (OMTs) represent a beginning. My contractual purpose was to consider potential model repository design for HLA and its progeny, and to simultaneously perform research on modeling. HLA is certainly not alone in its focus on execution. As it turns out, the vast majority of computer simulation books present excellent material on statistical experimental designs, methods for distributed and parallel execution, and coverage of state of the art simulation packages being executed. However, because modeling is so diverse, it has received insufficient

treatment. Nobody is quite sure what to do with modeling, where to place it, or how to categorize it. Some work has been done, especially at the mathematical model level, but more work remains to encompass all types of models. In 1994, I designed a taxonomy of modeling that closely parallels styles of computer programming: declarative, functional, constraint, and so forth. This taxonomy went beyond discrete event systems to capture the structure and *syntax* of models, as well as applying semantics to the structure. A salient point was to demonstrate that models can be classified by *style* rather than how they are to be interpreted *semantically* for execution (i.e., discrete-event, continuous). Programming languages are similarly organized. It is, perhaps, that modeling is part art and part engineeering that makes it difficult to grasp, and limits its audience. In any event, the influence of modeling is being increasingly felt in many areas of science and engineering. Pictures that were once drawn on paper now live inside the computer, and code can be generated from the pictures, which causes the pictures to gain status and importance. The pictures and the "look and feel" of a system capture the essence of modeling, since they act to mediate human interpretation of dynamic system characteristics.

4 BACKGROUND PREPARATION

The path that we have taken has yielded many insights, all of which I feel will be valuable to the field of simulation and to the Air Force, being a large consumer of this field. Our path begins during the writing of the proposal, when we finished our Multimodel Object-Oriented Simulation Environment (MOOSE) implementation. MOOSE was subsequently renamed OOPM, Object Oriented Physical Modeler, as a result of a potential trademark infringement which came to our attention after the significant press that we received from many newspapers and Wired Online. OOPM is a complete multimodeling design and simulation system using traditional 2D graphic modeling elements. The idea of multimodeling is to build models as we build houses—using diverse materials, each material being used where it is best needed. Although, this may sound trivial at first, multimodeling involves issues with formalism, inter-level coupling, and most importantly, the implementation of a system capable of simulating a model with many layers, and supporting five model types (finite state machine, functional block model, rule-based model, nth order nonlinear ordinary differential equation sets, and System Dynamics). In retrospect, it is not too surprising that most modeling systems are homogeneous since multimodeling is difficult to achieve in implementation, and expensive in the support of more than a single model type. Multiple user interfaces are required when a model is graphical. It is much easier to construct, for example, a Petri net modeler and simulator because this represents only one model type, and therefore only one user interface.

The work on OOPM had many lessons for us, with the biggest lesson being that of the creation and debugging of a Graphical User Interface (GUI), which was difficult

and labor-intensive. My hat is off to companies who can produce an effective, bug-free, GUI, since the undertaking is substantial. In an attempt to make the GUI platform independent, we started with Tcl/Tk on the front end and C++ on the back end. This was done prior to the onslaught of Java in the marketplace, but still remains a good decision given the performance advantages of C++ generated code, which tends to be highly efficient. Unfortunately, Tcl/Tk had its own warts and pecularities. We spent significant time trying to work around known bugs, and derived a method for "object-oriented programming by convention" since Tcl, being a scripting language, is not object-oriented. Working with C++ was fairly straightforward since the C++ language, and its common tools, were fairly mature by the mid 1990s, with the possible exception of template support.

5 QUESTION EVERYTHING

The idea behind a Gedanken experiment (i.e., mental model) is to question basic assumptions, see where these assumptions take you, and then proceed accordingly to create new paths. I created several Gedanken experiments to help drive our research forward into web-based modeling. Here are challenged assumptions starting with the homogenous model type developed circa 1991:

1. A model shall be of one homogeneous type.
2. A model shall be static and not dynamic.
3. A model must conform to pre-defined standards.
4. A model shall not have aesthetic properties.
5. Modeling should adopt object-oriented design principles.

The questioning of these assumptions came very slowly over the period of many months of study and research, often by reference to disciplines outside of the traditional simulation books. The goal was to attempt to *reinvent* modeling in simulation so that it could be seen as equivalent to the more common usage of the term. The modeling of clay and the modeling of a finite state machine should not, in theory, be that different. As we discovered, there is very little difference: symbols are pictures, and pictures in the models of science have tended to be minimal because of long-standing economic and cultural constraints.

5.1 Assumption 1: A model shall be of one homogeneous type

This assumption was challenged in my 1991 Winter Simulation Conference paper on heterogeneous modeling, and then subsequently discussed in more detail later on. There is a chapter on multimodeling in my 1995 book, *Simulation Model Design and Execution*. It is an amazingly simple assumption, that heterogeneity is not only possible in model design, but valuable as well, given that different model types may capture the appropriate semantics for different abstraction levels. Diversity in modeling is good.

5.2 Assumption 2: A model shall be static and not dynamic

Take a look a a finite state machine or even a multitrack based interface in your favorite animation language, such as Flash. The model does not move or shake. The idea is to cause the *scene* to be animated. The model structure doesn't move, but the thing being modeled does move. If you study a Petri net, for example, you realize that something is going on in this type of model that doesn't occur in many others: the model itself is dynamic, and not only the system that is being modeled by it. The only real objection to dynamic models is that they are not standard, which is an issue which we next address. And still, by breaking the idea of static model design, we yield new model forms, in the spirit of moving colored tokens in Petri nets, which are active. Components change shape, position, and appearance over time. Dynamicism suggests looking at existing static models and asking what might happen if we injected life into them. Wouldn't that make models more interesting, more informative, and less abstract? One interim result in this confluence of model and object is that we came to realize that there is not much difference between an object and a model of the object, just as there is not much difference between a small miniature sailboat and its big-as-life bretheren. They are both objects; one object serves to model another. This shatters the illusion that a model is an ethereal, mental creature and that the object is *real*, or that the two forms are fundamentally different. They are the same. If I pick up a yellow tennis ball and claim that it is a scale model of our sun, this explains a great deal, with the appropriate arm movements. There is nothing inherent in either the tennis ball or sun that makes them models. The tennis ball can be used on the court as well. Objects are not models by virtue of inherent structural characteristics. They are models only when someone produces a set of rules, a metaphoric mapping, that binds the source to the target. The arts have known this for a long time. The semioticians from Peirce to Eco have also known this. We live in a world filled with signs, and by connecting signs, we create meaning. We are just beginning to understand modeling, and appreciate it, after being held back for so long with expensive computers and minimal display elements. Too often, a sign or set of signs is misconstrued as "the real thing."

5.3 Assumption 3: A model must conform to pre-defined standards

The issue of standards comes into play in modeling as it does for virtually every other domain. If we make new model types, are we not destroying achievements made in standardization? There are many responses, the first being that standards, themselves, are dynamic. They change over time, but they change very slowly. This slow change causes potential problems for creativity and ground-breaking research, almost by definition. By constraining a basic design, one gains conformity and perhaps some human performance, but one also loses a considerable amount. And yet, standards, at some level are good and necessary. It turns out that we can have both standards and creativity, side by side. The Human-Computer Interface (HCI) domain is a perfect illustration of the issues. HCI is about standards, but that sounded good when the goal was to create designs palatable to the general populous. The rapid change in software and hardware economies are altering the HCI landscape. Technologies allowing users to choose or design their own interfaces are running rampant. Standards still exist, but at a lower level. Just as I can design my car but with the knowledge that it will always have a steering wheel, a pedal, and an engine, many other features are under my control. If I don't care to create my own design, I pick a default. The "skinz" movement is similar, allowing users to choose default, "standard" skinz, create their own, or choose one from thousands of skinz artists. The key point in HCI evolution is the democratization of the interface, rather than dictations pronounced from "on high." A recent special issue of Communications of the ACM (CACM) promotes the "personalized interface" in computing, rather than one interface force-fed to the masses. Marketing and advertising executives have known this all along, with the goal being to ultimately target the individual if you want to sell your product.

Relating the HCI phenomenon to modeling is fairly straightforward. Consider that modeling cultures have always existed, with standards being decided upon from the ground-up, and not top-down. Thirty-year old modeling communities exist in force from Petri nets, compartmental models, Bond Graphs, and System Dynamics to formalisms for Discrete Event Systems, Control Block-Models, the "Unified Modeling Language (UML)," and visual, multi-tiered finite state machines. Whereas the holy grail of physics suggests a unified model, this goal achieves a foundational theory on top of which models are born. Even in physics, numerous model types exist to aid scientists to mediate the boundary between theory and phenomenon. The theory is generally construed to be very low level and all-encompassing, whereas the domain of models is much larger and more varied. The theory is generally taken to be a standard, until a new Kuhnian paradigm knocks it from its pedestal.

5.4 Assumption 4: A model shall not have aesthetic properties

If you look at models for entertainment and art versus science, there is little doubt that modeling representations have been largely dictated by the state of the economy. Jaron Lanier once remarked (1989) that if *virtual clothing* had existed when language first developed, our languages would likely be much different than today. The idea is that any society capable of creating virtual clothing could just as easily create anything. Imagine the existence of a portable holographic display that could be positioned anywhere in space between people in a group. How would this affect language? This brief thought experiment causes one to question not only the adherence of our natural language to symbols but also our modeling methods to symbols and primitive, Platonic-like circles, arcs and rectangles. When one surveys the types of dynamic models that we have available, we are still stuck in the valley of circles, with communities debating which type of circle or rectangle should be connected to which other type. It is time to climb out from the valley. As economy changes, and we can efficiently create pictures, 3D shapes, and 3D audio with a wave of the hand, this is bound to have a profound affect on how we model. While at first, this sounds like a lot of science fiction, technologies exist now that can help us to achieve these goals.

Apart from the issue of scientists being couched in formalism and minimalist representations since the first day at school, there is cause to wonder whether aesthetics has a role to play in modeling, simulation and science. Our first contention is that the creation of aesthetic forms does not mitigate the need for formalism. Instead, one can build layers of interface on top of symbolically specified models in much the same way that compilers are constructed on top of computer architectures. With the straightjacket of economy not limiting us to symbolic and 2D models, we have lots more possibilities. The important thing to recognize is that our economic conditions, and not idealist design criteria, have dictated the minimalist form of our models. Can the "rate symbol" in System Dynamics be represented as something beautiful? Can we use a 3D valve, complete with a texture map and sound? The answers are in the affirmative if we can evolve beyond our minimalist modeling culture. Some personal observations of my own learning techniques over the years have taught me that if I try to understand a methodology that I find intrinsically boring, I will have to re-read the book twenty times. I had one such occasion with some simulated-related material that *I felt I should know*. The issue has little to do with capacity for learning, in anyone's case. The issue is capturing the interest and enjoyment of the participant. Personal choice and aesthetics are very important to modeling. If I enjoy something, it will affect my use of it, my cognition, and memory retention. Some recent experimental evidence of this can be found in HCI, under the rubric of "aesthetics for increased usability" (ref. Section 3 for a copy of this paper). Again, this should not come as a surprise to us. Our children, who are steeped in game console culture, will have little problem in relating to 3D virtual spaces. The only issue is whether we will have laid

a framework for this kind of modeling by the time they enter the workforce.

5.5 Assumption 5: Modeling should adopt object-oriented design principles

Object-oriented design (OOD) projects a paradigm onto the programmer and modeler. OOD is defined by encapsulation within classes and instantiated objects, and inheritance. We have found the most useful aspect of OOD is its metaphor toward the real-world, at least for making computer programs easier to create. Problems arise when we consider the previously challenged assumptions and try to rectify these with OOD. The main problem is that modeling is inherently *relational*. The simple example of the tennis ball and sun demonstrates this for scale modeling, but behavioral modeling is no different. The tennis ball and Sun have the same *object status*, and the fact that one is modeling the other is an imposed *relationship* between these two objects. Therefore, it does little good to imagine that a behavioral model should be defined as a series of methods encapsulated within an object. When models are made from abstract symbols, it can be imagined that these symbols live inside the object. However, any surfacing of models into full-fledged objects causes deep philosophical concerns with OOD. OOD, by itself, represents a good set of design principles for programming, but the introduction of models suggests that we create semantic networks among objects, with links defining the modeling relationship. When we study the problem more deeply, we realize that attributes have similar problems to methods when it comes to modeling. For example, I can define a variable that is a 2D array of integers, and make this variable an object attribute. So far, this sounds logical, with the variable encapsulated, and made private, inside the object, only to be modified by *accessor methods*. However, when surfacing the array as a scale model, which it really is if we think about what it means to be an array, then we encounter problems. A rectangular wooden case of mailboxes can model a farmer's field, in general shape and in cell-subdivision. But the field and the case are two independent objects. It makes little sense to imagine one encapsulated by the other.

6 THE rube PROJECT

The name "rube" is borrowed from Rube Goldberg, who constructed many fanciful cartoon machines. While Rube's machines are often referred to in a negative way for an engineering project gone bad, the positive aspect of Rube's work is that he created memorable and entertaining models of dynamic systems. After looking at one of his cartoons, it is ironic that you will better remember and reason about his machines than the supposedly simpler machine designs not designed as "Rube Goldberg machines." The area of mnemonics, and its associated techniques, suggests that we create artificial 3D worlds and spaces to better remember facts and figures, which tend to be boring and difficult to

digest. Surfacing abstractions in amusing ways help us to learn and remember.

It might not seem possible to reverse so many ingrained modeling assumptions given today's technology, but we happen to be at a critical juncture where several software technologies have recently appeared to aid our modeling movement. To create the sorts of challenging models we envision, I decided on the Virtual Reality Modeling Language (VRML) as the web-ready framework for modeling dynamics. In this framework, one can easily create 3D worlds and objects that represent both the model and modeled system. VRML was designed with the web in mind, and so the concept of "repository" becomes almost moot since VRML worlds of any size, down to single objects, can exist anywhere over the web, and by virtue of Javascript and Java, can be executed on the client machine. So, a simple repository is made possible through URL linkage. The key work in our implementation was to derive a methodology for constructing these dynamic models in a structured way, and to build a set of VRML Prototypes to make it easy for modelers to choose common dynamic model types in their multimodels. I have been active on the next-generation VRML mailing lists to ensure that VRML2000 and X3D (which will ultimately replace VRML) contain the right time and event models to support discrete event and continuous simulation efforts. X3D stands for "Extensible 3D" and is structured completely in XML, which is the upcoming language of the web. The transitioning from VRML to XML is natural and logical, as XML has many benefits for model design. Our delivered software and methodology can be found under my home page under Research ⇒ rube Project. Moreover, I have developed a large list of links for VRML and X3D under Teaching⇒VRML links.

7 DISCOVERIES

Our primary discovery is that modeling, as a field, achieves many possibilities by challenging core assumptions. Model types from scale models to artistic models become unified once we strip away artificial barriers, such as minimalist form, and promote upcoming movements directed toward the individual, and away from the general populace. The technique that we have developed is appplicable to software engineering, as well as system dynamics. Modeling has always been a way to build a structure in hopes of mediating the gap between human and phenomenon. With new techologies at our beckoning, the possibilities are limitless.

Despite our focus on VRML and X3D, we realized early on that it was not so much that we were taking 2D models and representing them in 3D, but that we were changing the fundamental structure of model components. The term we use for this ability to restructure components is *aesthetics*. By applying the principles of art to modeling, we are able to not only proceed toward 3D, but to create more interesting 2D pieces. Once we remove the self-imposed shackles, and get rid of our green flow-chart template for making

circles and rectangles, we realize a more all-encompassing proscenium for our modeling purposes.

8 EDUCATION

Our vision for modeling requires a different sort of computer scientist, one with a combined knowledge of the arts and engineering. We have developed a set of curricula to create this new student. Three programs exist within our department, Computer and Information Science & Engineering (CISE), and two programs exist in Fine Arts. All programs go under the rubric *Digital Arts & Sciences (DAS)*. The CISE program confers a *Degree of Engineering* with students taking required Engineering core classes in addition to Fine Arts classes. The student achieves fluency in the arts and engineering. A detailed curriculum attachment can be found in the third section of this report. The umbrella construct that oversees these programs is called the *Florida Digital Worlds Institute*, and the search for Director for this institute is underway. At first thought, the DAS students would appear to be able to obtain positions in multimedia, entertainment, gaming, simulation and scientific visualization; however, given the goal of aesthetic modeling, we see this student as becoming the prototypical computer scientist for the 21st Century.

9 AIR FORCE APPLICATION

Modeling is used for training for air conflicts, strategy, and in the huge investments involved in acquisition of hardware and software. Research and routine software development needs to taken place to support M&S. While today's mature software technologies require immediate deployment and short-term time horizons, we must also demonstrate methods and technologies that will dramatically change the face of M&S. Otherwise, the military would have its own long-term strategic views dictated by today's software marketplace. Our work in the rube Project is our contribution to this effort. Our goal is not be so theoretical and philosophical that we produce no software, or create minor improvements in modeling. We are methodology and software builders. Our goal is to define radically new modeling approaches that show tremendous potential over the next decade.

10 KEY WEB SITES

- Fishwick's Web Page: http://www.cise.ufl.edu/~fishwick
- rube Project Page: http://www.cise.ufl.edu/~fishwick/rube

- VRML Links Page: http://www.cise.ufl.edu/~fishwick/vrml
- Digital Arts & Sciences Program: http://www.cise.ufl.edu/fdwi

Issues with Web-Publishable Digital Objects

P. A. Fishwick

Computer and Information Science and Engineering Department
University of Florida, Gainesville, FL 32611

ABSTRACT

Our goal is to promote the publication and standardization of *digital objects* stored on the web to enable model and object reuse. A digital object is an electronic representation of a physical object, including models of the object. There remain many challenges and issues regarding the representation and utilization of these objects, since it is not clear, for example, who will create certain objects, who will maintain the objects, and what levels of abstraction will be used in each object. This article introduces some of the technical and philosophical issues regarding digital object publication, with the aim of enumerating technical, sociological and financial problems to be addressed.

Keywords: Digital Object, Multimodeling, Model Abstraction, Object Orientation

1. INTRODUCTION

One of the most critical problems in the field of computer simulation today is the lack of published models and physical objects within a medium—such as the World Wide Web—allowing such distribution. The web represents the future of information sharing and exchange, and yet it is used primarily for the publication of documents since the web adopts a "document/desktop metaphor" for knowledge. In the near future, we envision an "object metaphor" where a document is one type of object. A web predicated on digital objects is much more flexible and requires a knowledge in how to model physical phenomena at many different scales in space-time.

If a scientist or engineer (i.e., model author) works on a model, places the model inside objects, and constructs a working simulation, this work occurs most often within a vacuum. Consider a scenario involving an internal combustion engine in an automobile, where the engine is the physical object to be simulated. The model author's task is to simulate the engine given that a new engineering method, involving a change in fuel injection for example, is to be tested. By testing the digital engine and fuel injection system using simulation, the author can determine the potential shortfalls and benefits of the new technique. This task is a worthwhile one for simulation, and simulation as a field has demonstrated its utility for objects such as engines.

Let's analyze the problems inherent in this example. There is no particular location that will help the author to create the geometry of the engine and its dynamics. It may be that other employees of the company have made similar engine models in the past, and that these models may be partially reused. If this is the case, the model author is fortunate, but even if such a company-internal model exists, it may not be represented in "model form" (ref. Sec 3). There may be other model authors who have already constructed pieces that our model author could use, but there if there is no reuse and no standard mechanism for publishing the model or engine object, then this is all for naught. The model author may also be concerned with creating a fast simulation. While algorithms for speeding simulations are important, by solving the reusability problem, we also partially solve the speed problem since published quality models of engines will battle in the marketplace for digital parts, and the best engine models and testing environments–involving very fast and efficient simulation algorithms–will win out in the end. Therefore, the problem of reuse of engine objects and components lies at the heart of the simulationist's dilemma. Fast, efficient and quality models could be available at some point in the future, but today there is no infrastructure or agreed-upon standards (ref. Sec. 11) for true digital object engineering.

Many of the issues surrounding digital objects and their representation can be resolved, at least partially, using the *physical metaphor*. We ask a question such as *Who will maintain a digital object?* or *Where is the digital object located?* and we obtain answers by phrasing the question within the corresponding physical domain, yielding *Who maintains the physical object?* and *Where are the physical objects created?* The answers to the latter question

Other author information:Email: fishwick@cise.ufl.edu; Telephone: 352-392-1414; Fax: 352-392-1414; Supported by the U.S. Air Force, GRCI, Incorporated, and by the ATLSS Project of the Department of the Interior.

suggests possible answers to the former. This is a simple technique, but fairly powerful in addressing many of the issues that we will present. We will proceed to outline problems involving digital objects, first by defining them and then continuing with issues that surround digital objects and their future. We hope that this paper can serve as a starting point to debate some of these issues. Some issues were addressed in detail at a recent innaugural conference on web-based modeling and simulation[1]*.

2. DIGITAL OBJECTS

Before enumerating the problems that will face the model author, we will state our goal, which is to provide a representation for digital objects on the web. A "digital object" is the digital counterpart to the physical object. Therefore, the digital object contains attributes and methods, where some of these will be models. Two prominent models are the geometry model and the dynamic model since these models capture the shape and behavior of an object. For the engine, we might publish a geometry model based on NURBS (Non-Uniform Rational B-Splines), and for the dynamics, we might create a set of equations representing the transportation of gasoline through the pipelines involved in fuel-injection. These two models are components of the specific engine object and could also be components in an Engine class from which an engine object is created.

Who will initiate the process of storing digital objects? Where is the incentive? This is less of a technical consideration and more a question of whether or not we should deliberate on digital objects. In the remainder of the paper, we'll present arguments for the deliberation issue. In terms of incentive to create digital objects, the marketplace will play a key role. Those industries that plan to sell their products will find their sales increased if they offer their customers an opportunity to test digital products prior to buying the real ones. Currently, some companies on the web offer pictorial and technical specifications for their products, but these are not a complete substitute for digital objects. We feel that the federal government will play a major role in the adoption of standards and digital object proliferation. In particular, during DoD acquisition phases, a stipulation can be put in place where vendors must deliver bids with an accompanying digital object specification.

For the remainder of the article we will often refer to different components such as models, objects and classes. Objects can theoretically be defined without an associated class; however, it is most useful to create a corresponding class when creating any object. This fosters reuse and inheritance and makes it possible to create similar objects from the class. Models reside inside classes and objects.

3. MODEL VERSUS CODE

In many cases, the model (defined as an abstract, and often visual, representation of the engine's behavior), may not be surfaced at all. Instead, there may be a large program whose simulation yields the object's behavior. The model author for the engine, once coding is complete, may speak of having generated a "model", when in reality there is no model except in the model author's mind or sketched on a notepad. The cognitive, notepad-style, model is what is required to be surfaced where it serves as the human-computer interface. In this sense, a "model" and a "human-computer interface" are inseparable. The model serves as the interface which is compiled into target code with which the author need not be concerned. Unfortunately, program code is not a good substitute for a model even though it could potentially be seen as one in the extreme. The "model" must be abstract and must represent a close match to the model author's cognitive map of the engine. Programs tend to obscure these maps by focusing on detailed semantics rather than a more abstract syntax. Most model authors tend to like visual representations of phenomena, and the author of the engine model is no exception. Some of the models that the author will use will be equational in representation, and other models will be graphical.

In addition to graphically oriented cognitive representations that require surfacing, it is also critically important to have an underlying formal semantics for each model that can be accessed to resolve ambiguities and disputes. The more abstract the representation, the more easily it can fall victim to issues involving semantics. So, it is necesssary to maintain a chain of translation from high level abstract model to the semantics that are defined to a level of detail so that execution of the model is unique and unambiguous.

*An online version of these proceedings is available through the web page `http://www.cise.ufl.edu/~fishwick/webconf.html`.

4. REUSABILITY AND SPEED

The author for the engine may find it frustrating to have to reinvent the wheel by constructing a new model. This is where the expense of simulation raises its head. Simulation, as a method, is expensive because of the lack of available digital objects and models that can be reused. While it is true that this particular author may have a new set of questions to be asked of the model, the author could benefit from reuse with modification of the model to fit the author's specific questions to be answered from the simulation.

There is much research into the problems of speeding up model executions. The engine author may well wait hours for results from the simulation. This is unfortunate and costly in terms of hours and machine time. As a simulation community, we need fast algorithms, but without well-published digital objects and their component models, the speed issue resolves problems for only one project and one model author. With published digital objects, representing test environments for fast simulation, authors can spend less time worrying about speed and more time focusing on model structure and design. The web-based marketplace of objects will be fundamental in solving the speed problem just as the physical marketplaces does for the survival of only the most efficient objects.

What is to be reused? Certainly, objects will be reused just as physical objects are reused in the form of "plug and play" techiques used to construct everything from engines to houses. However, the models inside the objects should be reused and the classes, from which many objects will be created, serve as a vehicle for reusability. Therefore, reusable components are *class*, *object* and object structure in the form of individual *models*.

5. INTERFACE AND COUPLING

How might we mix and match components for an engine? Since the fuel injection delivery pipelines physically meet the gasoline tank, the digital equivalents must perform a similar conjunction. The interface to a digital object must be specified to ensure that the data types, at the very least, match. We let an object have three types of ports: input, output, and input/output. Objects can connect to one another via these ports. A line that extends from port 3 on Object A to port 1 on Object B must carry the same type of signal, and the data structure associated with the output on port 3/Object A must match the input on port 1/Object B. There may be additional constraints that can be specified for minimizing problems. These constraints can be specified in a formal constraint language or the language of predicate calculus.

6. AUTHORING AND MAINTENANCE

Who will create the digital objects? Should there be one repository for a specific type of object or should we compete in the marketplace of objects with multiple authors? A reasonable strategy is to let the marketplace dictate which authors are most successful. Some authors may be interested in their own particular object and internal models, whereas others may have alternate motives (ref. Sec. 9). There are many potential strategies for locating digital objects. One strategy suggests that digital objects be located where their physical object counterparts are manufactured. Therefore, the author the company creating the automobile engine will create the digital automobile engine, and the company making the piston rings will create the digital piston rings. We will term this strategy *developer-based colocation*. Reuse can be created at every level so that even the engine author can reuse objects since it is doubtful that all engine subcomponents are manufactured by the automobile company. While the developer-based strategy is appropriate for engineered objects, we must concern ourselves with natural phenomena as well. What about a representation of the Everglades National Park for ecological studies? The *management-based colocation* strategy suggests that the Department of the Interior, which is responsible for this piece of land and its ecology, host the site where the digital Everglades models[2] can be easily accessed. The Department of the Interior can likewise create contracts for industries to compete for the right to carry the objects on their sites, or for a more generally accessible site controlled by the Department. A more arbitrary approach suggests that digital objects can be located physically anywhere on the web. Experts in automotive design at a University might host a site containing digital engines or subcomponents.

7. MODEL ABSTRACTION, COMPLETENESS AND REUSE

Let's say that our model author finds an engine that contains fuel injection geometry and dynamics. Is this particular object appropriate for the model author's simulation requirements and goals? This is a particularly acute problem and one that is central to many issues concerning digital objects. First, we must acknowledge that the author may

indeed find that the engine that he obtains on the web may answer a certain percentage of the questions he wants answered, say 75%. Second, another engine may be available that is different from the first model, and yet contains some extra model semantics which will allow for answering another 20% of the remaining questions. At first, it may appear impossible to provide the functionality in a digital object that will satisfy everyone. This is true. No single object will satisfy everyone, however, there are key considerations and steps toward meaningful digital objects. First, we note that objects should be created, as in the physical world, with the ability to accept input while not dictating the exact nature of that input. Finite element programs and programs based on Newtonian physics are able to simulate a very large class of system. This is because, for instance, forces that affect an object may come from an infinite number of sources, but the source identity is not relevant if the object's input is a force or a vector field of forces. The manifold for an automobile engine may be affected by a wide variety of physical objects, but the identity of these objects does not affect the physics since one is concerned with an impressed, generalized force. The manifold is unconcerned whether the applied force eminates from a person's hand, a wrench or gas expansion. If this invariance to input did not exist, today's engineering software would be severely limited.

Through multiple inheritance, it is possible to inherit all necessary components to create a new hybrid class. The model author, once finished with the simulation, may decide to publish this hybrid class, thereby augmenting the base set of digital objects on the web. Dynamic models and methods incorporating finite element calculations can be combined with point-mass calculations through multiple inheritance, and need not necessarily be located in one monolithic class structure. Also, some of the models required may be located in fundamentally different objects that have the similar methods to what is required. The broader the dynamic or geometric model, the greater the possibility for the author to locate models instead of just objects that match the existing requirements. The author may also find that he needs to create some new models that he cannot locate either in class, object or model form. At the very least, the author has minimized wheel reinvention through a comprehensive search of the web for classes, objects and components. There is also the option of accepting a component that answers all the critical questions to the required level. There are conscious tradeoffs to be considered. We make these same tradeoffs when buying physical objects that may solve most of our needs but not all of them.

Most objects will contain multiple abstraction levels. The abstraction levels presented may not exactly match what is required, but through a search process and through reuse, we can create new levels if they are needed. It is probably unrealistic to imagine that published digital objects will behave in every way, and at every abstraction level, that the physical objects behave. This suggests that model authors who do publish objects be careful about stating up front the constraints of their objects–what they can do and what their limitations are. Over time, and given a free market for digital objects, we estimate that objects will improve over time to yield better and more accurate results that appeal to greater numbers of model authors. Moreover, the taxonomy of objects in the form of class hierarchies will improve in structure to maximize the benefits of aggregation and inheritance.

8. ACCESS

Given that the author of the engine needs to find objects and components, how is he to locate and access them? The most straightforward idea is that model search can be seen as similar to text-based search in modern web search engines. To find an engine, search for the keywords "automobile engine." The result should be a conceptual model consisting of classes and relations. The class from which a specific object is created can be highlighted and displayed with its immediate class context. This is similar in concept to the way that *Yahoo!*† organizes its subject taxonomy, except that our tree is concept/class-based. Other search methods include picture-based search and an immersive 3D-based search for objects.

Model repositories will contain class hierarchies and embedded geometry and dynamic models, and we envision that model repositories will proliferate over the web to support the model author. Along with the need for repositories, will be the need for *model bases* to support concurrent access, protected model information, querying and caching of model structure. One of the most significant changes to accessing will be based on a new metaphor for the web based on objects and classes, instead of on documents. Documents will still retain their importance, but will be viewed as one type of physical object rather than as the overriding metaphor for representing all forms of information.

†`http://www.yahoo.com`.

9. INTELLECTUAL PROPERTY AND ECONOMICS

If an author creates a digital engine object, then why should this author publish it? Isn't there a conflict with intellectual and industrial patent and copyright assumptions? And should the digital object be free or should a company charge for the object? We see the need for both commercial and public-domain digital objects. This would reflect the way that software is currently marketed over the web. A free version may be a "demo version" with the full version being sold for profit.

Could a digital engine maker disclose company secrets? This key issue is not only a concern to industry, but to all model builders. A model, and its enclosing object, reflects intellectual property similar to what is published in a technical paper. There are two concerns that we can address. The first deals with industrial objects. Industrial objects can be reverse engineered to see "what makes them tick." Therefore, releasing a certain amount of information in a published digital object may be reasonable given that it is information that can be easily obtained through other means. However, the reverse engineering of hardware may be expensive in terms of equipment and time, and so the manufacturer will want to ensure that a similar cost is levied against the purchaser for the digital equivalent.

It it possible to publish public information while securing private information, so that an automobile engine's overall performance can be modeled without exposing the internals of the dynamics and the parameter estimates. This can be done through "black box" approaches where the input-output behavior for a digital object is published, but the internal structure is obscured. Moreover, a company can make it impossible to see any model structure while allowing the model author to access the web site by providing input to its object. There is a wide range of possibilities from allowing others to copy the digital object and all of its internal models, to allowing users to copy only the highest level behavior, to allowing users to access only the behavior of the digital object without allowing any structural model access. Ultimately, the practice of a free marketplace will drive down the cost of digital objects and make them more accessible to everyone.

10. QUALITY CONTROL

What if the model author of the engine creates a digital engine that operates differently than the actual one? The automobile company could provide full access to an invalid model. We must have quality control measures in place to help us with this situation. The physical metaphor provides some help. Many consumer groups and institutions exist to protect consumers from bad products. Digital products will require similar groups and testing procedures. If a company knowingly markets a bad digital product, they will ultimately pay for this error in the marketplace. The digital object must be treated with the same level of quality control as the physical counterpart. In some cases, a company might make a mistake in production and a part or entire vehicle must be recalled. This type of recall is made easier with the digital product. It behooves the model authors to create valid, quality objects. It may be that anyone can publish a digital object but this is true of physical objects as well. The situation is somewhat more acute with a digital automobile since to create an automobile in the first place, one must have invested a fair amount of time and resources; however, a digital engine could be created by the neighbor down the street. One must learn to trust certain sources more than others based on past performance of prior digital objects. Also, we must have ways of verifying our sources, developers and producers with methods such as digital signatures, watermarks and encryption.

11. EXISTING STANDARDS

Standards are what make the post-industrial revolution work. Without standards for digital objects, we will be in the same situation as the rifle makers of the 19th century who made individual pieces without an assembly line or the benefits of reuse. There are no current standards for digital objects; however, there are movements in this direction.

The Unified Modeling Language (UML[3] [5]) is a potential standard for representing object oriented software components, not necessarily physical systems. Due to its breadth in an attempt to address software in general, it is missing dynamic and geometric specifications for physical objects. The basic approach in UML with visual class structures is fruitful, however, and represents an excellent step in representing and visualizing class and object structure.

The High Level Architecture (HLA[6]), created by the Department of Defense, is a detailed specification supporting distributed execution of simulation code. Code may be legacy code or could have been generated from models. What is missing from HLA is any description of dynamic or geometric *models*. HLA's focus is on tying together a set of

heterogeneous components, each of which may represent a physical simulator, personal computer or a simulation with human participants. Therefore, reusability is at the code level for software components, without an attempt to provide a standard for create the software from models. Reuse of software is a tremendous help to those who are looking for large-scale objects to plug into their simulations, but software reuse does not help the model author to construct the dynamics of an object itself.

The VLSI Hardware Definition Language (VHDL[7]) is a structural and behavioral language for representing integrated circuits. There is room in a VHDL specification for structure and layout as well as for behavior. It is made specifically for electronic applications, and does not attempt a broader purpose. We can look at the benefits and issues raised within the VHDL community to determine how these might affect digital object reusability beyond the VLSI domain.

12. MOOSE[8] AND RELATED WORK

Our goal is to create a formal specification for digital objects with class hierarchies, objects, geometric and dynamic models. There is no limitation as to the domain of application. The Multimodeling Object Oriented Simulation Environment (MOOSE[‡]) is a software implementation that accepts a specification in the form of a Distributed Modeling Markup Language (DMML). We have created our own Conceptual Modeling Language (CML) and Dynamic Modeling Language (DML) based on the multimodeling methodology, and we plan to use the Virtual Reality Markup Language (VRML[9]) specification for the geometric models. Regarding CML, both the HLA Object Model Template and UML class definitions suggest starting points for the CML grammar. Related work in the representation of classes, objects and models has been done by several groups. For example, Hill[10] focuses on the role of objects for simulation. Also, Zeigler's group[§] at the University of Arizona[11,12] and Mattsson[¶] at the Lund Institute of Technology in Sweden[13,14] have published widely in the area of object-oriented structures for simulation and control.

13. SUMMARY

If simulation is to make significant steps on the web, and in cost reduction, we need to move toward a digital object view of knowledge. If we take our telescopes and try to look into the future of modeling, we might be put off by the complexity of what lies ahead for digital objects. The idea that scientists and engineers will forever want to create their own models and simulations, without the ability to plug-and-play with digital objects represents an unfortunate situation. The model author might feel that no existing web objects can possibly match his requirements, however, we have demonstrated concrete steps that can be taken to alleviate the problems of reuse. Reusing digital objects may be a more profitable enterprise than for software components since digital objects bear a one-to-one relation with their physical cousins, and these physical objects have been already demonstrated in the marketplace to have real value. We feel that we must take proactive steps in making digital objects and their web-based representations a reality if simulation is to progress as a field. Nothing will happen overnight, but we need to seek out the really hard problems and then address them one by one.

ACKNOWLEDGMENTS

I would first like to thank all of the students who make MOOSE a reality. These students have contributed very significant amounts of time toward the digital object methodology and its implementation. Robert Cubert, Gyooseok Kim Youngsup Kim, and Kangsun Lee are all Ph.D. candidates—and core members comprising the MOOSE team—in the CISE Department within the University of Florida. I would like to thank the following funding sources that have contributed towards our study of modeling and implementation of the MOOSE multimodeling simulation environment: GRCI Incorporated (Gregg Liming) and Rome Laboratory (Steve Farr) for web-based simulation and modeling, as well as Rome Laboratory (Al Sisti) for multimodeling and model abstraction. We also thank the Department of the Interior under a contract under the ATLSS Project (Don DeAngelis, University of Miami). Without their help and encouragement, our research would not be possible.

[‡] `http://www.cise.ufl.edu/~fishwick/moose.html`.

[§] Ref. `http://www-ais.ece.arizona.edu/`.

[¶] `http://www.control.lth.se/~cace/`.

REFERENCES

1. P. A. Fishwick, D. R. C. Hill, and R. Smith, *1998 International Conference on Web-Based Modeling and Simulation*, Society for Computer Simulation International, San Diego, CA, 1998. 203pp.
2. P. A. Fishwick, J. G. Sanderson, and W. F. Wolff, "A Multimodeling Basis for Across-Trophic-Level Ecosystem Modeling: The Florida Everglades Example," *SCS Transactions on Simulation* , 1997. To be published.
3. P.-A. Muller, *Instant UML*, Wrox Press, Ltd., Olton, Birmingham, England, 1997.
4. R. C. Lee, *UML and C++: A Practical Guide to Object-Oriented Development*, Prentice Hall, 1997.
5. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, 1998.
6. "DoD High Level Architecture (HLA)," 1998. `http://hla.dmso.mil/`.
7. J. Bhasker, *A VHDL Primer: Revised Edition*, Prentice Hall, 1995.
8. R. M. Cubert and P. A. Fishwick, "MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework," *Simulation* , 1997. To be published.
9. B. Roehl, J. Couch, C. Reed-Ballreich, T. Rohaly, and G. Brown, *Late Night VRML 2.0 with Java*, Ziff-Davis Development Group, 1997.
10. D. R. C. Hill, *Object-Oriented Analysis and Simulation*, Addison-Wesley, 1996.
11. B. P. Zeigler, *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, 1990.
12. B. P. Zeigler, *Objects and Systems: Principled Design with Implementations in C++ and Java*, Springer Verlag, 1997.
13. S. E. Mattsson and M. Andersson, "Omola—An Object-Oriented Modeling Language," in *Recent Advances in Computer-Aided Control Systems Engineering*, M. Jamshidi and C. J. Herget, eds., vol. 9 of *Studies in Automation and Control*, pp. 291 310, Elsevier Science Publishers, 1993.
14. S. E. Mattsson, "Towards a New Standard for Modelling and Simulation Tools," in *SIMS'93, Applied Simulation in Industry — Proceedings of the 35th SIMS Simulation Conference*, T. Iversen, ed., pp. 1 10, SIMS, Scandinavian Simulation Society, c/o SINTEF Automatic Control, (Trondheim, Norway), June 1993. Invited paper.

Aesthetic Programming

Paul A. Fishwick

Department of Computer & Information Science & Engineering
University of Florida
Gainesville, Florida 32611, U.S.A.
fishwick@cise.ufl.edu

September 8, 2000

Abstract

When we marry traditional methods for computer programming with an artistic temperament, we give birth to a new phenomenon: the *aesthetic program*. Our work builds on visual approaches in programming as well as modeling for software, where we envision a gradual evolution from program to model. The need for the aesthetic model is strengthened with the importance of personalized, individually-tailored, models. We have formulated the *rube Project* methodology around the use of 3D web-based virtual world construction of models. Initial results suggest that these models are artistic, while containing sufficient symbolism and concise metaphoric mapping as to be executable on a computer.

1 Introduction

The merger of computer programming and art usually manifests itself under the rubric of terms such as algorithmic *X*, computer *X*, or digital *X*, where *X* is replaced by one of the arts, such as *art* or *music*. This naming trend denotes a qualification of art, or more particularly, how art is modified as a result of computer technology. However, if we reverse the word order to obtain "artistic computation," we find a relative dearth of information, with Sec. 4 overviewing the relevant literature. Software representations that lend themselves to an artistic representation tend to be minimalist and iconographic. More work needs to be done in inculcating the importance of the arts inside of the metallic computer box, and within the virtual spaces inhabiting the box. It's not just necessary that the box, monitor, and interaction devices be stylistic—we also need software that is aesthetic and of considerable sensory appeal. It is only by doing so, that we can reinvent programming

to be more humane, and interestingly, to become closer to traditional engineering which has historically managed to forge alliances with style, as in the field of architecture.

When one thinks of computer programs in their complex and cryptographic text-based glory, the idea of aesthetics does not generally come to mind. Knuth [26] authored a classic series of books entitled *The Art of Programming*, and later created an approach to text-based software development termed *literate programming*[27], in which programmers develop more readable programs that, in themselves, serve as complete textual, typeset documentation. For the most part, software is text-based, and its typical incarnations seem incongruous with the sorts of productions commonly found in fine art. Why are programs limited to text, and what directions can we follow to pursue path of artistic programs? To make software artistic, we need to endow software with some form of *aesthetics*. We do not delve into the complexities of aesthetics other than to suggest that an aesthetic direction implies one that yields art as a process and product. Rutsky [46] asserts that in "high tech," in which programming certainly finds a home, "technology becomes much more a matter of representation, of aesthetics, of style."

Aesthetic components and concepts are trying to make themselves known if the use of metaphor and analogy is an indication. Computing and software incorporate numerous metaphors [30, 29]. When we talk of program components, we speak of "looping around a section," "walking through code," "piping X into Y," "forking off processes (in Unix)," and "calling a sub-routine." Often, a mixed set of metaphors is loosely applied at the level of natural language, without making them concrete. For example, in Unix shells, one can pipe, fork, redirect and place jobs in the background. Little effort has been made to formalize and visualize these metaphoric constructs, since to do so would introduce extra overhead which was undoubtedly deemed prohibitively expensive when Bell Labs first gave birth to Unix in the late 70s. Interfacing with text would have to suffice. Most of these metaphors must live in the mental models [19] of their programmers' minds since they are rarely surfaced in a visible manner. While mental models represent important constructs, we should endeavor to free them from our minds, and to surface them as real and virtual models, which have sensory and aesthetic qualities. Currently, most programming metaphors are trapped inside complex software representations and in individual mental models, which cannot easily be communicated. There are key aspects of software that rely on metaphor—namely, programming structures that have no obvious real-world equivalent. For example, states and events are conceptual, and so require metaphor to clothe them in aesthetics. In a distributed system, one might represent an ATM machine as something that looks similar to a real one, and yet ATM states and events do not have transparent, concrete equivalents. Fortunately, this situation opens the door to applying metaphor with a great deal of flexibility. A state can be a circle, sphere, plot of land, architectural space, or a partition in space-time.

As with most new techniques, the creation of artistic software and computing finds its first home in science fiction. In a November 1976 BBC episode of Dr. Who, enti-

tled "The Deadly Assassin," the Doctor battles the Master in "the Matrix" which is a repository for all Time Lord knowledge [12, 13]. Similar *matrix* centered novels detailed the methods of cyberspace and virtual world manipulation, such as Gibson's Neuromancer [21] and Stephenson's Snowcrash [50]. These ideas culminated recently in the feature film "The Matrix", written and directed by the Wachowski brothers. Perhaps the longest running example of a matrix is found in the Holodeck of "Star Trek: The Next Generation" by Paramount Pictures. The concept of *program* in these media projects hints at programs as virtual objects and spaces, without there being anything formal or specific. However, it is hard to imagine Java being integral to programming the Holodeck; the ability to easily and quickly create 3D spaces suggests an altogether different paradigm for software development. Disney's film "Tron" presaged, not only the idea of a communicable, digital matrix, but more particularly, the virtual embodiment of aesthetic software. Users were represented by people who were similar to avatars in today's virtual space and software agent terminology.

2 From Program to Model

Programs have differing levels of detail, and translation among levels. The lowest level program is microcode. Continuing up the ladder of comprehensibility, we obtain assembly language and then hundreds of programming languages. Table 1 is pseudo-code that would easily map to one of these languages. This program is designed to allow people

Table 1: Algorithm for a Text Editor

1:	*Main program entry/exit screen displayed*
2:	*If any key is pressed go to 3, otherwise go to 2*
3:	*Enter text mode*
4:	*Process text entered by user*
5:	*If key ESC is pressed, go to 7*
6:	*If key ^M is pressed, go to 11, otherwise, go to 4*
7:	*Enter command mode*
8:	*Process commands entered by user*
9:	*If key ^X is pressed, go to 1*
10:	*If key Q is pressed, go to 3, otherwise go to 8*
11:	*Enter macro mode*
12:	*Process macro entered by user*
13:	*If key ENTER is pressed, go to 7, otherwise, go to 12*

to edit text on a display monitor. This type of program is called a *text editor*. *Emacs*, *vi*, and *NotePad* are example text editors. Our program design begins with a *Main* entry/exit

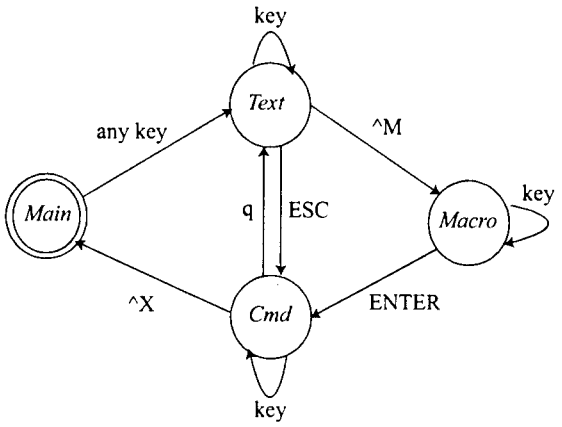
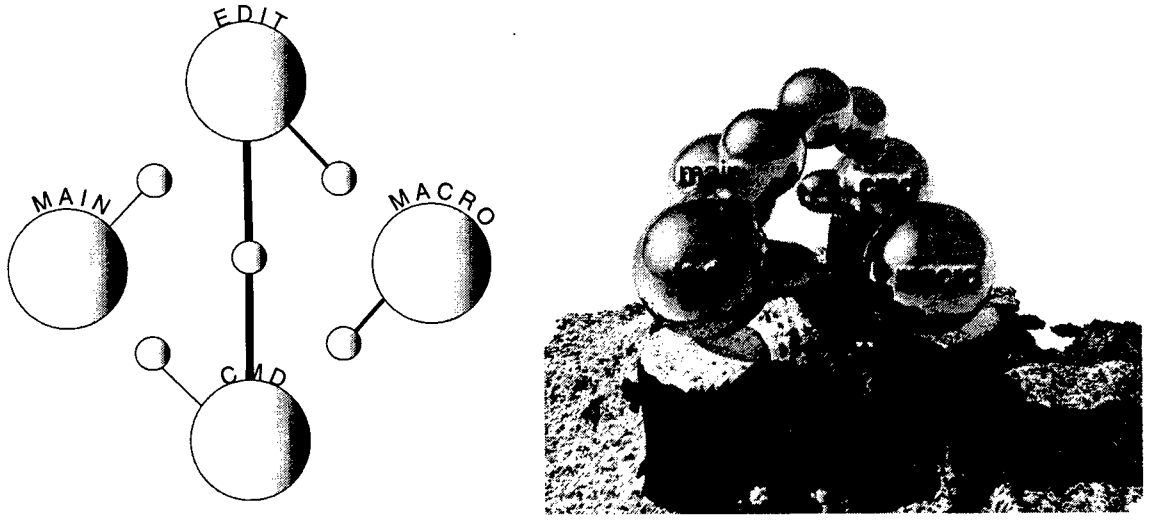


Figure 1: FSM with 4 states modeling a text editing program.

screen. From this screen, we transition into one of two modes: *Text* and *Cmd* (i.e., command). In text mode, the human enters text, as with a typewriter. In command mode, the user enters commands as to what to do with the text, such as cut,copy, and paste. There is a special *Macro* mode where a sequence of editing commands and data can be bound to an arbitrary key. Keys are those commonly found on a keyboard: ^ M means control-M, ESC references the escape key. Fig. 1 displays a two-dimensional graphic that represents a Finite State Machine (FSM). Circles are states, and directed arcs are transitions from one state to another based on conditions that are labeled adjacent to each arc. The machine begins in the start state *Main*. The machine stays in *Main* until a key is pressed. Fig. 1 is a model of the software in Table 1, with the term "model" being defined, typically, as a visual representation of a program or system. Fig. 1 is a dynamic model [14] since its components reflect the behavior of a system, as opposed to its physical or information-based structure. To the extent that art targets human senses, Fig. 1 shows promise over Table 1, and yet Fig. 1 yields a fairly iconic, diagrammatic display largely devoid of texture, sound and aesthetic content. Fig. 1 is a common diagram for the computer scientist, and most will recognize this form.

Figs 2(a) and 2(b) demonstrate two additional models for the editor software. The mapping of state and transition in Fig. 2(a) is sufficiently close to Fig. 1 that the mapping may be intuitively grasped. However, Fig. 2(b) is somewhat different. A *metaball* modeling approach was used to created oblong, directed spheres and a special welded join when a bidirectional set of transitions exist between states *text* and *cmd*. A landscape with towering butte-like structures is placed underneath the program, where each butte

serves to demarcate a state. In these models, there are some notable differences with re-



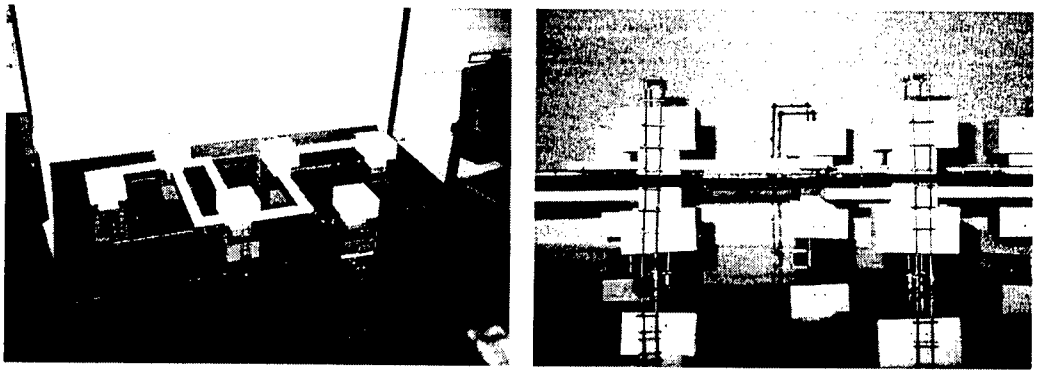
(a) 2D FSM

(b) 3D FSM

Figure 2: 2D and 3D model designs for the text editor

spect to Fig. 1, which is a *static* form. Figs 2(a) and (b) are manifested as *dynamic* graphical user interfaces so that not all information need be captured in a single perspective. By using the mouse and keyboard, and general navigational methods, additional models are viewed when interacting with the two models in Fig. 2. For example, in Fig. 2(a), the connecting lines change their width over time, designating the frequency of state transitions. In Fig. 2(b), state names only appear in response to a "mouse-over" event on the states. Fig. 1 fits within the *culture* and educational mindset of computer scientists, whereas new metaphoric mappings must be learned for Fig. 2. Nevertheless, it is common in modeling and programming for scientists to shop around for model types and metaphors and select the groups that best fit their cultural requirements. Modeling cliques are identifiable by the conferences they hold and journals that they edit; the scientists become adapted to a particular paradigm [28] to which they hold dear and find useful for their purposes. There is no singular best modeling approach. Fishwick [14] defines numerous formalisms and 2D designs for dynamic models.

Ugrankar [52] created a 3D architectural physical scale model of a virtual space designed to encode a six-state FSM [14] that represents a dynamic model of water temperature dynamics in response to heat conduction from a plate. The left-hand most 4 states are identical in topology to Fig. 1, even though the semantics differ. This illustrates the



(a) Overhead view

(b) Closeup sideview of states.

Figure 3: Architectural space for 6-state FSM.

powerful role of analogy in systems. Fig. 3(a) shows an overhead view of the model. The model has cubical rooms and long corridors connecting rooms. The entire space is divided horizontally by a glass pane. Rooms, representing states, are present on both sides of the pane. Corridors on the top-half of the pane represent external state transitions, whereas corridors on the underside are internal state transitions. For temperature dynamics, internal transitions are those that occur because of a change of internal system state (i.e., temperature reaching a value and thus indicating a phase boundary), whereas external transitions occur due to causes external to the system (i.e., a knob being turned). Fig. 3(b) shows a closeup illustrating metal ladder-like constructs denote reflexive transitions where the dynamics results in the same state given the appropriate conditions. Conditions are marked with tape on each corridor. Avatars, that are not present in the physical model and which represent signals into the system, move along the corridors. This model was designed and created as a prototype for a virtual world using the Virtual Reality Modeling Language (VRML), which is our primary design language for *rube* worlds.

3 The *rube* Project

In 1998 at the University of Florida, our research group initiated a methodology called *rube* [45, 16, 15], which helps modelers to construct models that incorporate aesthetics.

The models are used to both model physical phenomena as well as to design programs designed via modeling. The following are the *rube* modeling steps:

1. *Choose system to be modeled:* In the case of a modeling initiative, a system is initially specified. For example, we know that we must model two trophic levels in the Everglades [17] if that is our physical target. In the case of software, there are two sub-choices. The first is one where we have a distributed system, where the model structure reflects the physical system component topology. The second choice is where we have a program to create with no apparent objects. These two sub-choices are usually part of the same software since even though a distributed system suggests objects and models, large chunks of software will require innovative design and frequent use of metaphor if we are to create models from them.

2. *Select model types:* We take the software and specify the formal dynamic model types to be used. Dynamic model types are plentiful [14] and include automata, Petri nets, data flow networks, scripts, rules, and event graphs. This is the formal step of defining the nature of the dynamics in base model form—prior to exercising our desire to employ metaphor and aesthetics.

3. *Choose a style:* Elements from the previous step must be mapped, via metaphor. The style may be hierarchical. For example, the top level style could be architecture, and the second level style to reflect a particular type of architecture such as Classic Greek, Gaudi or Le Corbusier. If the style is painting, we could have Russian constructivist sketching or surrealistic painting in the style of Ernst, Dali or Magritte.

4. *Define Mapping:* Once we have defined a style, we must now carefully, and completely, create a mapping between the formal dynamic model type components and the stylistic components. For example, a collage sub-element of Ernst would map to a *place* of a Petri net if we had chosen the Petri net as our model type. It is critical to pay special attention in ensuring that the mapping preserves the full capacity of the formal dynamic model, otherwise the model will not be executable.

5. *Create Model:* This is the craft-worthy step of applying the mapping to create the model.

Regarding model-creation from programs, we now address the question of what programs are modeling. In the case of program construction for distributed systems, models are created of the objects and network composing the system. In the case of the non-distributed, ordinary program, we are free to create maps using the above methodology, by starting with model types. Here are some guidelines based on programming elements:

- *Sequence:* Open path in space-time. Examples: (1) entity movement across paper following arc; (2) path through building; (3) evolution of entity or space over passage of time.
- *Condition:* Branch in space-time. Examples: (1) fork in a road; (2) portals leading from a room; (3) plant and tree branching.
- *Iteration:* Closed path in space-time. Examples: (1) race track; (2) closed circuit on a landscape.
- *Hierarchy:* Space-time hierarchy. Examples: (1) Scaling in space-time using physical encapsulation; (2) Positioning levels spatially via horizontal or vertical displacement.
- *Recursion:* Self-referential Space-time hierarchy. Examples: (1) Zooming in and out of a point in space, or inside of an entity to find smaller, self-similar entities.
- *Assignment, Input and Output:* Human interaction using a sensor or tool. Examples: (1) measuring an entity; (2) using a tool to shape or color an entity.

These guidelines are meant to be very general. We have just begun to explore approaches to mapping software to physical phenomena, therefore, these guidelines are purely heuristic in value. The categories reflect basic mathematical categories as well as broadly defined cognitive partitions, similar to those described by Arnheim [2, 3], Volk [55], and Ching [10]. They also roughly correspond to principles in programming languages [20]. *Space-time* is a multidimensional space with an orthogonal time axis, as in physics. Iteration and recursion are generally coupled with sequence and condition, and should be mixed where appropriate. One such example involves creating a hierarchy where loops are present, so that where a loop is located, a hierarchy is created through an encapsulating object. Consider a routine sorting or searching task, which tend to be highly iterative. Such a task can be wrapped into a physical *machine* that reminds us of the task. This wrapping represents the introduction of hierarchy.

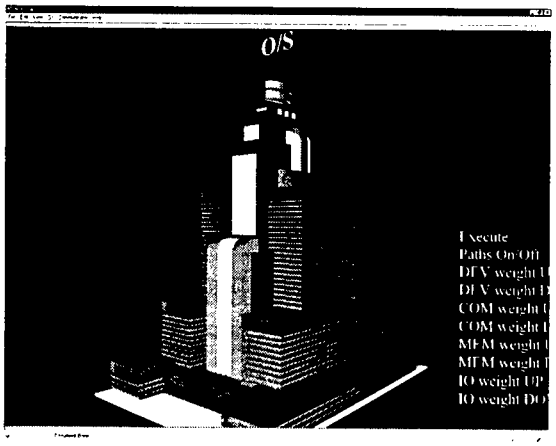
Hopkins et al. [23] constructed an operating system kernel using VRML. An operating system is one of the largest pieces of software in the typical computer. It controls all peripherals and resources, allowing tasks to request resources, obtain service, and continue processing. A simplification of an operating system is where we model the average task (i.e., executable program) as requesting the Central Processing Unit (CPU), and then requesting a resource such as I/O or memory, and then iterating in this fashion until the program terminates. We employ a metaphor based on business workflow where tasks, represented as human agents, move on the floor of a building but stay within the boundaries of assigned tracks. This is not unlike waiting lines with guide posts and ropes, or colored strips placed on an airport floor, as a means of partitioning waiting lines. Our

example task scheduler [23] is a non-preemptive, dynamic priority scheduling system that contains tasks, four priority queues, and the following five types of physical devices with their associated queues: 1) CPU, 2) DEV (i.e., external device), 3) COM (i.e., communication, such as via the parallel, serial, and USB ports), 4) MEM (i.e., memory load/store), and 5) I/O (i.e., input/output, such as disk read/write). By employing a workflow metaphor, we map a task to a person and a device to a "service facility," which is a person behind a desk (see Fig. 4(d)). The priority and device queues map directly to physical space as waiting lines. Persons can travel over paths between the devices and queues.

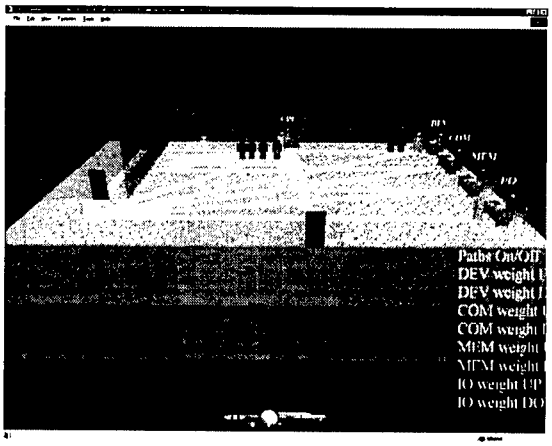
Fig. 4 displays different angles of the VRML world created with the [45] methodology. Fig. 4(a) starts us outside of the Operating System, which is constructed as an art deco building. On one of the floors, we find the task scheduling "workflow" identified by moving agents (see Figs. 4(b) through 4(d)). We are now concerned with the details of implementing the above task scheduler along with its metaphors in a three-dimensional environment. In the context of VRML, we create prototype (PROTO) nodes for each of the reusable items.

4 Related Work

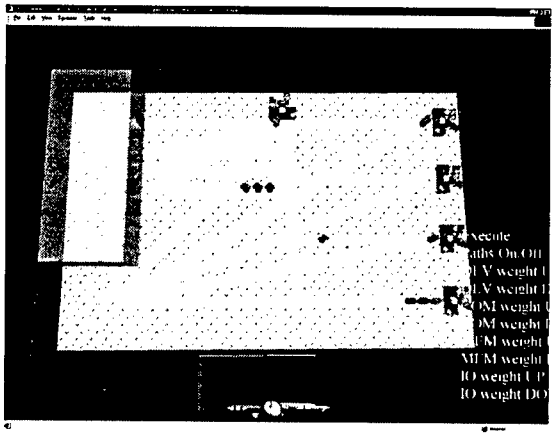
The Unified Modeling Language (UML) [6] represents a good beginning in viewing software as modeling, but this view centers around software engineering, rather than programming. Programming is typically viewed as a low-level activity underneath the umbrella of software engineering. This view should change if we are to more clearly represent programs as models, while relegating textual programs to the status currently occupied by assembly language—a necessary, but low level construct. Many languages targeted at novices [40] are model-based. The Logo language [41] was one of the first languages based on the idea of programming through the use of a turtle [1, 44] capable of carrying out a set of simple instructions, with graphical feedback for output. Karel the Robot [42] has similar aims with a robot replacing the turtle with a robot, and by extending the functionality of the moving agent with respect to its interaction environment. Methods of programming by demonstration or example [32, 11] are structured on agent and rule-based approaches to software development. The idea is to program through modeling, and where there may not be a natural physical object in a program, a microworld can be created so that the programmer benefits from thinking in terms that are natural, memorable, and aesthetic rather than artificial, arcane and ugly. Several commercial products exist, such as *Stagecast* [48], which allow modelers to create simulations using graphically-specified production rules. Modeling is the activity that allows humans to better reason about programs. The overall areas of *software visualization* [49] and *visual programming* [24] have produced many similar successes to those of program-



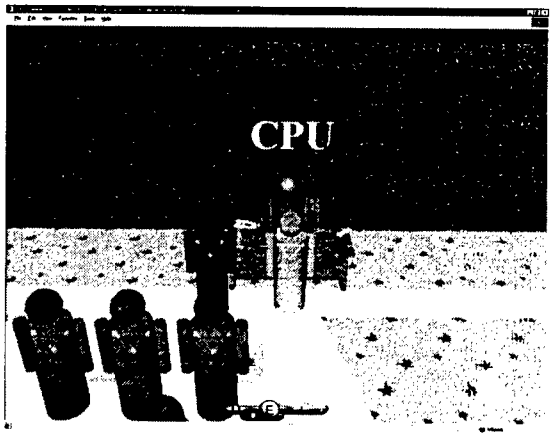
(a) View of operating system.



(b) Isometric view of operating system task scheduling.



(c) Overhead view of Fig. 4(b).



(d) Snapshot and zoom of CPU in action.

Figure 4: Views of the the operating system from outside and on the task scheduling floor.

ming by modeling, example and demonstration. There are areas where the importance of visualization and sensory-feedback are stressed, such as Human-Computer Interaction (HCI) and Visualization. In HCI, we obtain example research involving visualizing information [8] and texts on interface methodology [38, 37, 43]. Employing visualization and metaphors permits the human to better interact with the computer. For visualization, there has been significant work in visualizing data, program execution and software. *Data visualization* is, perhaps, the most active field where scientific and engineering data are viewed from multiple perspectives, in 2D and 3D, using a wide variety of icons and color range. Brown [7] discusses methods for visualizing the execution of programs in terms of input/output. Shu [47] specifies a dichotomy where we have *visual environments*, referring to program and data visualization, versus *visual language*, referring to the actual creation of software using visual methods. Early visual software developments were catalogued in an edited volume by Chang [9] and recently in a special issue of Communications of the ACM [32] on "programming by example" where the goal is to make programming easier through simulation and demonstration via rules.

A quick search through the literature on keywords "3D" and "programming" tends to sprout forth numerous articles and books on how to render 3D graphics. By turning this idea on its head, we imagine that 3D itself is used to do the programming, rather than vice versa. The area of 3D program structure is closely related research to ours and represents a relatively new area that holds much promise, especially with new 3D web-based technologies such as Java3D and VRML. Programming in 3D had to wait for efficient methods for 3D programming, but significant work has been done. Lieberman [31] pioneered one of the first efforts in transitioning from 2D programming to using 3D elements. Najork [36] created the *Cube* language, with cubes representing program nodes and pipes for flow. Oshiba and Tanaka [39] built *3D-PP*, which contains regular polyhedra connected with lines for representing declarative, logic-based programs.

In reviewing *algorithmic art*, representing the former concept of software creating art, there are many players. Cohen's AARON [34] was a system for semi-automatic generation of paintings. Verostko [54] discusses the interrelationships beteween art and software from the standpoint of one (programming) being used to create the other (art). In an essay called *Programming as Art* [53], Verostko states "As we shall see, an algorithm may be an instruction for any kind of procedure. For the artist or composer this means that any kind of algorithmic inquiry on the nature of form is possible." The artist uses the computer program as any other tool. More mechanical, and less organic, procedures for creating art and form also exist [51, 35]. These formal methods tend to capture pattern-based generation of art using tools familiar to the computer scientist, such as production rules and automata. Maeda's *Aesthetics + Computation Group* [22] focuses on interactive demonstrations of this bridge area, often using Java applets for the underlying technology [33]. Gelernter [18] presents cogent arguments for alternate, and beautiful, representations for machines and computing. The emphases on HCI and software visualization, with a focus

on the human interface, strikes a common chord with art. Differences come about mainly in the degree of creativity and metaphor employment afforded. An approach to visualizing arbitrary data or programs may involve a color landscape, but rarely does one see a rich landscape filled with color and texture. There is an element of artistic minimalism in these approaches, and this is in line with similar concerns in graphic design [5], which are driven primarily by efficiency.

5 Summary

Today's program in Java or C++ bears little resemblance to Fig. 4, and there are good reasons for this condition. Even though the state of the art, and information economy, fertilize the roots for ubiquitous 3D development, we have many legacy codes and tools, and the tools are nowhere near the point of general acceptance for aesthetic programming. A lot of further research is required, along with a gameplan for getting from A to B. Our work depends on a modeling framework, and even though significant efforts such as UML exist, free tools are not available as they are for Java, C and C++.

As if the problems of using graphical methods in programming were not enough, we also have an issue in education of computer science students who grow up with text-based programming. Making the leap from fairly minimalistic, diagrammatic models and code to representations that encourage massive infusions of art is not going to be easy. The educational challenges are paramount, and represent a cultural gap for most computer scientists who view artistic representations as flourishing, syntactically excessive, and luxurious. The cost of convincing visual and auditory renderings is ever-decreasing, causing a revolution in the way that future generations expect to interact with the world. The cultures must be bridged and connected if aesthetic software is to succeed. In preparation for the game-console generation to enter University, and for their expectations of immersive environments, at the University of Florida we have created a new set of engineering programs in unison with our College of Fine Arts. These programs are called *Digital Arts and Science* [4] (DAS), and their aim is to educate a new breed of student who is as familiar with sketching, textures, sculpture, form and music as they are with data structures, discrete math, and translation theory. Our vision is that these students will have the aesthetic sensibilities to take advantage of the rapid technologies that now support fast 2D/3D graphics and audio.

Even though we began our studies with model construction with an extra dimension, it became increasingly clear that the primary thrust of this work was in aesthetics, since one can equally construct flat models that have meaningful and desirable qualities. Our modeling examples are still primitive, and no doubt professional artists can weave far greater patterns. Even when we consider 3D programming, work has been done before in this area, but what remains to be done for the future is tantamount to

a renaissance in modeling—to free models, not just from flatland, but also from simple geometries. Much of the work in 3D programming languages, for example, implements 3D iconography, which needs to evolve into landscapes, cities, organic systems, physical architectures, and style-guided formations if the very idea of aesthetics is to take hold—the point being that it *does* matter whether one uses a octagonal polyhedron versus an art deco house. One is aesthetically-challenged and Platonic whereas the other promotes familiar sensory appeal.

The study of aesthetic models in the representation of computer programs represents a small potential when compared with the more encompassing subject area of computer science. Models are used frequently in most sub-areas: databases, machine organization, organization and architecture, data and program structures, and artificial intelligence. These areas will all benefit from direct artistic influence. The same approach to aestheticism can be applied to other art forms, such as storytelling and theatre. Stories can be mapped onto model structures, and serve to entertain as well as to educate and remind us about the target system. Our implementation on models for aesthetic computing is directly inline with Alan Kay's definition of computer literacy [25]:

> "Computer literacy is a contact with the activity of computing deep enough to make the computational equivalent of reading and writing fluent and enjoyable. As in all the arts, a romance with the material must be well under way. If we value the lifelong learning of arts and letters as a springboard for personal and societal growth, should any less effort be spent to make computing a part of our lives?"

Acknowledgments

I am indebted to my students and sponsors. I would like to thank students, past and present, for their continued efforts in *rube*: Robert Cubert, Andrew Reddish, John Hopkins, and Linda Dance. I would also like to thank research sponsors of our modeling and simulation work: Air Force/Rome Laboratory (contract: F30602-98-C-0269) and the Department of the Interior (grant 14-45-0009-1544-154).

References

[1] Harold Abelson and Andrea diSessa. *Turtle Geometry*. MIT Press, 1980.

[2] Rudolf Arnheim. *Visual Thinking*. London, Faber and Faber, 1969.

[3] Rudolf Arnheim. *The Dynamics of Architectural Form*. University of California Press, 1977.

[4] Digital Arts and Science Programs. http://www.cise.ufl.edu/fdwi, 1999.

[5] Jacques Bertin. *Semiology of Graphics: Diagrams, Networks, Maps.* University of Wisconsin Press, 1983.

[6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[7] Marc H. Brown. *Algorithm Animation.* MIT Press, 1987.

[8] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in Information Visualization: Using Vision to Think.* Morgan Kaufman, 1999.

[9] Shi-Kuo Chang, editor. *Visual Languages and Visual Programming.* Plenum Press, 1990.

[10] Francis D. K. Ching. *Architecture: Form, Space and Order.* Van Nostrand Reinhold Co., 1979.

[11] Allen Cypher, Daniel C. Halbert, David Kurlander, and Ellen Cypher, editors. *Watch What I Do: Programming by Demonstration.* MIT Press, 1993.

[12] Terrance Dicks. *Doctor Who and the Deadly Assassin.* British Broadcasting Corporation, 1977.

[13] The Deadly Assassin. *Doctor Who Magazine,* (108):44–47, January 1986.

[14] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall, 1995.

[15] Paul A. Fishwick. A Modeling Strategy for the NASA Intelligent Synthesis Environment. *Journal of Space Mission Architecture (JSMA),* (1):23–42, 1999. Center for Space Mission Architecture and Design, Jet Propulsion Laboratory.

[16] Paul A. Fishwick. 3D Behavioral Model Design for Simulation and Software Engineering. In *2000 Web3D/VRML Conference,* pages 7–16, February 2000.

[17] Paul A. Fishwick, James G. Sanderson, and Wilfried F. Wolff. A Multimodeling Basis for Across-Trophic-Level Ecosystem Modeling: The Florida Everglades Example. *SCS Transactions on Simulation,* 15(2):76–89, June 1998.

[18] David Gelernter. *Machine Beauty: Elegance and The Heart of Technology.* Basic Books, 1998.

[19] Dedre Gentner and Albert Stevens. *Mental Models.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1983.

[20] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley, second edition, 1987.

[21] William Gibson. *Neuromancer*. Ace Books, 1984.

[22] Aesthetics + Computation Group. http://acg.media.mit.edu/.

[23] John F. Hopkins and Paul A. Fishwick. Synthetic Human Agents for Modeling and Simulation. *Proceedings of the IEEE*, 2000. Submitted for publication.

[24] Tadao Ichikawa, Erland Jungert, and Robert R. Korfhage, editors. *Visual Languages and Applications*. Plenum Press, New York, 1990.

[25] Alan Kay. Computer Software. *Scientific American*, 251(3):53–59, September 1984.

[26] Donald Knuth. *The Art of Programming: Volumes 1 to 3*. Addison-Wesley, 1968.

[27] Donald Knuth. *Literate Programming*. Stanford University Center for the Study of Language and Information (Lecture Notes, No. 27), 1992.

[28] Thomas Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 3rd edition, 1996.

[29] George Lakoff. *Women, Fire, and Dangerous Things: What Categories Reveal about the Mind*. University of Chicago Press, 1987.

[30] George Lakoff and Mark Johnson. *Metaphors we Live By*. University of Chicago Press, 1980.

[31] Henry Lieberman. A Three-Dimensional Representation for Program Execution. In E. P. Glinert, editor, *Visual Programming Environments: Applications and Issues*. IEEE Press, 1991.

[32] Henry Lieberman. Programming by Example. *Communications of the ACM*, 43(3):73–74, March 2000.

[33] John Maeda. *Design by Numbers*. MIT Press, 1999.

[34] Pamela McCorduck. *Aaron's Code: Meta-art, Artificial intelligence and the Work of Harold Cohen*. W. H. Freeman, 1991.

[35] William Mitchell. *The Logic of Architecture: Design, Computation and Cognition*. MIT Press, 1990.

[36] Marc Najork. Programming in Three Dimensions. *Journal of Visual Languages and Computing*, 7(2):219–242, June 1996.

[37] Jakob Nielsen. *Usability Engineering*. Morgan Kaufman, 1993.

[38] Donald A. Norman. *The Design of Everyday Things*. Doubleday Books, 1990.

[39] Takashi Oshiba and Jiro Tanaka. 3D-PP: Visual Programming System with Three-Dimensional Representation. In *International Symposium on Future Software Technology (ISFST '99)*, pages 61–66, 1999.

[40] John F. Pane and Brad A. Myers. Usability issues in the design of novice programming systems. Technical report, Carnegie Mellon University, 1996. Report CMU-CS-96-132, http://www.cs.cmu.edu/~pane/ftp/CMU-CS-96-132.pdf.

[41] Seymour Papert. *Children, Computers and Powerful Ideas*. Basic Books, New York, 1980.

[42] Richard E. Pattis, Jim Roberts, and Mark Stehlik. *Karel the Robot: A Gentle Introduction to the Art of Programming*. John Wiley and Sons, 1994.

[43] Jef Raskin. *The Humane Interface*. Adisson-Wesley, 2000.

[44] Mitchel Resnick. *Turtles, Termites and Traffic Jams: Exporations in Massively Parallel Microworlds*. MIT Press, 1997.

[45] rube Project. http://www.cise.ufl.edu/~fishwick/rube, 1998.

[46] R. L. Rutsky. *High Technē: Art and Technology from the Machine Aesthetic to the Posthuman*. University of Minnesota Press, 1999.

[47] Nan C. Shu. *Visual Programming*. Van Nostrand Reinhold Company, 1988.

[48] Stagecast Software. http://www.stagecast.com.

[49] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1998.

[50] Neil Stephenson. *Snowcrash*. Spectra Books, 1993.

[51] George Stiny and James Gips. *Algorithmic Aesthetics: Computer Models for Criticism and Design in the Arts*. University of California Press, 1978.

[52] Prajakta Ugrankar. Finite State Automaton for Boiling Water Multimodel, May 2000. Independent Study (University of Florida).

[53] Roman Verostko. http://www.verostko.com.

[54] Roman Verostko. Epigenetic Painting. *Leonardo*, 23(1):17–23, 1990.

[55] Tyler Volk. *Metapatterns: Across Space, Time and Mind*. Columbia University Press, 1995.

3D Behavioral Model Design for Simulation and Software Engineering*

Paul A. Fishwick

University of Florida

November 24, 1999

Abstract

Modeling is used to build structures that serve as surrogates for other objects. As children, we learn to model at a very young age. An object such as a small toy train teaches us about the structure and behavior of an actual train. VRML is a file standard for representing the structure of objects such as trains, while the behavior would be represented in a computer language such as ECMAScript or Java. VRML is an abbreviation for Virtual Reality Modeling Language [2], which represents the standard 3D language for the web. Our work is to extend the power of VRML so that it is used not only for defining shape models, but also for creating structures for behavior. "Behavior shapes" are built using metaphors mapped onto well-known dynamic model templates such as finite state machines, functional block models and Petri nets. The low level functionality of the design still requires a traditional programming language, but this level is hidden underneath a *modeling* level that is visualized by the user. We have constructed a methodology called *rube* which provides guidelines on building behavioral structures in VRML. The result of our endeavors has yielded a set of VRML Prototypes that serve as dynamic model templates. We demonstrate several examples of behaviors using primitive shape and architectural metaphors.

1 INTRODUCTION

One physical object captures some information about another object. If we think about our plastic toys, metal trains and even our sophisticated scale-based engineering models, we see a common thread: to build one object that says something about another—usually larger and more expensive—object. Let's call these objects the *source object* and the *target object.* Similar object definitions can be found in the literature of metaphors [6] and semiotics [10]. The source object *models* the target, and so, modeling represents a relation between objects. Often, the source object is termed *the model* of the target. We have been discussing scale models identified by the source and target having roughly proportional geometries. Scale-based models often suffer from the problem where changing the scale of a thing affects more than just the geometry. It also affects the fundamental laws applied at each scale. For example, the hydrodynamics of the scaled ocean model may be different than for the real ocean. Nevertheless, we can attempt to adjust for the scaling problems and proceed to understand the larger universe through a smaller, more manipulable, version.

Our task is to construct a software architecture that encourages 3D construction of objects and their models. Our focal point is the *behavioral model* where one defines the behavior or dynamics of an object using another set of objects. This sort of modeling may be used in representing the models of large-scale systems and software in the case where models have been used to specify the requirements and program design [1, 11]. We call this modeling architecture *rube.* An example use of *rube* is defined in the Sec. 2, and the methodology of *rube* in Sec. 3. Facets of the VRML implementation are defined in Secs. 4, 5 and 6. We close the paper with philosophical issues on the art of modeling in Sec. 7 and the summary in Sec. 8.

2 NEWELL'S TEAPOT

In the early days of computer graphics (c. 1974-75), Martin Newell rendered a unique set of Bézier surface spline patches for an ordinary teapot, which currently resides in the Computer Museum in Boston. The teapot was modeled by Jim Blinn and then rendered by Martin Newell and Ed Catmull at the University of Utah in 1974. While at this late date, the teapot may seem quaint, it has been used over the years as an icon of sorts, and more importantly as a benchmark for all variety of new techniques in rendering and modeling in computer graphics. The Teapot was recently an official emblem of the 25th anniversary

*Department of Computer & Information Science and Engineering, P.O. Box 116120, Gainesville, FL 32611, Email: fishwick@cise.ufl.edu

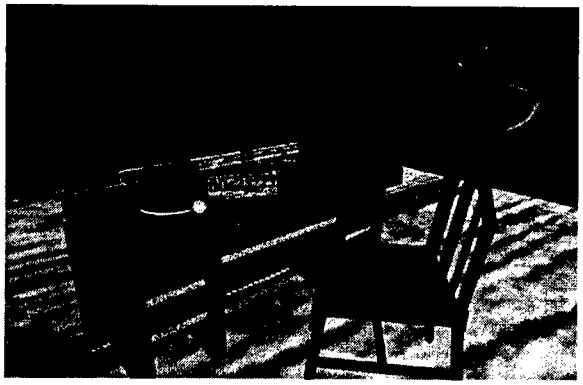


Figure 1: Office scene with Newell Teapot, dynamic model and props.

of the ACM Special Interest Interest Group on Computer Graphics (SIGGRAPH).

Since 1989 at the University of Florida, we have constructed a number of modeling and simulation packages documented in [4, 3] In late 1998, we started designing *rube*, named in dedication to Rube Goldberg [8], who produced many fanciful cartoon machines, all of which can be considered *models* of behavior. One of our goals for *rube* was to recognize that the Teapot could be used to generate another potential benchmark—one that captured the entire teapot, its contents and its models. The default teapot has no behavior and has no contents; it is an elegant piece of geometry but it requires more if we are to construct a fully *digital teapot* that captures a more complete set of knowledge. In its current state, the teapot is analogous to a building façade on a Hollywood film studio backlot; it has the shape but the whole entity is missing. In VRML, using the methodology previously defined, we built TeaWorld in Fig. 1. We have added extra props so that the teapot can be visualized, along with its behavioral model, in a reasonable contextual setting. The world is rendered in Fig. 1 using a web browser. *World* is the top-most root of the scene graph. It contains a *Clock*, *Boiling_System*, and other objects such as the desk, chairs, floor and walls. The key fields in Fig. 2 are VRML nodes of the relevant field so that the *contains* field refers to multiple nodes for its value. This is accomplished using the VRML *MFNode* type. The hierarchical VRML scene graph for Fig. 1 is illustrated in Fig. 2. The scene contains walls, a desk, chair and a floor for context. On the desk to the left is the teapot which is filled with water. The knob controlling whether the teapot heating element (not modeled) is on or off is located in front of the teapot. To the right of the teapot, there is a pipeline with three machines, each of which appears in Fig. 1 as a semi-transparent cube.

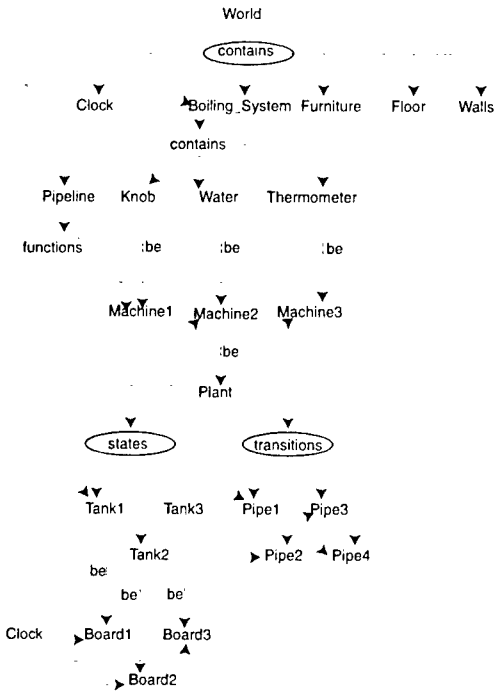


Figure 2: VRML Scene Graph for the Teapot and its models.

Each of these machines reflects the functional behavior of its encapsulating object: *Machine1* for *Knob*, *Machine2* for *Water* and *Machine3* for *Thermometer*. The *Thermometer* is a digital one that is positioned in *Machine3*, and is initialized to an arbitrary ambient temperature of 0° C. Inside *Machine2*, we find a more detailed description of the behavior of the water as it changes its temperature as a result of the knob turning. The plant inside *Machine2* consists of *Tank1*, *Tank2*, *Tank3*, and four pipes that move information from one tank to the next. Inside each tank, we find a blackboard on which is drawn a differential equation that defines the change in water temperature for that particular state. The following modeling relationships are used: *Pipeline* is a Functional Block Model (FBM), with three functions (i.e., machines); *Machine* is a function (i.e., semi-transparent cube) within an FBM; *Plant* is a Finite State Machine (FSM) inside of Machine 2; *Tank* is a state within a FSM, and represented by a red sphere; *Pipe* is a transition within a FSM, and represented by a green pipe with an attached cone denoting direction of control flow; and *Board* is a differential equation, represented as white text. The following metaphors are defined in this example. The three cubes represent a sequence of machines that create a pipeline. One could have easily chosen a factory floor sequence of numerically controlled machines from the web and then used this

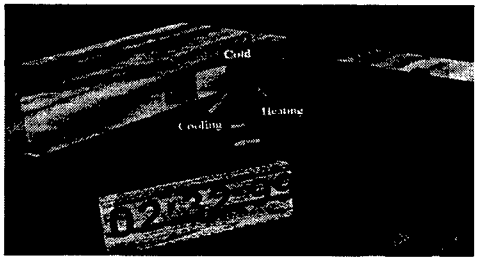


Figure 3: Pipeline closeup.

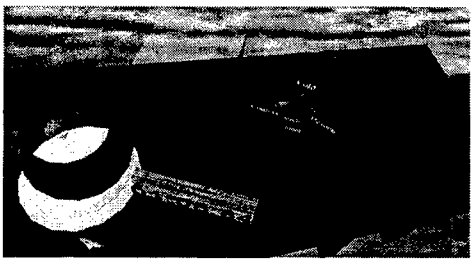


Figure 4: Cold state.

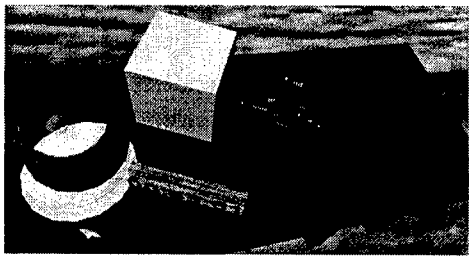


Figure 5: Heating state.

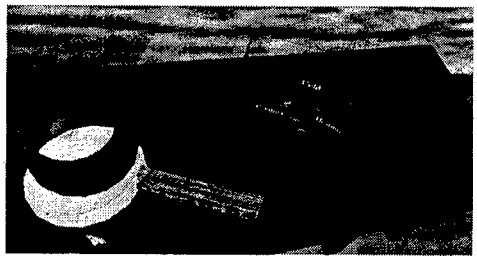


Figure 6: Cooling state.

in TeaWorld to capture the information flow. Inside the second machine, we find a plant, not unlike a petroleum refinery with tanks and pipes.

The *Pipeline* and its components represent physical objects that can be acquired from the web. For our example, we show simple objects but they have been given meaningful real-world application-oriented names to enforce the view that one object models another and that we can use the web for searching and using objects for radically different purposes than their proposed original function. The overriding concern with this exercise is to permit the modeler the freedom to choose *any* object to model *any* behavior. The challenge is to choose a set of objects that provide metaphors that are meaningful to the modeler. In many cases, it is essential that more than one individual understand the metaphorical mappings and so consensus must be reached during the process. Such consensus occurs routinely in science and in modeling when new modeling paradigms evolve. The purpose of *rube* is not to dictate one model type over another, but to allow the modelers freedom in creating their own model types. In this sense, *rube* can be considered a meta-level modeling methodology.

The simulation of the VRML scene shown in Fig. 2 proceeds using the dashed line thread that begins with the *Clock*. The clock has an internal time sensor that controls the VRML time. The thread corresponds closely with the routing structure built for this model. It starts at *Clock* and proceeds downward through all behavioral models. Within each behavioral model, routes exist to match the topology of the model. Therefore, *Machine1* sends information to *Machine2*, which accesses a lower level of abstraction and sends its output to *Machine3*, completing the semantics for the FBM. The FSM level contains routes from each state to its outgoing transitions.

Fig. 3 shows a closeup view of the pipeline, that represents the dynamics of the water, beginning with the effect of turning of the knob and ending with the thermometer that reads the water temperature.

Figs. 4 through 6 show the pipeline during simulation when the knob is turned ON and OFF at random times by the user. The default state is the cold state. When the knob is turned to the OFF position, the system moves into the heating state. When the knob is turned again back to the OFF position, the system moves into the cooling state and will stay there until the water reaches the ambient temperature at which time the system (through an internal state transition) returns to the cold state. Temperature change is indicated by the color of *Water* and *Machine3*, in addition to the reading on the *Thermometer* inside of *Machine3*. The material properties of *Machine1* change depending on the state of the knob. When turned to the OFF position, *Machine1* is semi-transparent. When turned on, it turns opaque. Inside *Machine2*, the current state of the water is reflected by the level of intensity of each *Plant*. The current state has an increased intensity, resulting in a bright red sphere.

The dynamics of temperature is indicated at two

Figure 7: Outside the Heating phase.

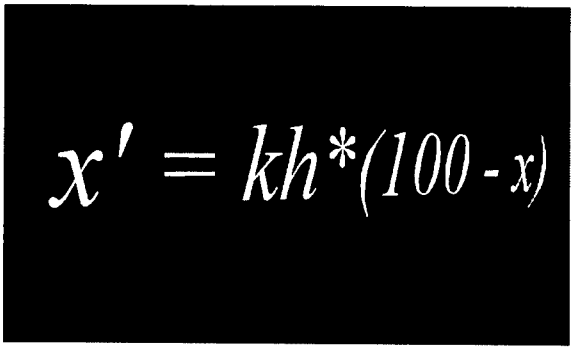


Figure 8: Inside the Heating phase.

levels. At the highest level of the plant, we have a three state FSM. Within each state, we have a differential equation. The equation is based on Newton's Law of Cooling and results in a first order exponential decay and rise that responds to the control input from the knob. The visual display of temperature change confirms this underlying dynamics since the user finds the temperature changing ever more slowly when heating to 100°C or cooling back to the ambient temperature. Fig. 7 displays a closeup of the heating phase from the outside, and Fig. 8 is a view from inside the red sphere modeling the phase.

3 *rube* Methodology

The procedure for creating models is defined as follows:

1. The user begins with an object that is to be modeled. This object can be primitive or complex—such as a scene–with many sub-objects. In the case of the teapot, we identify the teapot, water, knob, heating element as well as other aspects of the environment: room, walls, desk and floor. Not all of these objects require embedded behavioral models: some objects are props or exist for contextual reasons.

2. The *scene* and object interactions are sketched in a story board fashion, as if creating a movie or animation. A scene is where all objects, including those modeling others, are defined within a VRML file. The *rube* model browser is made available so that users can "fly though" an object to view its models without necessarily cluttering the scene with all objects. However, having some subset of the total set of models surfaced within a scene is also convenient for aesthetic reasons. The modeler may choose to build several scenes with models surfaced, or choose to view objects only through the model browser that hides all models as fields of VRML object nodes. In Fig. 1, the object *Pipeline* models the heating and cooling of the water, which is inside the teapot to the left of *Pipeline*. We could also have chosen to place a GUI behavioral model handle for the teapot *inside* the teapot itself or within close proximity of the teapot.

3. The shape and structure of all objects are modeled in any modeling package that has an export facility to VRML. Most packages, such as Kinetix 3DStudioMax and Autodesk AutoCAD have this capability. Moreover, packages such as CosmoWorlds and VRCreator can be used to directly create and debug VRML content. We used CosmoWorlds for the walls, floor and *Pipeline*. Other objects, such as the teapot, desk and chair were imported from the web.

4. VRML PROTO (i.e., prototype) nodes are created for each object, model and components thereof. This step allows one to create *semantic attachments* so that we can define one object to be a behavioral model of another (using a *behavior* field) or to say that the water is contained within the teapot. Without prototypes, the VRML file structure lacks semantic relations and one relies on simple grouping nodes, which are not sufficient for clearly defining how objects relate to one another. PROTOs are created for all physical objects, whether or not the objects are role-playing as a behavior model or as a behavior model component. This is discussed in more depth in Sec. 4.

5. Models are created. While multiple types of models exist, we have focused on dynamic models of components, and the expression of these components in 3D. Even textually-based models that must be visualized as mathematical expressions can be expressed using the VRML text

node. Models are objects in the scene that are no different structurally from pieces of visible objects being modeled—they have shape and structure. The only difference is that when an object is "modeling" another, one interprets the object's structure in a particular way, using a dynamic model template for guidance.

6. Several dynamic model templates exist. For Newell's Teapot (in Sec. 2), we used three: FBM, FSM, and EQN. These acronyms are defined as follows: FSM = Finite State Machine; FBM = Functional Block Model; EQN = Equation Set. Equations can be algebraic, ordinary differential, or partial differential. The FBM serves to capture the control flow from the activity of the knob to the temperature change of the water, and on to the thermometer. The FSM inside *Machine2* of *Pipeline* models the water temperature changes.

7. The creative modeling act is to choose a dynamic model template for object behavior, and then to pick objects that will convey the meaning of the template within the scenario. This part is a highly artistic enterprise since literally any object can be used. It is not the policy of *rube* to recommend or insist upon one metaphor. In practice, different groups will evolve and certain metaphors may compete in a process akin to natural selection. Our *Pipeline* could easily have been more artistically modeled so that it appeared more as a pipeline, and so the *Plant* looked more like an industrial plant. We were caught between trying to employ metaphor to its fullest extent and wanting those familiar with traditional 2D behavior models to follow the *rube* methodology.

8. There are three distinct types of roles played by modelers in *rube*. At the lowest level, there is the person creating the *model templates* (FSM,FBM,EQN,PETRI-NET). Each dynamic model template reflects an underlying system-theoretic model [5]. At the mid-level, the person uses an existing model template to create a *metaphor*. An industrial plant is an example of a manufacturing metaphor. At the highest level, a person is given a set of metaphors and can choose objects from the web to create a model. These levels allow modelers to work at the levels where they are comfortable. Reusability is created since one focuses on the level of interest.

9. The simulation proceeds by the modeler creating threads of control that pass events from one VRML node to another. This can be done in one of two ways: 1) using VRML Routes, or 2) using exposed fields that are accessed from other nodes. Method 1 is familiar to VRML authors and also has the advantage that routes that extend from one model component to an adjacent component (i.e., from one state to another or from one function to another) have a topological counterpart to the way we visualize information and control flow. The route defines the topology and data flow semantics for the simulation. Method 2 is similar to what we find in traditional object-oriented programming languages where information from one object is made available to another through an assignment statement that references outside objects and classes. Such an assignment is termed "message passing." In method 1, a thread that begins at the root node proceeds downward through each object that is role-playing the behavior of another. The routing thread activates Script nodes that are embedded in the structures that act as models or model components for the behaviors. All objects acting as behavioral model components are connected to a VRML clock (i.e., TimeSensor) so that multimodeling is made possible by allowing model components to gain control of the simulation and proceed in executing lower level model semantics.

10. Pre- and Post-processing is performed on the VRML file to check it for proper syntax and to aid the modeler. Pre-processing tools include wrappers (that create a single VRML file from several), decimators (that reduce the polygon count in a VRML file), and VRML parsers. The model browser mentioned earlier is a post-production tool, allowing the user to browse all physical objects to locate objects that model them. In the near future, we will extend the parser used by the browser to help semi-automate the building of script nodes. The browser and underlying VRML parser is based in Java (using JavaCUP) and therefore can be activated through the web browser.

rube treats all models in the same way. For a clarification of this remark, consider the traditional use of the word "Modeling" as used in everyday terms. A model is something that contains attributes of a target object, which it is modeling. Whereas, equation and 2D graph-based models could be viewed as being fundamentally different from a commonsense model, *rube* views them in exactly the same context: everything is an object with physical extent and modeling is a relation among objects. This unification is theoretically pleasing since it unifies what it means

to "model" regardless of model type. We are able to unify the commonsense view of modeling (i.e., scale or clay models) with more abstract modeling techniques used on the computer.

4 VRML IMPLEMENTATION OF THE TEAPOT

Newell's Teapot has the VRML scene graph structure as shown in Fig. 2, but there are also key prototype definitions used for objects and the models. The structure of the PROTOs are as follows. First, we have the PROTO for the dynamic model type FSM (Finite State Machine):

```
EXTERNPROTO FSM [
  eventIn      SFFloat   set_clock
  field        SFVec3f   position
  exposedField SFBool    input
  eventIn      SFString  set_state
  field        SFNode    start_state
  field        MFNode    sounds
  field        MFNode    states
  field        MFNode    transitions
  field        SFBool    passive
  field        MFNode    active
] "fsm.wrl#FSM"
```

The FSM is composed of `states` and `transitions`, with each state having a sound. There is a `start_state` and a `position` for placing the FSM in the scene. `set_clock` is the clock input and allows the FSM to take its `input` to drive the state transition changes.

Each FSM may be modeled at one of two levels: `active` and `passive`. The use of the 3 tanks and 4 pipes is passive since there is no motion–only a change in intensity of each tank when a state is enabled. An active mode implies the existence of two extra nodes, a `mover` and a `path`, both of which define the geometry associated with an object moving along a path. For example, if `active` has a field value of `[avatar24 spline3]` then node `avatar24` would make a motion along a physical path defined by node `spline3` to denote a change in state.

```
EXTERNPROTO FSM_STATE [
  eventIn      SFFloat   set_clock
  exposedField SFBool    enabled
  field        MFNode    audio
  exposedField MFNode    geometry
  field        MFNode    behavior
] "fsm.wrl#FSM_STATE"
```

Each state and transition has a geometry model and a behavior mode. The geometry model is that which defines how the object is to be rendered. The behavior model defines how the state is to be executed. `behavior` may terminate in a VRML Script node, but may also be further defined by another 3D structure to any abstraction level. Using `geometry`, we may allow any 3D scene or object to reflect the notion of state.

```
EXTERNPROTO FSM_TRANSITION [
  eventIn      SFFloat  set_clock
  exposedField SFBool   enabled
  eventOut     SFString state_changed
  field        SFNode   from
  field        SFNode   to
  field        SFNode   fsm
  field        SFNode   object
  exposedField MFNode   geometry
  field        MFString behavior
] "fsm.wrl#FSM_TRANSITION"
```

The transition nodes are similar in that one may assign both `geometry` and `behavior` to them. Each transition has `from` and `to` states. The transition also carries the behavior "logic" that determines whether a `state_change` occurs.

The metaphor elements are mapped directly to model templates, each of which is defined by a PROTO node:

- Industrial Plant metaphor → Finite state machine → FSM PROTO
- Tank (in Plant) metaphor → State → FSM-STATE PROTO
- Pipe (in Plant) metaphor → Transition → FSM-TRANSITION PROTO

5 PROGRAMMING USING VRML

Object-Oriented Software engineering has long advocated the use of modeling in defining the creation of software. A recent example is the significant interest in the Unified Modeling Language (UML) [12, 9]. The embedded nature of software encourages the modeling approach since the design reflects the distributed nature of the hardware components. Software engineers evolve into system modelers. Using VRML, we created a small operating system kernel that involves metaphors for tasks (using avatars), routes through the system (using colored paths) and resources (using office desks with attendants). The overall operating system is shown as an office building in Fig. 9

and inside the building there is a floor designated as the kernel (Fig. 10). Tasks begin in a waiting

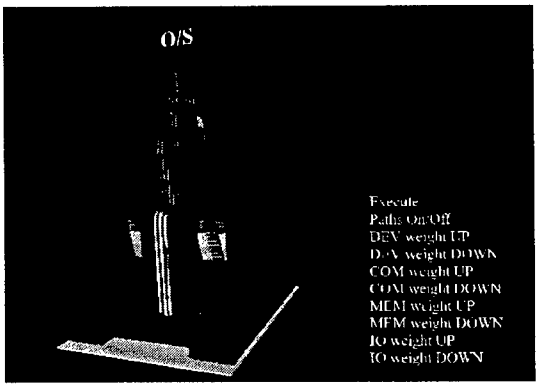


Figure 9: The operating system structure.

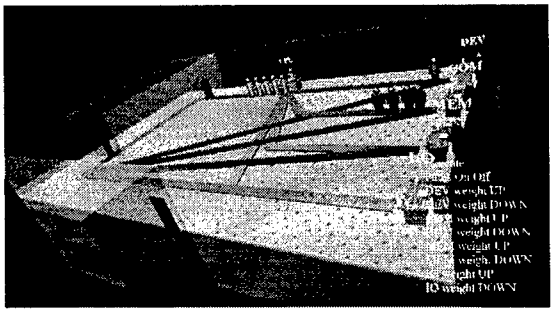


Figure 10: Inside the O/S kernel.

area on the left side of Fig. 10 and proceed to the CPU desk. Tasks continue to move toward the key resources (DEVice, COMmunications,MEMory,I/O). The paths to the resources are in orange and the return paths back to the holding area are blue. When a resource is requested, this begins an audio track specifying which resource is being used.

6 VRML ISSUES

We found VRML to be fairly robust for the task set side within *rube* but there are a number of issues that need to be resolved in the future. The most serious issues, having an effect on *rube*, are delineated:

1. VRML needs a strong object-oriented (OO) structure with classes, inheritance, object instances and a capability for referring to a current object with a *this* field. We found it difficult to create a regional scoping of public variables, for example, in allowing a component to gain access to its parents fields. Instead, one has to expose a field to every other node. One side-effect of a solid OO architecture would be that there would be a distinct difference between defining a Node and creating one. The existing `DEF` both defines and creates.

2. Exposed fields should operate exactly as if one were to specify an `eventIn`, `field` and `eventOut`. Several VRML texts suggest an equivalence, but in practice they are quite different. Currently, if one creates an `exposedField`, then one cannot define an `eventIn` for setting values in a script node using a function of the `eventIn` name. There are many instances where it would be useful to use both methods of access to an exposedField: 1) directly with `node.set_field`, or 2) indirectly with a route using `set_field` as an `eventIn` within a script node. Routes are useful in surfacing connections between one node and another, but prudent use of `exposedFields` (if they were more completely implemented) simplifies a spaghetti-like network of routes.

3. Both forward and backward references to nodes should be possible. Currently, one cannot specify `USE somenode` unless it has already been defined with `DEF`. This may require a multi-pass option on parsing the VRML scene graph, which would slow the parsing but give the VRML author the freedom to choose forward referencing where they may wish to implement it.

4. Scripting capabilities need to be expanded to support native code, and need to be consistent among browsers in supporting Java in the Script node and full implementations of Javascript. Instead of being implementor options, they should be required. Native code is essential if VRML is to both *compete with*, *expand upon* and *reuse* other 3D software.

5. We found PROTO and EXTERNPROTO support to be variable among VRML software developers. Since these are among the most important node types in VRML, their implementations should be ubiquitous in all VRML modelers and software support.

7 ART OF MODELING

Given the Newell Teapot scene, there are some key issues which we should ask ourselves:

- *Is it "just" a visualization?* The work in *rube* provides visualization, but models such as

Newell's Teapot demonstrate active modeling environments whose existence serves an engineering purpose and not only a post-project visualization purpose for outside visitors. This sort of modeling environment is needed from the very start of a mission—as an integral piece of the puzzle known as model design. There is little question that this sort of production is useful for teaching purposes, but we also view this as a precursor to next generation software engineering. The power of VRML in this regard is that it can be used to reinvent software engineering through the surfacing of 3D models. It is one thing to think of this as a *visualization* of an Operating System kernel, but it is quite another to call it the Operating System *itself*. We need to bridge this gap if we are to progress beyond textual, linear programming styles.

- *Is it economical?* Is this a lot of work just to create an FSM? All 3D objects are reused and so can be easily grabbed from the web. The concept of reuse is paramount to the *rube* approach where the metaphor can be freely chosen and implemented. Without the web, *rube* would not be possible. 3D object placement can be just as economical as 2D object placement, but object repositories are required.

- *What is the advantage?* If we consider psychological factors, the 3D metaphor has significant advantages. First, 3D spatially-specific areas serve to improve our memory of the models (i.e., mnemonics). Second, graphical user interfaces (GUIs) have shown that a human's interaction with the computer is dramatically improved when the right metaphors are made available. *rube* provides the environment for building metaphors. One should always be wary of mixed metaphors. We leave the ultimate decision to the user group as to which metaphors are effective. A Darwinian-style of evolution will likely determine which metaphors are useful and which are not. Aesthetics plays an important role here as well. If a modeler uses aesthetically appealing models and metaphors, the modeler will enjoy the work. It is a misconception to imagine that only the general populous will benefit from fully interactive 3D models. The engineers and scientist need this sort of immersion as well so that they can understand better what they are doing, and so that collaboration is made possible.

- *Is this art or science?* The role of the Fine Arts in science needs strengthening. With fully immersive models, we find that we are in need of workers with hybrid engineering/art backgrounds. It is no longer sufficient to always think "in the abstract" about modeling. Effective modeling requires meaningful human interaction with 3D objects. So far, the thin veneer of a scale model has made its way into our engineering practices, but when the skin is peeled back, we find highly abstract code and text. If the internals are to be made comprehensible (by anyone, most importantly the engineer), they must be surfaced into 3D using the powerful capabilities of metaphors [7, 6]. This doesn't mean that we will not have a low level code-base. Two-dimensional metaphors and code constructs can be mixed within the 3D worlds, just as we find them in our everyday environments with the embedding of signs. At the University of Florida, we have started a *Digital Arts and Sciences* Program with the aim to produce engineers with a more integrated background. This background will help to produce new workers with creative modeling backgrounds.

- *What role does aesthetics play in modeling?* It is sometimes difficult to differentiate models used for the creation of pieces of art from those used with scientific purposes in mind. Models used for science are predicated on the notion that the modeling relation is unambiguously specified and made openly available to other scientists. Modeling communities generally form and evolve while stressing their metaphors. In a very general sense, natural languages have a similar evolution. The purpose of art, on the other hand, is to permit some ambiguity with the hopes of causing the viewer or listener to reflect upon the modeled world. Some of the components in worlds such as Fig. 1 could be considered non-essential modeling elements that serve to confuse the scientist. However, these elements may contribute to a more pleasing immersive environment. Should they be removed or should we add additional elements to please the eye of the beholder? In *rube*, we have the freedom to go in both directions, and it isn't clear which inclusions or eliminations are appropriate since it is entirely up to the modeler or a larger modeling community. One can build an entirely two dimensional world on a blackboard using box and text objects, although this would not be in the spirit of creating immersive worlds that allow perusal of objects and their models.

  It may be that a select number of modelers may find the TeaWorld room exciting and pleasing, and so is this pleasure counterproductive to the scientist or should the scientist be concerned only

with the bare essentials necessary for unambiguous representation and communication? Visual models do not represent *syntactic sugar* (a term common in the Computer Science community). Instead, these models and their metaphors are essential for human understanding and comprehension. If this comprehension is complemented with a feeling of excitement about modeling, this can only be for the better. Taken to the extreme, a purely artistic piece may be one that is so couched in metaphor that the roles played by objects isn't clear. We can, therefore, imagine a kind of continuum from a completely unambiguous representation and one where the roles are not published. Between these two extremes, there is a lot of breathing space. Science can be seen as a form of consensual art where everyone tells each other what one object *means*. Agreement ensues within a community and then there is a mass convergence towards one metaphor in favor of another.

8 SUMMARY

Effort to unify behavioral and software engineering modeling methodologies are useful, but we should also have a way to express models more creatively and completely. Model communities will naturally evolve around 2D and 3D metaphors yet to be determined. *rube* has a strong tie to the World Wide Web (WWW). The web has introduced a remarkable transformation in every area of business, industry, science and engineering. It offers a way of sharing and presenting multimedia information to a worldwide set of interactive participants. Therefore any technology tied to the web's development is likely to change modeling and simulation. The tremendous interest in Java for doing simulation has taken a firm hold within the simulation field. Apart from being a good programming language, its future is intrinsically bound to the coding and interaction within a browser. VRML, and its X3D successor, represent the future of 3D immersive environments on the web. We feel that by building a modeling environment in VRML and by couching this environment within standard VRML content, that we will create a *Trojan Horse* for simulation modeling that allows modelers to create, share and reuse VRML files.

Our modeling approach takes a substantial departure from existing approaches in that the modeling environment and the material object environment are merged seamlessly into a single environment. There isn't a difference between a circle and a house, or a sphere and a teapot. Furthermore, objects can take on any role, liberating the modeler to choose whatever metaphor that can be agreed upon by a certain community. There is no single syntax or structure for modeling. Modeling is both an art and a science; the realization that all objects can play roles takes us back to childhood. We are building *rube* in the hope that by making all objects virtual that we can return to free-form modeling of every kind. Modeling in 3D can be cumbersome and can take considerable patience due to the inherent user-interface problems when working in 3D using a 2D screen interface. A short term solution to this problem is to develop a model package that is geared specifically to using one or more metaphors, making the insertion of, say, the petroleum refinery a drag and drop operation. Currently, a general purpose modeling package must be used to carefully position all objects in their respective locations. A longer term solution can be found in the community of virtual interfaces. A good immersive interface will make 3D object positioning and connections a much easier task than it is today.

ACKNOWLEDGMENTS

We would like to thank the students on the *rube* Project: Robert Cubert, Andrew Reddish, John Hopkins and Linda Dance. Also, we thank the following agencies that have contributed towards our study of modeling and simulation: (1) Jet Propulsion Laboratory under contract 961427 *An Assessment and Design Recommendation for Object-Oriented Physical System Modeling at JPL* (John Peterson, Stephen Wall and Bill McLaughlin); (2) Rome Laboratory, Griffiss Air Force Base under contract F30602-98-C-0269 *A Web-Based Model Repository for Reusing and Sharing Physical Object Components* (Al Sisti and Steve Farr); and (3) Department of the Interior under grant 14-45-0009-1544-154 *Modeling Approaches & Empirical Studies Supporting ATLSS for the Everglades* (Don DeAngelis and Ronnie Best). We are grateful for their continued financial support.

References

[1] Grady Booch. *Object Oriented Design.* Benjamin Cummings, 1991.

[2] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual.* Addison-Wesley, 1997.

[3] Robert M. Cubert and Paul A. Fishwick. MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework. *Simulation*, 70(6):379–395, 1998.

[4] Paul A. Fishwick. Simpack: Getting Started with Simulation Programming in C and C++. In *1992 Winter Simulation Conference*, pages 154–162, Arlington, VA, 1992.

[5] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall, 1995.

[6] George Lakoff. *Women, Fire and Dangerous Things: what categories reveal about the mind.* University of Chicago Press, 1987.

[7] George Lakoff and Mark Johnson. *Metaphors We Live By.* University of Chicago Press, 1980.

[8] Peter C. Marzio. *Rube Goldberg, His Life and Work.* Harper and Row, New York, 1973.

[9] Pierre-Alain Muller. *Instant UML.* Wrox Press, Ltd., Olton, Birmingham, England, 1997.

[10] Winfried Noth. *Handbook of Semiotics.* Indiana University Press, 1990.

[11] James Rumbaugh, Michael Blaha, William Premerlani, Eddy Frederick, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[12] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, MA, 1999.

AUTHOR BIOGRAPHY

PAUL FISHWICK is Professor of Computer and Information Science and Engineering at the University of Florida. He received the PhD in Computer and Information Science from the University of Pennsylvania in 1986. His research interests are in computer simulation, modeling, and animation, and he is a Fellow of the Society for Computer Simulation (SCS). Dr. Fishwick will serve as General Chair for WSC00 in Orlando, Florida. He has authored one textbook, co-edited three books and published over 100 technical papers.

A Hybrid Visual Environment for Models and Objects

Paul A. Fishwick

Department of Computer & Information Science & Engineering
University of Florida
Gainesville, Florida 32611, U.S.A.

July 13, 1999

ABSTRACT

Models and objects that are modeled are usually kept in different places when we consider most modern simulation software packages. Software that permits the user to view 3D objects may also permit a viewing of the dynamic models for the objects, but these views are usually separate. The object can be rotated, translated and navigated while the model is represented in a 2D fashion using text or 2D iconic graphics. We present an approached based on the Virtual Reality Modeling Language (VRML), where the object and model reside in the same space. A browsing capability is built to allow the user to search for models "within" objects. Aside from the visual benefits derived from this integrated approach, this methodology also suggests that models are really not very different from objects. Any object can serve to model another object and when these objects are made "web friendly," it becomes feasible to use VRML to create distributed models whose components can reside anywhere over the web.

1 THE NATURE OF MODELING

One physical object captures some information about another object. If we think about our plastic toys, metal trains and even our sophisticated scale-based engineering models, we see a common thread: to build one object that says something about another—usually larger and more expensive—object. Let's call these objects the *source object* and the *target object.* Similar object definitions can be found in the literature of metaphors (Lakoff 1987) and semiotics (Noth 1990). The source object *models* the target, and so, modeling represents a relation between objects. Often, the source object is termed *the model* of the target. We have been discussing scale models identified by the source and target having roughly proportional geometries. Scale-based models often suffer from the problem where changing the scale of a thing affects more than just the geometry. It also affects the fundamental laws applied at each scale. For example, the hydrodynamics of the scaled ocean model may be different than for the real ocean. Nevertheless, we can attempt to adjust for the scaling problems and proceed to understand the larger universe through a smaller, more manipulable, version.

Later on in our education, we learned that modeling has other many other forms. The mathematical model represents variables and symbols that describe or model an object. Learning may begin with algebraic equations such as $d = \frac{1}{2}at^2 + v_o t + d_0$ where d, v and a represent distance, velocity and acceleration, and where d_0 and v_0 represent initial conditions (i.e., at time zero) for starting distance and initial velocity. These models are shown to be more elegantly derived from Newton's laws, yielding ordinary differential equations of the form $f = ma$. How do these mathematical, equational models relate to the ones we first learned as children?

To answer this question, let's first consider what is being modeled. The equations capture attributes of an object that is undergoing change in space (i.e., distance), velocity and acceleration. However, none of the geometrical proportions of the target are captured in the source since the structure of the equations is invariant to the physical changes in the target. A ball can change shape during impact with the ground, but the equations do not change their *shape.* If a ball represents the target, where is the source? The source is the medium in which the equations are presented. This may, at first, seem odd but it really is no different than the toy train model versus the actual train. The paper, phosphor or blackboard—along with the medium for the drawing, excitation or marking—has to exist if the equations are to exist. In a Platonic sense, we might like to think of the equations as existing in a separate, virtual, non-physical space. While one can argue their virtual existence, this representation-less and non-physical form is impractical. Without a physical represen-

tation, the equation cannot be communicated from one human to another. The fundamental purpose of representation and modeling is communication. Verbal representations (differential air pressure) are as physical as those involving printing or the exciting of a phosphor via an electron beam.

2 RUBE

Since 1989 at the University of Florida, we have constructed a number of modeling and simulation packages documented in (Fishwick 1992; Cubert and Fishwick 1998) In late 1998, we started designing Rube, named in dedication to Rube Goldberg (Marzio 1973), who produced many fanciful cartoon machines, all of which can be considered *models* of behavior. The procedure for creating models is defined as follows. *Step 1*: The user begins with an object that is to be modeled. For JPL, this can be the Cassini spacecraft with all of its main systems: propulsion, guidance, science instrumentation, power, and telecommunication. If the object is part of a larger scenario, this scenario can be defined as the top-most root object; *Step 2*: *scene* and interactions are sketched in a story board fashion, as if creating a movie or animation. A scene is where all objects, including those modeling others, are defined within the VRML file. VRML stands for Virtual Reality Modeling Language (Carey and Bell 1997), which represents the standard 3D language for the web. The Rube model browser is made available so that users can "fly though" an object to view its models without necessarily cluttering the scene with all objects. However, having some subset of the total set of models surfaced within a scene is also convenient for aesthetic reasons. The modeler may choose to build several scenes with models surfaced, or choose to view objects only through the model browser that hides all models as fields of VRML object nodes; *Step 3*: The shape and structure of all Cassini components are modeled in any modeling package that has an export facility to VRML. Most packages, such as Kinetix 3DStudioMax and Autodesk AutoCAD have this capability. Moreover, packages such as CosmoWorlds and VRCreator can be used to directly create and debug VRML content; *Step 4*: VRML PROTO (i.e., prototype) nodes are created for each object and component. This step allows one to create *semantic attachments* so that we can define one object to be a behavioral model of another (using a *behavior* field) or to say that the Titan probe is part of the spacecraft (using a *contains* field), but a sibling of the orbiter. Without prototypes, the VRML file structure lacks semantic relations and one relies on simple grouping nodes, which are not sufficient for clearly defining how objects relate to one another; *Step 5*: Models are created for Cassini. While multiple types of models exist, we have focused on dynamic models of components, and the expression of these components in 3D. Even textually-based models that must be visualized as mathematical expressions can be expressed using the VRML text node. Models are objects in the scene that are no different structurally from pieces of Cassini—they have shape and structure. The only difference is that when an object is "modeling" another, one interprets the object's structure in a particular way, using a dynamic model template for guidance; *Step 6*: Several dynamic model templates exist. For Newell's Teapot (in Sec. 3), we used three: FBM, FSM, EQN. These acronyms are defined as follows: FSM = Finite State Machine; FBM = Functional Block Model; EQN = Equation Set. Equations can be algebraic, ordinary differential, or partial differential; *Step 7*: The creative modeling act is to choose a dynamic model template for some behavior for Cassini and then to pick objects that will convey the meaning of the template within the scenario. This part is a highly artistic enterprise since literally any object can be used. In VRML, one instantiates an object as a *model* by defining it: `DEF Parthenon-Complex FSM {...}`. In other words, a collection of Parthenon-type rooms are interconnected in such a way that each Parthenon-Room maps to a state of the FSM. Portals from one room to another become transitions, and state-to-state transitions become avatar movements navigating the complex; *Step 8*: There are three distinct types of roles played modelers in Rube. At the lowest level, there is the person creating the *model templates* (FSM,FBM,EQN,PETRI-NET). Each dynamic model template reflects an underlying system-theoretic model (Fishwick 1995). At the mid-level, the person uses an existing model template to create a *metaphor*. A Parthenon-Complex as described before is an example of an architectural metaphor. At the highest level, a person is given a set of metaphors and can choose objects from the web to create a model. These levels allow modelers to work at the levels where they are comfortable. Reusability is created since one focuses on the level of interest; *Step 9*: The simulation proceeds by the modeler creating threads of control that pass events from one VRML node to another. This can be done in one of two ways: 1) using VRML Routes, or 2) using exposed fields that are accessed from other nodes. Method 1 is familiar to VRML authors and also has the advantage that routes that extend from one model component to an adjacent component (i.e., from one state to another or from one function to another) have a topolog-

ical counterpart to the way we visualize information and control flow. The route defines the topology and data flow semantics for the simulation. Method 2 is similar to what we find in traditional object-oriented programming languages where information from one object is made available to another through an assignment statement that references outside objects and classes. In method 1, a thread that begins at the root node proceeds downward through each object that is role-playing the behavior of another. The routing thread activates Java or Javascript Script nodes that are embedded in the structures that act as models or model components for the behaviors; *Step 10*: Pre- and Post-processing is performed on the VRML file to check it for proper syntax and to aid the modeler. Pre-processing tools include wrappers (that create a single VRML file from several), decimators (that reduce the polygon count in a VRML file), and VRML parsers. The model browser mentioned earlier is a post-production tool, allowing the user to browse all physical objects to locate objects that model them. In the near future, we will extend the parser used by the browser to help semi-automate the building of script nodes.

Rube treats all models in the same way. For a clarification of this remark, consider the traditional use of the word "Modeling" as used in everyday terms. A model is something that contains attributes of a target object, which it is modeling. Whereas, equation and 2D graph-based models could be viewed as being fundamentally different from a commonsense model, Rube views them in exactly the same context: everything is an object with physical extent and modeling is a relation among objects. This unification is theoretically pleasing since it unifies what it means to "model" regardless of model type.

3 NEWELL'S TEAPOT

In the early days of computer graphics (c. 1974-75), Martin Newell rendered a unique set of Bézier surface spline patches for an ordinary teapot, which currently resides in the Computer Museum in Boston. The teapot was modeled by Jim Blinn and then rendered by Martin Newell and Ed Catmull at the University of Utah in 1974. While at this late date, the teapot may seem quaint, it has been used over the years as an icon of sorts, and more importantly as a benchmark for all variety of new techniques in rendering and modeling in computer graphics. The Teapot was recently an official emblem of the 25th anniversary of the ACM Special Interest Interest Group on Computer Graphics (SIGGRAPH).

One of our goals for Rube was to recognize that the Teapot could be used to generate another potential benchmark—one that captured the entire teapot, its contents and its models. The default teapot has no behavior and has no contents; it is an elegant piece of geometry but it requires more if we are to construct a fully *digital teapot* that captures a more complete set of knowledge. In its current state, the teapot is analogous to a building façade on a Hollywood film studio backlot; it has the shape but the whole entity is missing. In VRML, using the methodology previously defined, we built TeaWorld in Fig. 1. We have added extra props so that the teapot can be visualized, along with its behavioral model, in a reasonable contextual setting. The world is rendered in Fig. 1 using a web browser. *World* is the top-most root of the scene graph. It contains a *Clock*, *Boiling_System*, and other objects such as the desk, chairs, floor and walls. The key fields in Fig. 2 are VRML nodes of the relevant field so that the *contains* field is refers to multiple nodes for its value. This is accomplished using the VRML *MFNode* type. The hierarchical VRML scene graph for Fig. 1 is illustrated in Fig. 2. The

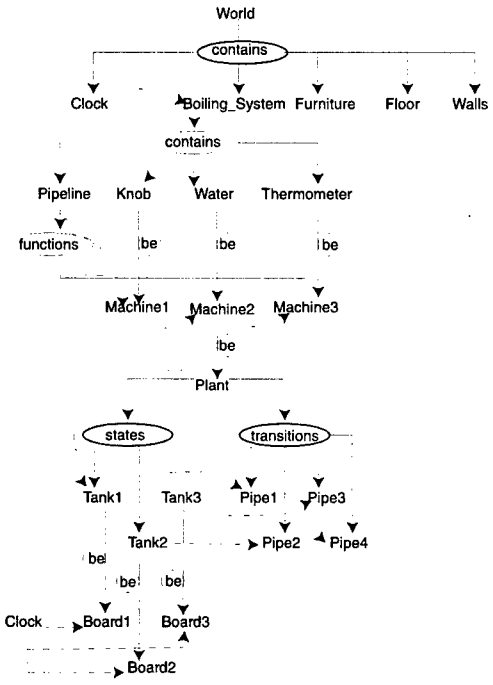


Figure 2: VRML Scene Graph for the Teapot and its models.

scene contains walls, a desk, chair and a floor for context. On the desk to the left is the teapot which is filled with water. The knob controlling whether the teapot heating element (not modeled) is on or

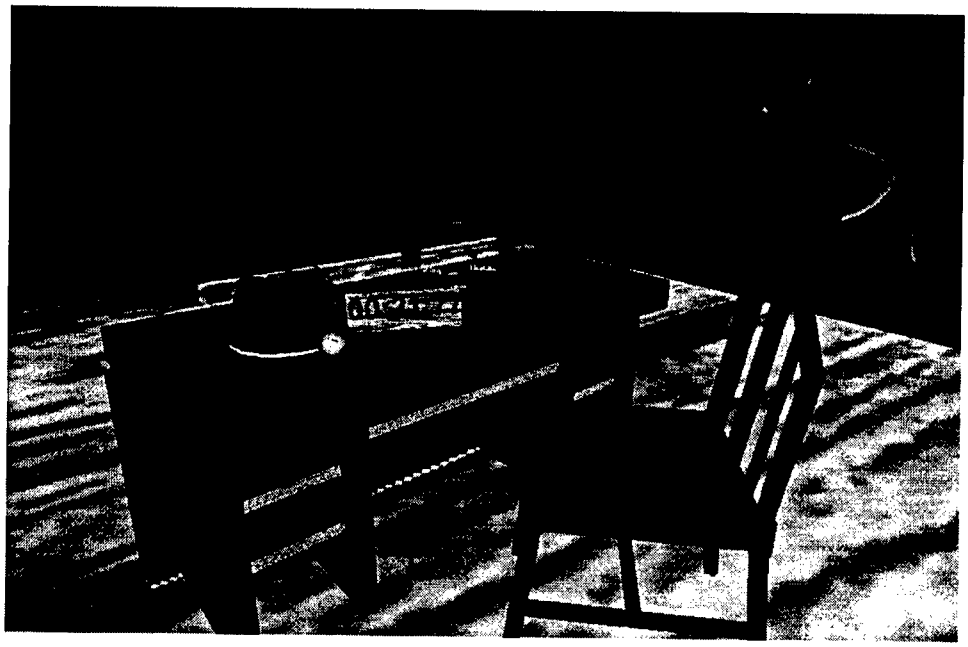


Figure 1: Office scene with Newell Teapot, dynamic model and props.

off is located in front of the teapot. To the right of the teapot, there is a pipeline with three machines, each of which appears in Fig. 1 as a semi-transparent cube. Each of these machines reflects the functional behavior of its encapsulating object: *Machine1* for *Knob*, *Machine2* for *Water* and *Machine3* for *Thermometer*. The *Thermometer* is a digital one that is positioned in *Machine3*, and is initialized to an arbitrary ambient temperature of 0° C. Inside *Machine2*, we find a more detailed description of the behavior of the water as it changes its temperature as a result of the knob turning. The plant inside *Machine2* consists of *Tank1*, *Tank2*, *Tank3*, and four pipes that move information from one tank to the next. Inside of each tank, we find a blackboard on which is drawn a differential equation that defines the change in water temperature for that particular state. The following modeling relationships are used: *Pipeline* is a Functional Block Model (FBM), with three functions (i.e., machines); *Machine* is a function (i.e., semi-transparent cube) within an FBM; *Plant* is a Finite State Machine (FSM) inside of Machine 2; *Tank* is a state within a FSM, and represented by a red sphere; *Pipe* is a transition within a FSM, and represented by a green pipe with a conical point denoting direction of control flow; and *Board* is a differential equation, represented as white text. The following metaphors are defined in this example. The three cubes represent a sequence of machines that create a pipeline. One could have easily chosen a factory floor sequence of numerically controlled machines from the web and then used this in TeaWorld to capture the information flow. Inside the second machine, we find a plant, not unlike a petroleum plant with tanks and pipes.

The *Pipeline* and its components represent physical objects that can be acquired from the web. For our example, we show simple objects but they have been given meaningful real-world application-oriented names to enforce the view that one object models another and that we can use the web for searching and using objects for radically different purposes than their proposed original function. The overriding concern with this exercise is to permit the modeler the freedom to choose *any* object to model *any* behavior. The challenge is to choose a set of objects that provide metaphors that are meaningful to the modeler. In many cases, it is essential that more than one individual understand the metaphorical mappings and so consensus must be reached during the process. Such consensus occurs routinely in science and in modeling when new modeling paradigms evolve. The purpose of Rube is not to dictate one model type over another, but to allow the modelers freedom in creating their own model types. In this sense, Rube can be considered a meta-level modeling methodology.

The simulation of the VRML scene shown in Fig. 2 proceeds using the dashed line thread that begins with the *Clock*. The clock has an internal time sensor that controls the VRML time. The thread corresponds closely with the routing structure built for this model. It starts at *Clock* and proceeds downward through all behavioral models. Within each behav-

ioral model, routes exist to match the topology of the model. Therefore, *Machine1* sends information to *Machine2*, which accesses a lower level of abstraction and sends its output to *Machine3*, completing the semantics for the FBM. The FSM level contains routes from each state to its outgoing transitions.

Fig. 3 shows a closeup view of the pipeline, that represents the dynamics of the water, beginning with the effect of the turning of the knob and ending with the thermometer that reads the water temperature.

Figs. 4,5 and 6 show the pipeline during simulation when the knob is turned on and off at random times by the user. The default state is the cold state. When the knob is turned to the on position, the system moves into the heating state. When the knob is turned again back to an off position, the system moves into the cooling state and will stay there until the water reaches ambient room temperature at which time the system (through an internal state transition) returns to the cold state. Temperature change is indicated by the color of *Water* and *Machine3*, in addition to the reading on the *Thermometer* inside of *Machine3*. The material properties of *Machine1* change depending on the state of the knob. When turned off, *Machine1* is semi-transparent. When turned on, it turns opaque. Inside *Machine2*, the current state of the water is reflected by the level of intensity of each *Plant*. The current state has an increased intensity, resulting in a bright red sphere.

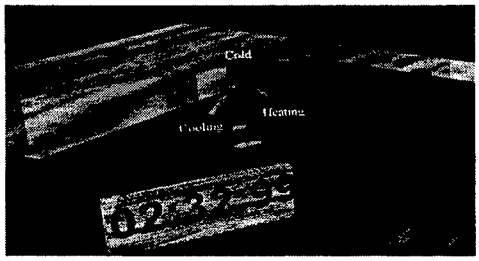


Figure 3: Pipeline closeup.

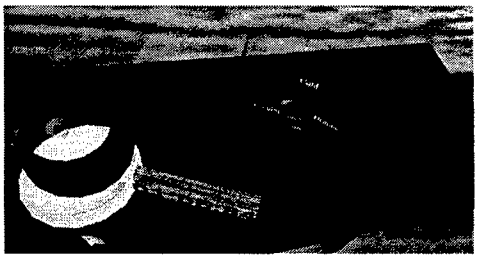


Figure 4: Cold state.

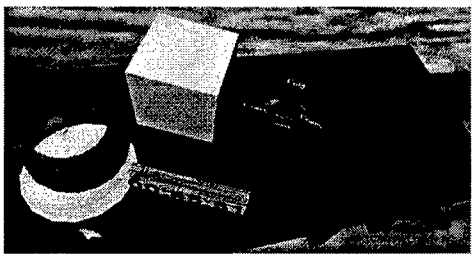


Figure 5: Heating state.

Figure 6: Cooling state.

Figure 7: Outside the Heating phase.

The dynamics of temperature is indicated at two levels. At the highest level of the plant, we have a three state FSM. Within each state, we have a differential equation. The equation is based on Newton's Law of Cooling and results in a first order exponential decay and rise that responds to the control input from the knob. The visual display of temperature change confirms this underlying dynamics since the user finds the temperature changing ever more slowly when heating to 100°C or cooling back to the ambient temperature. Fig. 7 displays a closeup of the heating phase from the outside, and Fig. 8 is a view from inside the red sphere modeling the phase.

Given the Newell Teapot scene, there are some key issues which we should ask ourselves: *Is it a visualization*? The work in Rube provides visualiza-

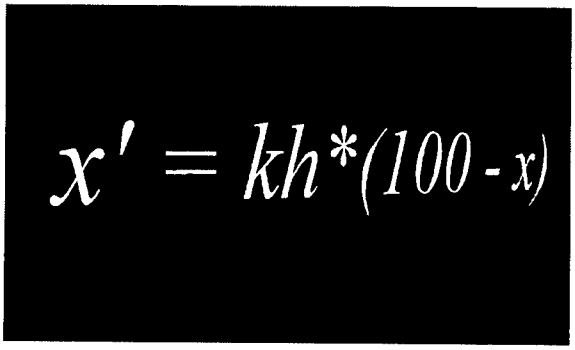


Figure 8: Inside the Heating phase.

tion, but models such as Cassini and Newell's Teapot demonstrate active modeling environments whose existence serves an engineering purpose and not only a post-project visualization purpose for outside visitors. This sort of modeling environment is needed from the very start of a mission—as an integral piece of the puzzle known as model design; *Is it economical?* Is this a lot of work just to create an FSM? Why go through the bother of creating the Parthenon, the complex and the avatar? All of these items are reused and so can be easily grabbed from the web. The concept of reuse is paramount to the Rube approach where the metaphor can be freely chosen and implemented. Without the web, Rube would not be possible. 3D object placement can be just as economical as 2D object placement, but object repositories are required not only for Cassini and Titan, but also for objects that serve to model the dynamic attributes of other objects (i.e., the Parthenon). Another economical aspect centers on the issue of computational speed for these models. Would creating a simulation in a more typical computer language would be more efficient? The structure of objects and their models within a VRML scene can be translated or compiled into native machine code as easily as source code; the 3D model structure becomes the "source code;" *What is the advantage?* If we consider psychological factors, the 3D metaphor has significant advantages. First, 3D spatially-specific areas serve to improve our memory of the models (i.e., mnemonics). Second, graphical user interfaces (GUIs) have shown that a human's interaction with the computer is dramatically improved when the right metaphors are made available. Rube provides the environment for building metaphors. One should always be wary of mixed metaphors. We leave the ultimate decision to the user group as to which metaphors are effective. A Darwinian-style of evolution will likely determine which metaphors are useful and which are not. Aesthetics plays an important role here as well. If a modeler uses aesthetically appealing models and metaphors, the modeler will enjoy the work. It is a misconception to imagine that only the general populous will benefit from fully interactive 3D models. The engineers and scientist need this sort of immersion as well so that they can understand better what they are doing, and so that collaboration is made possible: *Is this art or science?* The role of the Fine Arts in science needs strengthening. With fully immersive models, we find that we are in need of workers with hybrid engineering/art backgrounds. It is no longer sufficient to always think "in the abstract" about modeling. Effective modeling requires meaningful human interaction with 3D objects. So far, the thin veneer of a scale model has made its way into our engineering practices, but when the skin is peeled back, we find highly abstract codes and text. If the internals are to be made comprehensible (by anyone, most importantly the engineer), they must be surfaced into 3D using the powerful capabilities of metaphors (Lakoff and Johnson 1980; Lakoff 1987). This doesn't mean that we will not have a low level code-base. Two-dimensional metaphors and code constructs can be mixed within the 3D worlds, just as we find them in our everyday environments with the embedding of signs. At the University of Florida, we have started a *Digital Arts and Sciences* Program with the aim to produce engineers with a more integrated background. This background will help in the production of new workers with creative modeling backgrounds.

4 ART OF MODELING

It is sometimes difficult to differentiate models used for the creation of pieces of art from those used with scientific purposes in mind. Models used for science are predicated on the notion that the modeling relation is unambiguously specified and made openly available to other scientists. Modeling communities generally form and evolve while stressing their metaphors. In a very general sense, natural languages have a similar evolution. The purpose of art, on the other hand, is to permit some ambiguity with the hopes of causing the viewer or listener to reflect upon the modeled world. Some of the components in worlds such as Fig. 1 could be considered non-essential modeling elements that serve to confuse the scientist. However, these elements may contribute to a more pleasing immersive environment. Should they be removed or should we add additional elements to please the eye of the beholder? In Rube, we have the freedom to go in both directions, and it isn't clear which inclusions or eliminations are appropriate since it is entirely up to the modeler or a larger modeling

community. One can build an entirely two dimensional world on a blackboard using box and text objects, although this would not be in the spirit of creating immersive worlds that allow perusal of objects and their models.

It may be that a select number of modelers may find the TeaWorld room exciting and pleasing, and so is this pleasure counterproductive to the scientist or should the scientist be concerned only with the bare essentials necessary for unambiguous representation and communication? Visual models do not represent *syntactic sugar* (a term common in the Computer Science community). Instead, these models and their metaphors are essential for human understanding and comprehension. If this comprehension is complemented with a feeling of excitement about modeling, this can only be for the better. Taken to the extreme, a purely artistic piece may be one that is so couched in metaphor that the roles played by objects isn't clear. We can, therefore, imagine a kind of continuum from a completely unambiguous representation and one where the roles are not published. Between these two extremes, there is a lot of breathing space. Science can be seen as a form of consensual art where everyone tells each other what one object *means*. Agreement ensues within a community and then there is a mass convergence towards one metaphor in favor of another.

We are not proposing a modification to the VRML standard although we have found that poor authoring support currently exists in VRML editors for PROTO node creation and editing. We are suggesting a different and more more general mindset for VMRL—that it be used not only for representing the shape of objects, but all modeling information about objects. VRML should be about the complete digital object representation and not only the representation of geometry with low-level script behaviors to support animation. Fortunately, VRML contains an adequate number of features that makes this new mindset possible even though it may not be practiced on a wide scale. While a VRML file serves as the digital object, a model compiler is also required for the proper interpretation of VRML objects as models.

5 SUMMARY

There is no unified modeling methodology, nor should there be one. Instead, modelers should be free to use and construct their own worlds that have special meaning to an individual or group. With Rube, we hope to foster that creativity without limiting a user to one or more specific metaphors. Rube has a strong tie to the World Wide Web (WWW). The web has introduced a remarkable transformation in every area of business, industry, science and engineering. It offers a way of sharing and presenting multimedia information to a world-wide set of interactive participants. Therefore any technology tied to the web's development is likely to change modeling and simulation. The tremendous interest in Java for doing simulation has taken a firm hold within the simulation field. Apart from being a good programming language, its future is intrinsically bound to the coding and interaction within a browser. VRML, and its X3D successor, represent the future of 3D immersive environments on the web. We feel that by building a modeling environment in VRML and by couching this environment within standard VRML content, that we will create a "trojan horse" for simulation modeling that allows modelers to create, share and reuse VRML files.

Our modeling approach takes a substantial departure from existing approaches in that the modeling environment and the object environment are merged seamlessly into a single environment. There isn't a difference between a circle and a house, or a sphere and a teapot. Furthermore, objects can take on any role, liberating the modeler to choose whatever metaphor that can be agreed upon by a certain community. There is no single syntax or structure for modeling. Modeling is both an art and a science; the realization that all objects can play roles takes us back to childhood. We are building Rube in the hope that by making all objects virtual that we can return to free-form modeling of every kind. Modeling in 3D can be cumbersome and can take considerable patience due to the inherent user-interface problems when working in 3D using a 2D screen interface. A short term solution to this problem is to develop a model package that is geared specifically to using one or more metaphors, making the insertion of, say, the Parthenon complex rooms a drag and drop operation. Currently, a general purpose modeling package must be used carefully position all objects in their respective locations. A longer term solution can be found in the community of virtual interfaces. A good immersive interface will make 3D object positioning and connections a much easier task than it is today.

We will continue our research by adding to Rube and extending it to be robust. In particular, we plan on looking more closely into the problem of taking legacy code and making it available within the VRML model. This is probably best accomplished through TCP/IP and a network approach where the Java/Javascript communicates to the legacy code as a separate entity. We plan on extending the VRML

parser, currently used to create the model browser, so that it can parse a 3D scene and generate the Java required for the VRML file to execute its simulation. Presently, the user must create all Script nodes. The model browser will be extended to permit various modes of locating models within objects. A "fly through" mode will take a VRML file, with all object and model prototypes, and place the models physically inside each object that it references. This new generated VRML file is then browsed in the usual fashion. Multiple scenes can be automatically generated.

ACKNOWLEDGMENTS

I would like to thank the students on the Rube Project: Robert Cubert, Andrew Reddish, and John Hopkins. I would like to thank the following agencies that have contributed towards our study of modeling and simulation: (1) Jet Propulsion Laboratory under contract 961427 *An Assessment and Design Recommendation for Object-Oriented Physical System Modeling at JPL* (John Peterson, Stephen Wall and Bill McLaughlin); (2) Rome Laboratory, Griffiss Air Force Base under contract F30602-98-C-0269 *A Web-Based Model Repository for Reusing and Sharing Physical Object Components* (Al Sisti and Steve Farr); and (3) Department of the Interior under grant 14-45-0009-1544-154 *Modeling Approaches & Empirical Studies Supporting ATLSS for the Everglades* (Don DeAngelis and Ronnie Best). We are grateful for their continued financial support.

REFERENCES

Carey, R., and G. Bell. 1997. *The Annotated VRML 2.0 Reference Manual.* Addison-Wesley.

Cubert, R. M., and P. A. Fishwick. 1998. MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework*Simulation* 70(6), 379–395.

Fishwick, P. A. 1992. Simpack: Getting Started with Simulation Programming in C and C++. In *1992 Winter Simulation Conference*, Arlington, VA, 154–162.

Fishwick, P. A. 1995. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall.

Lakoff, G. 1987. *Women, Fire and Dangerous Things: what categories reveal about the mind.* University of Chicago Press.

Lakoff, G., and M. Johnson. 1980. *Metaphors We Live By.* University of Chicago Press.

Marzio, P. C. 1973. *Rube Goldberg, His Life and Work.* New York: Harper and Row.

Noth, W. 1990. *Handbook of Semiotics.* Indiana University Press.

AUTHOR BIOGRAPHY

PAUL FISHWICK is Professor of Computer and Information Science and Engineering at the University of Florida. He received the PhD in Computer and Information Science from the University of Pennsylvania in 1986. His research interests are in computer simulation, modeling, and animation, and he is a Fellow of the Society for Computer Simulation (SCS). Dr. Fishwick will serve as General Chair for WSC00 in Orlando, Florida. He has authored one textbook, co-edited three books and published over 100 technical papers.

On the Use of 3D Metaphor in Programming

John F. Hopkins[a], Paul A. Fishwick[b]

Dept. of Computer Information Science and Engineering, Univ. of Florida, Gainesville, FL 32611

ABSTRACT

The use of metaphor in programming can be a powerful aid to the programmer, inasmuch as it provides concrete properties to abstract ideas. In turn, these concrete properties can aid recognition of, and reasoning about, programming problems. Another potential benefit of the use of metaphor in programming is the improvement of mental retention of facts and solutions to programming problems. Traditionally, programs have been produced in a textual medium. However, a textual medium may be inferior to a three-dimensional medium in the development and use of metaphor, as the concrete properties that metaphors provide are real-world phenomena, which are naturally three-dimensional. An example of the use of three-dimensional metaphors in programming was created. This consisted of a mock operating system task scheduler, along with some associated hardware devices, developed in a VRML environment using VRML PROTO nodes. These nodes were designed as objects based on real-world metaphors. The issues, problems, and novelties involved in programming in this manner were explored.

Keywords: Metaphor, 3D, Three-Dimensional, Programming Method, VRML, Rube

1. INTRODUCTION

The act of programming began as a mathematical activity, extending from machine code to assembly language, and then onto languages with humanly readable text with natural language variable and function names. In 40 years, programming has evolved from its abstract roots and yet text is still the primary medium in which programs are created, often with shorthand notations for program components. For economic reasons, there has always been a strong tendency toward abstraction in Computer Science and this is evident in the nature of modern programming languages. For example, the process of iteration is accomplished in a way that has not changed much in all these years. The same can be said for most other programming constructs, even though we have come far with new design paradigms such as component, agent, and object-based software architectures.

We present a method of programming that is significantly different than many others, although some of our techniques are shared by many modeling paradigms (e.g., agent-based, swarm-based, biologically-based, and object-oriented) and several burgeoning areas such as Software and Information Visualization, Programming by Example, and Visual Programming. These efforts, while diverse, all have one thing in common: they suggest new programming techniques that are based on the real world. We have built upon this general idea to create a paradigm called "Rube" (for Rube Goldberg), in which real-world objects are an integral part of the language.

Our programs are multimodels, where a "multimodel" is defined as a hierarchically connected set of behavioral models, each of a specific type. The basic types are numerous and include finite state machine (FSM), functional block model (FBM), Petri net (PNET), and queuing net (QNET). These base behavioral model types and their components are mapped to an arbitrary domain using metaphor. The choices of domain and metaphor are artistic choices that are left to the programmer. We use the Virtual Reality Modeling Language (VRML) as our language for encoding these models and bringing them to life.

In the current research, we demonstrate and examine the interplay between the multimodeling and metaphor aspects of the Rube paradigm through the synthesis of an example program. First, we conduct a short survey of related research and compare it to the current research. Next, we create a hypothetical operating system task scheduler using the Rube multimodeling and metaphor methodology. Lastly, we discuss the issues, advantages, and disadvantages of the approach.

[a] e-mail: jhopkins@cise.ufl.edu
[b] e-mail: fishwick@cise.ufl.edu

2. BACKGROUND AND RELATED RESEARCH

In order to provide a context for the current research, we provide some background. This background includes the consideration of object-oriented design philosophy, Software and Information Visualization, and work done with three-dimensional authoring tools. We conduct a brief survey of other three-dimensional programming research, including CUBE and Programming by Example/Demonstration. Then, we compare and contrast the aforementioned research with the current research. We find that the current research shares many similarities with related research, yet is significantly different in several ways.

2.1. Object Oriented Thinking and Design

In part, the object-oriented approach to software design and construction is concerned with modeling. Specifically, it suggests that before any textual code is written, a model of the software's functionality should be created which will guide and inform the rest of the process in a "top-down" fashion – from abstract/general to concrete/specific. In this effort, modularization is a primary design criterion. In a broad sense, and as suggested by the term "object-oriented," these modules may be viewed as individual objects, each with their own functions, interfaces, attributes, and allowable operations. These objects confer many benefits on the programmer, as they facilitate the easy management of what otherwise might be significant complexity, simplify the debugging process, and create the potential for code re-use, to name a few.[5]

It is not difficult to see the parallels between object-oriented design and the world in which we live every day, both in an abstract and a physical sense. In fact, a common way used by computer science instructors to teach novices to understand the object-oriented approach is to draw parallels between it and the real world. For example, when we are patrons of restaurants, we deal with the waitstaff, and not directly with the kitchen. In an object-oriented world, the waiter or waitress is an object, we are objects, and the kitchen is an object. The waitstaff serve as our interfaces to the kitchen, since they possess that privilege and we do not. We would violate the modularity enforced by the commonly termed "client-server" relationship if we attempted to deal directly with the kitchen as patrons.

This suggests a form of design and programming in three dimensions, where these parallels can be surfaced visually during the design and programming process. At the moment, programmers typically may use pencil and paper to make sketches of program modules and the relationships between them, use visual programming tools, visualize these modules mentally, or do none of the above. Some may proceed directly to text coding. However, a characteristic of all the methods mentioned above is that they are implemented in two dimensions (or less). If we suppose that we as humans are physical creatures that live in a three-dimensional world, it may follow that a better approach to design and programming lies in the utilization of a three-dimensional design and programming environment. Such an environment would allow for the incorporation of geometry and behavioral dynamics into object-oriented design.

Although object-oriented design has no specific relationship with the use of three-dimensional metaphors in programming, a frequent occurrence during the design process is that one or more abstract types or functions are created. If we must attempt to visualize an abstract function such as a sorter, we may arbitrarily decide to visualize it as a green pyramid. However, if it is possible to visualize the sorter as an animated person sorting boxes, what we have done is created a metaphor for the sorter. We have made an analogy between the abstract sorter, and a concrete, real-world object such as the person. If the sorter is visualized as a person in this fashion, there is no need to memorize the previous mapping of the green pyramid to the sorter. The visualization of the person and the metaphor introduced now provides this mapping implicitly. In effect, the visualization provides a semantic clue as to the object's function. This sort of use of metaphor, then, may find great utility during the design and debugging process.

2.2. Software and Information Visualization

Software Visualization is primarily concerned with using computer graphics and animation to illustrate programs, processes, and algorithms. Software Visualization systems are sometimes used as teaching aids and also during program development as a way to help programmers understand their code.[1] Information Visualization is primarily concerned with visualizing data as an aid in interpreting and understanding the information.[2] In general, the focus of Software and Information Visualization is on visualizing programs or data after they have been created using traditional means.

Several problems worthy of study have been encountered during visualization research. These include problems of abstraction of operations, data, and semantics, levels of detail, scaling, and user navigation.[1, 2] Additionally, visualization

may occur in two dimensions, three dimensions, or a combination of these. Obviously, there are myriad ways to visualize programs and data, and perhaps the challenge of visualization research is to find the most usable and efficient ways of accomplishing this task.

The difficulties encountered in visualization research are also frequently encountered in other fields and disciplines. Any two or three-dimensional programming method or environment must tackle many of the same issues encountered in visualization research. This suggests that visualization is a sub-problem of any such development system, either implicitly or explicitly.

2.3. Work in VRML and Other Three-Dimensional Authoring Tools

Many three-dimensional geometry-authoring tools have been developed, such as CosmoWorlds and 3DSMax. These tools are appropriate for creating three-dimensional environments and objects, and also for modeling relatively simple behaviors of these environments and objects. Occasionally, environments and objects created in this fashion are endowed with the ability to interact with the user. Artists are one community that make frequent use of these tools in their work, as it provides them a relatively cheap, reusable, flexible, and unique medium. Geometry, sound, light, color, movement, and interaction can be combined into a whole using these tools.

VRML, or Virtual Reality Modeling Language, is a popular three-dimensional language that is based on the concept of a tree hierarchy of nodes. Some reasons for its popularity include the fact that it is non-proprietary, portable, and many browsers for it are freely available. Another reason is the proliferation and accessibility of the Internet. An approximation of "generic" and "object-oriented" programming can be achieved through the use of a type of VRML node called PROTO (short for "prototype") and the use of a type of structure called ROUTE that can be conceptualized as a "pipe" that connects structures to events. Code for behaviors of objects can be written in JavaScript or some likeness of it. These attributes, among others, make it attractive for an early attempt at development of a three-dimensional programming environment. However, the use of VRML is not without problems and difficulties, especially concerning the implementation of data structures and complex behaviors that are typical of modern programs.

2.4. Three-Dimensional Programming

A primary goal of three-dimensional programming is to enable the user to program using executable graphics. A brief review of two approaches to this problem is given below.

2.4.1. CUBE

Marc Najork's CUBE is an example of a three-dimensional visual programming language that has a three-dimensional syntax. In this language, programs consist of an arrangement of three-dimensional shapes instead of a linear stream of text. CUBE supports recursion, function mapping, type checking, and user-defined types. One aim of the language is that of supporting virtual-reality based editing of programs for the user.[7]

The CUBE language is divided into two fragments. The first fragment consists of predicate definitions and logic formulas, and the second fragment consists of type definitions and type expressions. Three-dimensional cubes represent terms and atomic formulas in the logic fragment. Pipes connect the arguments of predicate applications (i.e., atomic formulas), forming a data flow diagram. The third dimension is used to group various two-dimensional diagrams together. Sets of base types and primitive predicate definitions are provided initially. The prototype implementation of CUBE functions by translating visual programs into a text format, and then operates on the text format.[7]

CUBE does not make extensive use of visual metaphor. However, it does make use of the data flow metaphor on a high level and as a guiding paradigm for the representation of the language.[7] As described above, this metaphor also happens to extend into the three-dimensional realm. From this, it can be seen that the use of metaphor need not be limited to its typical, everyday senses, but also can be used as a guiding framework for a form of representation.

2.4.2. Programming by Example/Demonstration

Programming by Example and Programming by Demonstration (PBE/PBD) are synonymous terms for a type of programming characterized by the following phases: 1) the user performs a series of steps on some example of a problem,

2) the computer records the steps, and 3) the computer applies these steps on some new, similar example. The programs generated in the recording step may not be applied exactly as they were recorded, but may adapt according to the differences in future examples.[4] This sometimes requires such programs to be able to generalize the steps to new instances, and to perform inference in varying degrees, from no inference to a great deal of inference.[6]

One focus of PBE is that of catering to the novice or non-programmer. For a novice programmer, there is usually a significant difference between the computer's representation of a problem and his or her own. As programmers become more experienced, they become more conditioned to think of a given problem in the context of the computer's representation, since the problem must ultimately be expressed in this fashion and there has been no efficient way to communicate the programmer's representation to the computer. PBE would bridge this gap by "bringing the computer closer to the programmer" rather than vice-versa.[9]

PBE attempts to lower the barriers between novice users and computer programming by eliminating, as much as possible, the idiosynchrocies of traditional programming language syntax. It attempts to provide visual, interactive, intuitive, and functional interfaces for program construction, and performs inference (to varying degrees) to reduce the amount of inference that must be performed by the programmer.[6] A recent trend in the PBE community has been to extend into the three-dimensional programming language arena.[7, 10] This extension is natural since we are conditioned to think of the world around us in three dimensions, and so this is one way to bring the computer closer to the programmer. Attempts may be made, so far as possible, to keep programs and constructs in concrete domain terms rather than abstract computer terms.[8] In short, a goal is to let an end-user program without realizing that he or she is programming.

Ken Kahn's *Toon Talk* is a PBD system designed for children in which "the programmer is a character in an animated virtual world in which programming abstractions are replaced by their tangible analogs." Data structures are given three-dimensional geometry, such as a box with holes, into which different objects may be placed. Birds and nests perform "send" and "receive" operations. Robots are guarded clauses that are trained to perform actions when they receive boxes. The programmer may generalize robots by using a vacuum cleaner to remove details from the robots' guard clauses.[3] It can be seen here that *Toon Talk* makes use of visual metaphor.

2.5. Comparison of the Current Research with Related Research

The current research, with its focus on extending programs and their components to resemble real-world objects (in an abstract, non-visual sense), has many parallels with object-oriented design approaches to programming. Since these parallels are somewhat obvious, they will not be elucidated here. However, the current research also seeks to give these objects three-dimensional representations that convey meaning through visual metaphor. In this way it is unlike, and could be considered an extension of object-oriented design.

Software and Information Visualization research and the current research share several problems in common. These include abstraction vs. concretion, levels of detail, navigation, and many others. Without doubt, solutions to these problems that have arisen and will arise during the course of either line of research should easily translate into the other line. However, in the previous description of Software and Information Visualization research it was stated that a general focus of that research is on visualizing programs and etc. after they have been created using traditional means. Therefore, this research tends not to focus on the immediate aspects of program development, such as the potential for creating a three-dimensional programming interface for users. Furthermore, metaphor is not commonly considered during such research. In these ways, the current research is unlike, and could again be considered an extension of Software and Information Visualization.

Work in three-dimensional authoring tools and the current research are alike inasmuch as objects are identified by their geometry and have specific physical and behavioral attributes. The current research was carried out in part within a three-dimensional authoring tool (CosmoWorlds) and in large part within a three-dimensional language (VRML). However, the current research is distinguished from typical three-dimensional authoring since most of the work was done in a text editor (for VRML coding), and a great deal of scripting was performed to make the behaviors of the objects in the world complex, dynamic, and interactive. The heavy use of PROTO nodes and ROUTEs in the current research also distinguish it from straightforward VRML environments.

The current research is most like CUBE and PBE, but is significantly unlike these in several respects. We did not use a three-dimensional editor significantly during the programming effort. The "dragging and dropping" performed was on three-dimensional objects that had only geometric representation. Unlike CUBE, there was no formal metaphor that informed a

three-dimensional syntax, nor was a set of three-dimensional representations of primitive data types and operations available for use. Also unlike CUBE, type checking was performed by exclusively by the programmer, and the text program was translated to the visual program. The use of metaphor and geometric shape in the current research was more free form than that of CUBE. The current research is unlike PBE inasmuch as the computer did not record the programmer's steps or perform any generalization or inference. Also unlike PBE, the current research is not immediately concerned with novice programmers, and there was no attempt to shield the programmer from the complexities of the implementation, although an ultimate goal of continued research would be to simplify the programming task for the programmer. And again, the use of metaphor in the current research was unrestricted and user-defined, unlike that of PBE.

3. EXAMPLE

In this section, we demonstrate the practical application of the current research through the presentation of an example program. The program represents a hypothetical operating system task scheduler that is synthesized utilizing metaphors and three-dimensional visualization. Several issues that arise during the programming effort are discussed. Based on this example it is found that, though not without difficulty, this approach to programming is viable and worthy of further research.

3.1. A First Attempt: A Hypothetical Operating System Task Scheduler/Queuing System

Since we are making the claim that three-dimensional metaphor can be used as a high-level design aid, we set out to choose for our example a complex program that should be familiar to our audience. Programs meeting these criteria should best demonstrate the utility of three-dimensional metaphor as a design aid. Though there are many choices that meet the familiar and complex criteria, we focus here on the operating system (OS) as a generic type.

In our example, we envision a MINIX-like OS, which can be characterized by hierarchical, vertical layers (top to bottom): 1) user processes (e.g., word processor), 2) server processes (e.g., file system), 3) I/O tasks (e.g., disk), and 4) process management (e.g., scheduling and communication). In the example, we focus on the process management portion of the OS, and specifically upon a hypothetical task scheduler in that layer. In an abstract sense, the task scheduler may be viewed as a complex queuing system, or QNET.

3.2. Metaphor Approach and Mappings

The sort of vertically layered structure of the OS described above lends itself naturally to an architectural metaphor, where the OS itself may be visualized as a building, as pictured in Fig. 1. The layers of the OS then correspond to the different floors of the building. This is the metaphor we choose. Besides floors, a couple of things that can typically be found inside buildings are people and furniture.

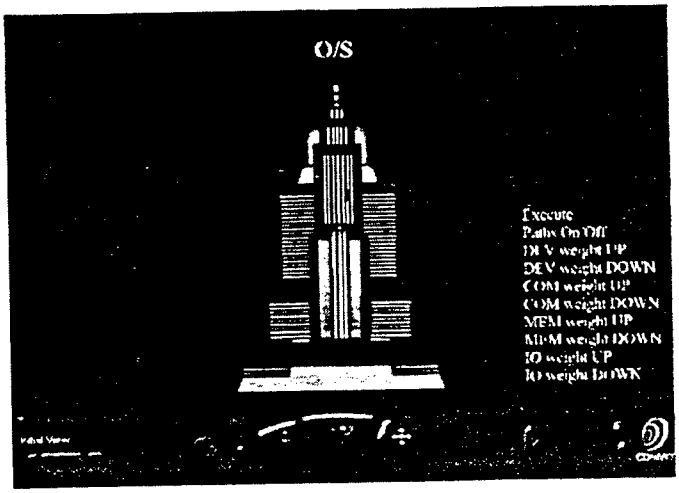


Figure 1. The architectural metaphor applied to the operating system.

Keeping these things in mind, we now proceed to model the operations of the layers of the OS. The primary interactions between the layers of our hypothetical OS occur through message passing. Although we do not immediately focus on a choice of metaphor for these interactions since the actual implementation of message passing may be somewhat complex, we choose a prospective metaphor and set it aside temporarily in order to concentrate on the construction of a single layer. The metaphor we choose for message passing is that of a person transporting a briefcase that contains the message from one place to another.

First, we decide to model the task scheduler portion of the process management layer. Our example task scheduler is a non-preemptive, dynamic priority scheduling system that contains tasks, four priority queues, and the following five types of physical devices with their associated queues: 1) CPU, 2) DEV (i.e., device), 3) COM (i.e., communication), 4) MEM (i.e., memory load/store), and 5) I/O (e.g., disk read/write). Expanding our previous metaphors, we map a task to a person and a device to a "service facility" which is a person behind a desk. The priority and device queues map directly to physical space as waiting lines. Persons can travel over paths between the devices and queues. Taking stock of our progression so far, overall we have at least a minimal way to represent the significant portions of our hypothetical task scheduler.

3.3. Implementation

We are now concerned with the details of implementing the above task scheduler along with its metaphors in a three-dimensional environment. In the context of VRML, we create PROTOs for each of the items in our task scheduler that will be used again and again. A VRML convention for naming PROTOs is to fully capitalize the PROTO name, e.g. MYPROTO. We create the following PROTOs: 1) PERSON, 2) CPUFACILITY, 3) FACILITY, 4) QUEUEMANAGER, and 5) PATH. PERSON is described below. The CPU has special management functions, so it is given its own PROTO. All devices besides the CPU will be instances of the generic FACILITY. When a person reaches a FACILITY, they are held there for some random amount of time and then released. Instances of QUEUEMANAGER will manage each of the priority queues and the queues leading to each of the service facilities. PATHs connect the devices together physically and manage the movement of persons throughout the world.

Each of the PROTOs mentioned has a physical geometry node included inside that will give it an identifiable and appropriate physical presence in the three-dimensional environment. Some of these physical geometry nodes are modeled in a three-dimensional authoring tool prior to their inclusion in the PROTOs. For example, instances of PERSON look like a stylized version of an actual person, the geometry of which is modeled in CosmoWorlds. These appearances are according to the metaphors we are using.

Some items of note within the PROTOs are the attributes they include in their declarations. For example, PERSON has the fields: 1) task type (one of CPU, DEV, COM, MEM, or I/O), 2) cpu run time left (only relevant if this is a CPU task), and 3) task priority (from 1 through 4). The implication here is that each instance of PERSON in our task scheduler carries around its own state information, which can be accessed and modified by other components of the scheduler. Other instances of PROTOs have similar ways of utilizing attributes.

After the objects of the task scheduler have been physically arranged in the environment, we have a static representation of the example task scheduler. The next job is to make the task scheduler function dynamically and in an animated fashion. This is accomplished in VRML through the occurrence and utilization of events. Events are generated and sent through ROUTEs to their destinations. The actual implementation involves directing an eventOut field of a VRML object to the left half of a ROUTE, and then directing the right half of the ROUTE to an eventIn field of another object. For example:
ROUTE ObjectName1.eventOutFieldname1 TO ObjectName2.eventInFieldname1

Note that PATH is a metaphor in the following respect: it has a parallel function as that of a ROUTE. PATH physically transports "movers" through the world in a like manner as ROUTE transports events through the world. See Fig. 2. PATHs are given physical geometry that can be turned on or off by the user.

Two more PROTOs are created to aid the implementation: 1) MFACTORY and 2) RANDOMIZER. There are other PROTOs that aid the implementation that are not mentioned here, but of these, MFACTORY and RANDOMIZER are the most important. MFACTORY generates people (tasks) dynamically, and RANDOMIZER randomly sets the fields of the persons to specific task types and assigns them a cpu run time if necessary. These two PROTOs have little to do with our initial metaphors. As such, it is not necessary for them to have a physical presence in the world and therefore they have no geometry associated with them. If desired, geometry could be added to them later.

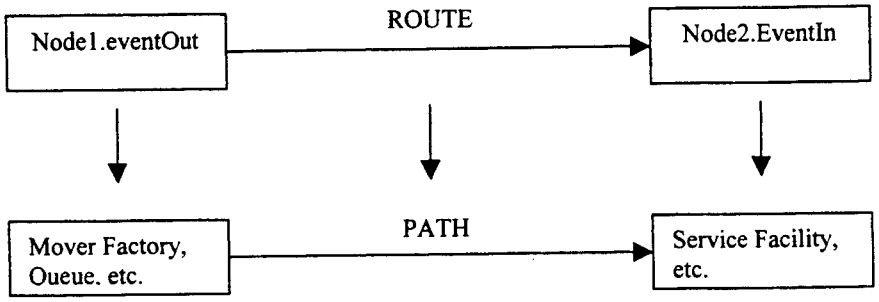


Figure 2. The path metaphor as applied to a generic ROUTE.

In the final product, proceeding in a top-down fashion, we first encounter the shell of the building of the OS as pictured in Fig. 1. Taking off the shell, we see a series of floors stacked one on the other as shown in Fig. 3. On one of these floors we see the task scheduler operating as shown in Fig. 3 and Fig. 4. Inside the task scheduler, we see persons moving through queues and along paths toward service facilities, where they spend some random amount of time receiving service as pictured in Fig. 5. Service itself has no animation (yet) but is associated with a unique type of sound that continues as long as that type of service is being provided. When their service at some facility is complete, persons leave the service facilities, enter a "void" area, and have their attributes reset to start the process again. This "recycling" of persons avoids destroying and regenerating them in order to continue the simulation. Lastly, a heads-up-display (HUD) is provided, which allows the user to turn the geometric representations of the paths on and off and alter the random probabilities of task type assignment, all while the simulation is operating.

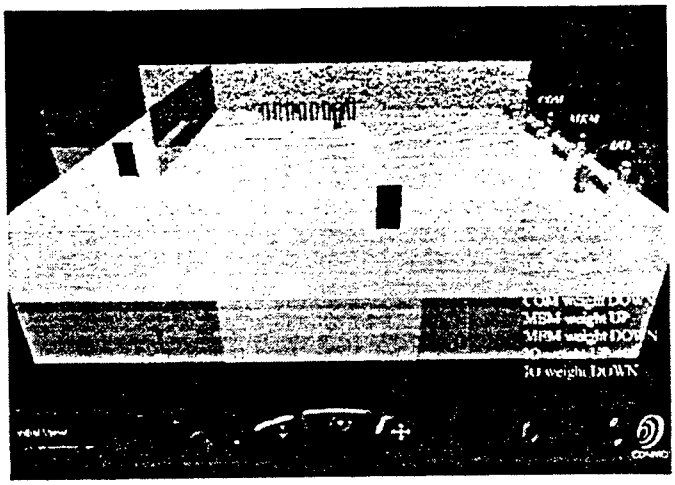


Figure 3. The floors of the OS building with the task scheduler shown on the top floor.

A prospective addition to the above-described world is being implemented at time of writing. This addition involves placing an FSM inside each person that reflects the associated task's current state: 1) In Transit, 2) Waiting, and 3) Running. These states roughly correspond to the familiar "waiting/running/ready" FSM for processes in an OS task scheduler. The user is able to zoom in on any particular person, and see through the person's transparent skin into the person's chest, where the FSM is shown progressing through its various states, as pictured in Fig. 6. This demonstrates the both the multimodeling and metaphor aspects of Rube.

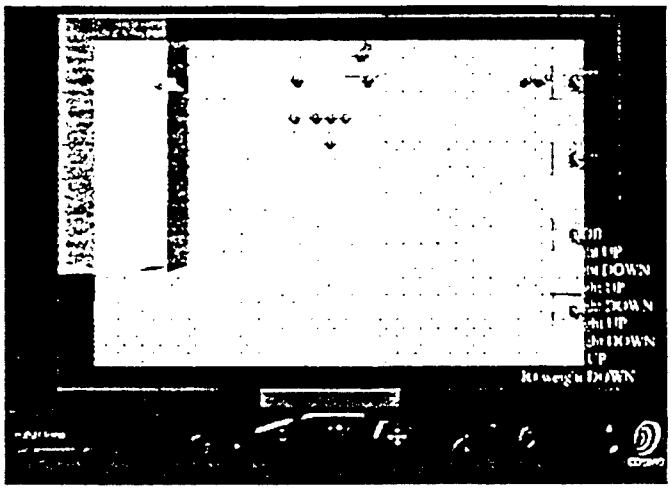


Figure 4. Top view of the task scheduler. The CPU facility is shown at top center, with its priority queues just below and to the left of it. DEV, COM, MEM, and I/O service facilities are on shown on the right, top to bottom respectively, with queues to the left of each.

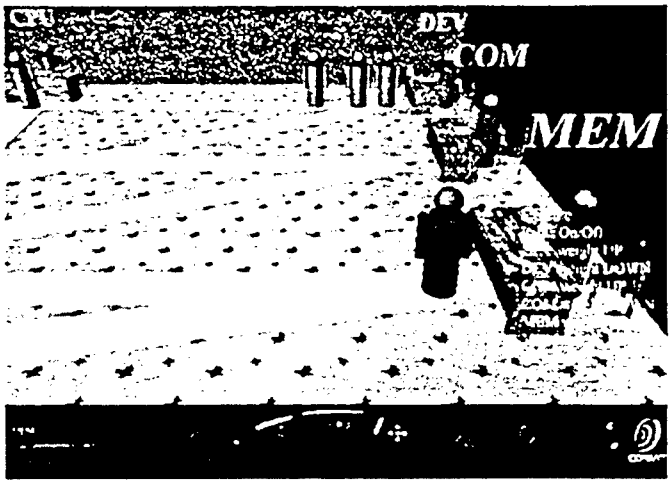


Figure 5. Close-up view of task scheduler operation.

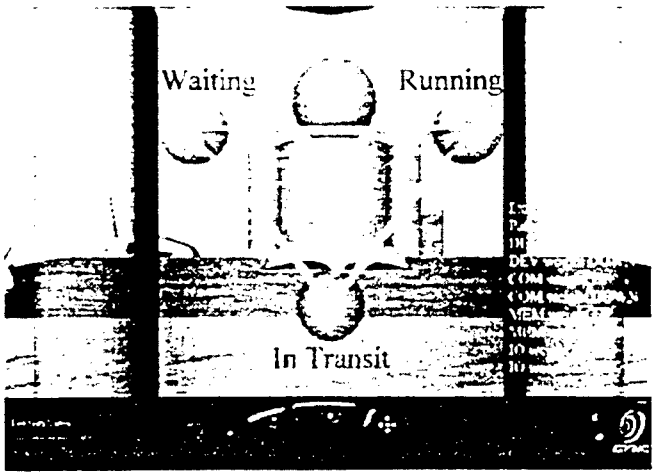


Figure 6. A task's internal FSM with the In Transit, Waiting, and Running states. The task shown is being serviced at the CPU.

The actual implementation of the above described three-dimensional world/program is located at the following URL at time of writing: http://www.cise.ufl.edu/~fishwick/rube/worlds/os.html

A more detailed account of the implementation and operation of the world can be found at the given URL, also. A VRML browser such as CosmoPlayer 2.1.1 or Blaxxun Contact 4.2 is required to view the world itself. The FSM addition described above is already implemented for Blaxxun Contact 4.2, but has minor bugs in CosmoPlayer 2.1.1, and so will not be released until such bugs are fixed.

3.4. Issues

3.4.1. Modeling, Metaphor, and the "Real World"

The choice of the architectural metaphor for the OS may guide the rest of the design process, but it is not the only possible metaphor that can be chosen. This choice is left to the programmer. Neither is it necessary that the architectural metaphor exclusively dominate the rest of the design process. There is great freedom in subsequently choosing peer or sub-metaphors that may or may not have some relation to the architectural metaphor. These choices may lead to a "multi-metaphor" construction. The importance of this concept is that one need not necessarily come up with an over-arching metaphor that can include every possible circumstance that will need to be modeled in advance. Note here that we do not return to the person and briefcase message passing metaphor, although we must do so eventually. The point is that we do not need to expand on it immediately.

Note that we attempt to physically surface as many relevant attributes of the metaphor and object we are modeling as possible. This is key to the use of three-dimensional metaphor. If it cannot be seen visually, the utility of the metaphor may be greatly reduced.

Not every object or process that can be modeled is easily translated to three-dimensional metaphor, and it may often not be desirable to do so. Also, it is probable that many modeling or programming tasks have no metaphor that is a "perfect fit." We do not claim that all things should be translated to metaphor. The reason for this is especially evident when the programmer must deal with the "nuts and bolts" of the detailed implementation of a given program. No metaphor is appropriate or metaphor is simply unnecessary when these situations are encountered. This barrier will remain until such time as programmers no longer need to deal with the details of traditional programming. Rather, we claim that the use of three-dimensional metaphor can be of great aid and may simplify the programming task in some instances. However, a programmer must first keep his or her mind open to the possibility.

3.4.2. Implementation and Technical Issues

First and foremost, VRML is not an object-oriented language. In fact, VRML seems to have no identifiable paradigm, if even one is possible. This makes it difficult to achieve object-oriented design goals with VRML. PROTOs, ROUTEs, and scripts alone are not enough to satisfy these goals, because there are few restrictions on their structure and function.

The standards for VRML are widely and differently interpreted, resulting in different and sometimes unpredictable behavior between browsers. This is especially evident for event ordering and handling. Overall, it becomes extremely difficult to produce consistent behavior between browsers, and programmers of VRML may find themselves inconvenienced by the necessity of adapting their code depending on the type of browser being used.

There is no good three-dimensional editing tool that allows one to "program" VRML using three-dimensional graphics. Ideally, we would like to be able to write a PROTO, place it in a "toolbox," and then later gain drag-and-drop access to that PROTO through a three-dimensional editing tool. Then we would like to connect ROUTEs like "pipes" to and from our different objects and PROTOs in the three-dimensional editor. There is no such editor – it remains to be developed.

Lastly, VRML is a translated language. As complexity of a given world increases, there can be a noticeable drain on computer resources, even with very fast PCs. Also, there are no good debugging tools for VRML, as browser error messages can be cryptic. This has the effect of lengthening the design-debug cycle.

ACKNOWLEDGMENTS

We would like to thank the following funding sources that have contributed towards our study of modeling and simulation: (1) Jet Propulsion Laboratory under contract 961427 *An Assessment and Design Recommendation for Object-Oriented Physical Modeling at JPL* (John Peterson and Bill McLaughlin); (2) Rome Laboratory, Griffiss Air Force Base under contract F30602-98-C-0269 *A Web-Based Model Repository for Reusing and Sharing Physical Object Components* (Al Sisti and Steve Farr); and (3) Department of the Interior under grant 14-45-0009-1544-154 *Modeling Approaches & Empirical Studies Supporting ATLSS for the Everglades* (Don DeAngelis and Ronnie Best). We are grateful for their continued financial support.

REFERENCES

1. Graphics, Visualization & Usability Center at Georgia Tech, "Software Visualization," http://www.cc.gatech.edu/gvu/softviz/
2. Graphics, Visualization & Usability Center at Georgia Tech, "Information Visualization," http://www.cc.gatech.edu/gvu/softviz/infoviz/infoviz.html
3. K. Kahn, "Generalizing by Removing Detail," in *Communications of the ACM* **43**(3), pp. 104-106, March 2000.
4. H. Lieberman, "Programming by Example," in *Communications of the ACM* **43**(3), pp. 73-74, March 2000.
5. B. Meyer, *Object-Oriented Software Construction Second Edition*, Prentice Hall PTR, Upper Saddle River, N. J., 1997.
6. B. A. Myers, R. McDaniel, and D. Wolber, "Intelligence in Demonstrational Interfaces," in *Communications of the ACM* **43**(3), pp. 82-89, March 2000.
7. M. Najork, *Programming in Three Dimensions*. Ph D thesis, University of Illinois at Urbana-Champaign, Urbana, I. L., 61801, 1994.
8. A. Repenning and C. Perrone, "Programming by Analogous Examples," in *Communications of the ACM* **43**(3), pp. 90-97, March 2000.
9. D. C. Smith, A. Cypher, and L. Tesler, "Novice Programming Comes of Age," in *Communications of the ACM* **43**(3), pp. 75-81, March 2000.
10. R. St. Amant, H. Lieberman, R. Potter, and L. Zettlemoyer, "Visual Generalization in Programming by Example," in *Communications of the ACM* **43**(3), pp. 107-114, March 2000.

3.4.3. Aesthetic Issues

The current research brings out the issue of aesthetics in programming in a novel way. At the moment, programmers may discuss the aesthetics of programs that exist as text code. These discussions may include the subjects of code commenting, brace indexing, overall design efficiency, and etc. Sometimes, aesthetics may not be considered at all. With the introduction of three-dimensional metaphor, aesthetics play a larger role, and the potential develops for programs to evolve into a type of visual art.

In the present example, some effort is made to have the program interact with and entertain the user. This is done with view changes, animation, and sound. It should be noted that the programmers are not artists, but rather use some creativity that might not otherwise have been called for. For programmers that would rather not consider this issue, it may be ignored, or it may prove to be a hindrance to using the method altogether. Others may view it as an opportunity to explore and pioneer a new art form.

3.4.4. Psychological Issues

The utilization of three-dimensional metaphor necessitates that the programmer thinks in an object-oriented fashion. When the program and its operation will ultimately be represented in three dimensions with concrete and familiar objects, the programmer has little choice but to consider this during the programming process. Still, the programmer must make a mental effort to keep this in mind, and there is some extra work involved in invoking three-dimensional metaphors. Additionally, a programmer must now be concerned with problems associated with visualization. Finally, since we are extending the concept of object-oriented design, a very slight paradigm shift is involved.

With the freedom to choose any type of metaphor, including a combination of metaphors to represent a program, we also introduce the possibility that there can be "too much freedom." In other words, great freedom could result in poor structure, or vacillation over what is "the best metaphor." A programmer must be disciplined enough to adhere to good design principles and be creative at the same time.

Two prime concerns for programmers are the desire to do interesting things and the desire to do them efficiently and/or expeditiously. While the present example may be interesting, the efficiency and expeditiousness of it for designing and implementing an actual, practical operating system task scheduler is beyond discussion. This is because the design task, minus the three-dimensional metaphors (that of designing an OS and its task scheduler), had already been largely completed prior to the beginning of the project by virtue of the fact that we had chosen MINIX as a rough model. The design and generation of a novel program would better demonstrate the utility of three-dimensional metaphor in the design task. As far as implementation is concerned, the visualization itself would make the scheduler far too slow to perform the scheduling task efficiently. Eventually, the visualization would probably be stripped away to boost efficiency (if the visualization is not an integral part of the program), perhaps only to resurface again during debugging efforts.

Ultimately, the utility of three-dimensional metaphor will need to be judged from the results of empirical research. Such research might concern itself with both tangible and psychological factors. These may include such things as gains or losses in efficiency and correctness compared with traditional methods, length of design and debug cycles, programmer memory retention of program structures and design after varying periods of time, and ease and enjoyability of use.

4. CONCLUSIONS

Through the creation of an example operating system task scheduler program using VRML, we have explored the utility and viability of the multimodeling and metaphor aspects of the Rube paradigm. This approach involved modeling components of the task scheduler as a QNET, and in the near future will incorporate FSMs inside the tasks. Further, appropriate metaphors and their mappings to real-world objects were used to aid in the design and visualization of the system. The issues, advantages, and disadvantages of the approach were discussed. It was found that the method was both useful and viable for the example program and that the general method was worthy of further research and extension.

The overall goal of the research was to take some of the first steps toward an approximation of a three-dimensional, visual programming language. The value of such a language, in part, is that it could simplify the programmer's task in many ways. This is because the programming would take place in a visual environment that is more like the real world in which we live than the traditional text medium. Only time and further research will determine how this goal can be achieved.

A Three-Dimensional Synthetic Human Agent Metaphor for Modeling and Simulation†

John F. Hopkins and Paul A. Fishwick, *Senior Member, IEEE*

Abstract—**The use of metaphor can be a potential aid to the novice modeler in several ways. Metaphor can imbue abstract ideas with concrete properties, thereby making the abstract ideas more accessible. The analogies suggested by metaphor might also aid reasoning about modeling and implementation problems. Another potential benefit of metaphor in modeling is the improvement of mental retention of model architecture and functionality. Traditionally, models and programs have been produced in a two-dimensional or textual medium. However, these media may be inferior to a three-dimensional medium in the development and use of metaphor, as the concrete properties that metaphors often provide are real-world phenomena, which are naturally three-dimensional. We developed an example of the use of metaphors in modeling and three-dimensional simulation. The example consists of a simplified operating system task scheduler, along with associated hardware devices, developed in a VRML environment using VRML PROTO nodes. These nodes are designed as modular objects based on real-world metaphors. We were able to construct a set of metaphors and prototypes that may, if extended, ease the modeling and design of agent-oriented systems for novices. A proposed extension of one metaphor presented in the research is the *synthetic human agent.***

Index Terms—**Metaphor, Visual Programming, 3D, Programming, VRML, rube™**

I. Introduction

THE act of programming began as a mathematical activity, extending from machine code to assembly language, and then onto languages with humanly readable text with natural language variable and function names. In 40 years, programming has evolved from its abstract roots and yet text is still the primary medium in which programs are created, often with shorthand notations for program components. For economic reasons, there has always been a strong tendency toward abstraction in computer science and this is evident in the nature of modern programming languages. For example, the process of iteration (i.e., looping) is accomplished in a way that has not changed much in all these years. The same can be said for most other programming constructs, even though we have come far with new design paradigms such as component, agent, and object-based software architectures.

We present a method of modeling and programming that is significantly different than many others, although some of our techniques are shared by many modeling paradigms (e.g., agent-based [20], swarm-based [4], and object-oriented [29]) and several burgeoning areas such as Software and Information Visualization (see [39] and [5]), Programming by Example (see [22], [25], [34], [37], and [38]), and Visual Programming (see [1], [9], [23], [24], and [31]). These efforts, while diverse, all have one thing in common – they suggest new modeling and programming techniques that are based on the real world. We have built upon this general idea to create a paradigm called *rube*™ (after Rube Goldberg), in which real-world objects are integral parts of the language. The *rube*™ paradigm centers naturally within the domain of modeling for novices, and may progress eventually (as with the case of the *desktop metaphor*) to assist experts.

Our programs are *multimodels* (see [11], [12], [14], and [16]) where a *multimodel* is defined as a hierarchically connected set of behavioral models, each of a specific type. The basic types are numerous and include finite state machine (FSM), functional block model (FBM), Petri net (PNET), and queuing net (QNET) (see [12]). These base behavioral model types and their components are mapped to an arbitrary domain using metaphor. The choices of domain and metaphor are artistic choices that are left to the programmer. We use the Virtual Reality Modeling Language (VRML) as our language for encoding these models and bringing them to life.

In the current research, we demonstrate and examine the interplay between the multimodeling and metaphor aspects of the *rube*™ paradigm through the synthesis of an example simulation. In Section II, we conduct a short survey of related research and contrast it with the current research. We create a greatly simplified operating system task scheduler using the *rube*™ multimodeling and metaphor methodology in Section III. In Sections III and IV, we discuss the issues, advantages, disadvantages, and extensions of the approach.

John F. Hopkins and Paul A. Fishwick are with the Department of Computer Science and Engineering, University of Florida, Gainesville, FL 32611

† This paper is based on work reported in [18].

II. BACKGROUND AND RELATED RESEARCH

Background includes the consideration of the use of metaphor in software development (Part A), and the possible specific applications of the *synthetic human agent metaphor* to Software Agent systems (Part B). We also consider object-oriented design philosophy (Part C), Software and Information Visualization (Part D), Visual Programming (Part E), and work done with three-dimensional (3D) authoring tools (Part F). Next, we briefly sample some other 3D programming research (Part G). Then we compare and contrast the aforementioned research with the current research (Part H). Finally, we compare the current research with some of our own previous research in Object-Oriented Physical Multimodeling (OOPM) (Part I).

A. *The Use of Metaphor in Software Development*

Before we begin a discussion of the use of metaphor in software development, let us first give a formal definition of the word *metaphor.* According to *Merriam-Webster s Collegiate Dictionary* (10^{th} ed., 1998) a *metaphor* is "a figure of speech in which a word or phrase literally denoting one kind of object or idea is used in place of another to suggest a likeness or analogy between them." This definition is admittedly shallow, but the operative phrases are "object or idea . . . in place of another to suggest [an] . . . analogy between them." In sum, the key to the power of metaphor is the suggestion of analogies between ideas and objects.

From the above, it may be intuitively concluded that metaphors help people communicate their ideas to others, view ideas from different perspectives, and organize knowledge. In the field of computer science, users are concerned with organizing information and communicating instructions to computers through various interfaces. The commonly termed *desktop metaphor* is one ubiquitous example of the use of metaphor in operating system (OS) interface design.

The usefulness of metaphor in programming (visual or otherwise) is a contentious issue in a wide variety of research literature [2]. Part of the problem is that the simple definition of *metaphor* given above belies the complexity of the volumes of active multi-disciplinary research that is devoted to it [41]. Since there are several perspectives on the nature and operation of metaphor, the quantification of the effects of metaphor becomes a non-trivial task. To further complicate matters, there are several different ways to define *usefulness.* Productivity is only one of many possible measures of usefulness.

Even though we may not be able to come to a consensus on the nature and operation of metaphor, research concerning the use of metaphor in programming continues. Some research suggests that intuitive, optimistic expectations of the effectiveness of metaphor in programming are not well supported by empirical studies [2]. The benefits of using metaphors seem marginal at times, and positive effects may be attributable to other factors, such as direct manipulation of icons or the effects of using diagrams [2]. However, the use of metaphor in diagrams has been shown to provide some mnemonic assistance, which appears to be greatest when the user of the diagram constructs her own metaphor [2].

Since the "metaphor usefulness issue" is complex and beyond the scope of the paper, we might instead pursue the question: "Can we show that the use of metaphor doesn t hurt?" It has been claimed that some commonly used metaphors in computer science do not fit well, and so these metaphors cause people to draw incorrect analogies [32]. Some evidence also suggests that the use of concrete representations can limit the formation of abstractions in learning programming [9]. However, we suggest that if metaphors are thoughtfully crafted to fit well, and we do not cling too tightly or overly long to them, these dangers can be avoided.

Despite the unclear conclusions described above, metaphors are commonly used by experienced computer scientists to express otherwise complex concepts to those whom have little or no background knowledge to draw upon. And, being a part of everyday speech patterns, metaphors have also worked their way into communications amongst computer scientists (expert or not). For example, we *flush* buffers to empty them of their contents, and computers can have *viruses.* We use these metaphors because they seem to fit well, and they provide us with some economy and power of representation. Research concerning the use of metaphor in the communication patterns of software developers that must work in groups and with end-users seems to be scarce. Considering the central role that communication plays in facilitating teamwork, this seems to be an unfortunate omission.

B. *Application of the Synthetic Human Agent Metaphor to Software Agent Systems*

During the Dagstuhl Workshop [15], we explored a number of issues regarding the relationships between agents and M&S (modeling and simulation). This simple use of metaphor comes to the rescue in demonstrating a connection: Let the agents be human in appearance and in action, and then have the agents operate within a micro-world common to agents, such as a business workplace.

Let us examine more closely some intuitive reasons for creating a metaphor that suggests an analogy between humans and agents. First, we must define what a *software agent* is. Since we know of no agreed-upon, formal definition for this term, we provide a functional definition by reviewing the various characteristics and abilities that have been used to describe software agents in research literature. Some of these characteristics and abilities

include 1) the autonomous representation of another entity, 2) the ability to communicate, interact, and collaborate with other entities, 3) the ability to perceive, move through, react to, and alter a complex, dynamic environment, 4) the ability to reason, learn, and adapt, and 5) the ability to take initiative in pursuing goals [17], [20]. The foregoing list is neither definitive nor exhaustive. However, a cursory comparison of these characteristics with some of the defining functional characteristics of a typical human being will reveal that the list could have been entitled "Some defining functional characteristics of a typical human being," and readers should not have noticed the difference. Furthermore, a brief survey of agent literature reveals that the functions of agents are commonly analogous to the functions of human beings in the real world. Therefore, it is not a great stretch of the imagination to metaphorically equate agents with human beings. Thus, the *synthetic human agent* is born.

By placing synthetic human agents in the business workspace environment previously mentioned, we create a *business workflow metaphor.* This metaphor can be mapped to any model, process, or procedure whose elements involve queued entities, resources, and pathways. Additionally, this approach creates a direct relationship between M&S and agents – namely, agents are thought of and represented as synthetic humans.

In this research, we create synthetic human agents to act as representatives of operating system (OS) tasks inside a simplified task scheduler. However, we stress that these agents are not perceptive, intelligent, or autonomous. Rather, they may be considered to be agents inasmuch as they serve as representatives of another entity (i.e., they are subtasks of a higher-level task), and they maintain their own internal state information. At this point, some readers may reasonably reject the notion that these aspects of our synthetic human agents are sufficient to qualify them as *agents*. However, we hope that these readers will bear with us a while longer while we expand the concept and, so doing, show that our synthetic humans have the potential to lay a stronger claim to the title of *agent*.

For example, consider a *software agent OS* in which several synthetic human agents may be used to represent the same task within disparate portions of the OS (e.g., file system manager), and within a computer s hardware (e.g., memory). In this environment, a single task may possess several agents throughout the system. The agents common owner – the task itself – would link these. This use of a synthetic human as an agent of a higher-level task could potentially involve a shift away from the idea that tasks are simply manipulated by an OS. Rather, tasks and their agents within the system could play a more active role as seekers of services. Traditionally, tasks have no perception of the OS, but rather the OS perceives and manages tasks. For the tasks and their agents to become more active in these respects, a possible course of action is to endow them with perceptive abilities and varying degrees of autonomy. Without doubt, this can be considered as a radical departure from traditional views of monolithic OS functionality (i.e., the OS is a resource manager and user interface). Less controversial is the notion that software agents may inhabit a physically distributed system (OS or other), in which they autonomously travel about between different hosts in their quest to perform tasks (i.e. mobile agents, [20]). To reduce the contrast between these ideas, we might assert that there is little difference between the distributed system and the monolithic system from an abstract point of view – after all, they both contain tasks and physically distributed hardware devices. To raise the contrast, we might consider pragmatic issues such as network delays, bandwidth, and security. We suggest that the out-of-hand dismissal of unusual abstract thinking incited by the use of metaphor may rob us of potentially useful and interesting insights like those above. The inspirational metaphor in this case is the synthetic human agent.

How might synthetic human agents help novices understand difficult modeling scenarios? In short, the agent-human metaphor provides novices with a ready-made *storehouse* of conceptual analogies and mappings that exist between agents and humans. These analogies can be used to simplify thinking about problems in modeling, design, and sometimes implementation of solutions. For example, assume that we are assigned the task of designing a distributed system that is implemented over a wide area network. Also assume that the system contains mobile agents. Time-tested processes employed to simplify thinking about such a potentially complex system are abstraction, modeling, and visualization. The nodes of the network could be modeled and visualized as a web of spheres interconnected by pathways. Synthetic human agents could be used to represent the mobile agents described above. These synthetic human agents could physically move over the paths between the spheres in an effort to extract services from the nodes, while the visualization of this process reflects the position of each agent in the network with respect to the nodes and pathways. Additionally, a great deal of information can be gleaned from this sort of visualization. For example, users may be able to literally see "broken" links between nodes, find "misplaced" agents, and derive an intuitive knowledge of system load and activity. Admittedly, the types of knowledge listed do not *require* the use of the synthetic human agent metaphor. However, the metaphor can give users the visual cue necessary to differentiate agents from other objects in the world, which is an important aspect of determining the functions of agents, objects, and the relationships between them.

Let us give another example that shows a more direct use of the synthetic human agent metaphor. A common problem area in agent research is that of security, especially in the context of mobile agents [19], [35]. How might the synthetic human agent metaphor help novices think about this problem? They could draw an analogy between the nodes of the network and buildings in the real world. Mobile agents travel amongst the nodes of the network just as humans travel amongst buildings. How do humans gain

access to buildings? They pass through the doors, and if the building is secure, the doors are locked. However, if a human has a key to the door s lock, then he or she can get into the building. This suggests that *locks* should be placed on the nodes of the network at their communication access points (software or hardware), and specific agents should be granted access to these nodes with *keys*. This might be enough to satisfy a novice modeler, but admittedly it is not nearly enough to satisfy a more advanced modeler or provide an adequate architecture for airtight security. Still, the analogies can be expanded, and a better solution might be developed by incorporating public key techniques – yet another construct that is best explained to novices through the use of metaphor and visualization. Solving the problem of security in a network that contains mobile agents is not our focus. Rather, our focus is on suggesting a mode of thinking about modeling and design problems that is inspired by metaphor and visualization, and that remains familiar and clear to novices because it accords with everyone s everyday experience. In the present case, we have chosen to use a synthetic human agent metaphor, and this metaphor is one of many that might have been chosen by a novice or expert. It just happens that the synthetic human agent metaphor seems to suggest analogies that draw close parallels between humans and agents for reasons previously elucidated.

It has been stated that it is difficult, if not impossible, to *prevent* an attack on a mobile agent that is operating on a malicious host [19]. This is because the host completely controls the environment and operation of the agent [19]. A novice modeler might easily surmise the foregoing technical fact by using the synthetic human agent metaphor in the following fashion. We, as human beings, possess our own bodies. Mobile agents, on the other hand, must rely on hosts to provide them with a *body*, or computational ability, when they are away from their home machine. Thus, there is no way to prevent a malicious host from tampering with the *bodies* of whatever agents it is hosting. This is a case when the synthetic human agent metaphor seems to break down. However, this seeming failure of the metaphor to provide an exact analogy for the persistence and security of a mobile agent is as instructive as any successful analogy the metaphor might suggest. According to the suggested analogy, then, the ultimate form of security for a mobile agent would be for it to stay on its home machine, where its *body* is safest. Obviously, this requires the agent to be immobile. It seems that either we must acknowledge that it is impossible to prevent attacks on mobile agents and do our best to detect and minimize the effects of these attacks, or we must make our agents immobile. Again, the object here is not to suggest forms of security for mobile agents or launch an attack against the concept of mobile agents, but rather to demonstrate a style of reasoning that novice modelers might develop through the use of the synthetic human agent and related metaphors.

The synthetic human agent metaphor may also prove to be useful in the development and visualization of agents that must interact with humans (i.e., interactive agents. [20]). An example of such work is the *believable agent* described in [26] and [28]. It seems obvious that if human-behaved agents are ever to interact with humans on a regular basis, these agents should ideally have human-like physical intricacies and inhabit a 3D world. Of course, this sort of thing will not happen in the immediate future since it requires great leaps forward in many research areas, but there seems to be good reason to believe that the field will eventually evolve in this direction.

Indeed, at least one commercial agent development company, *ReyesWorks* (http://www.reyes-infografica.com/), has already incorporated a synthetic human agent metaphor in its agent development and visualization tools. See Fig. 1 for an example of *synthetic human agent* visualization provided courtesy of *ReyesWorks.* Another company, *Blaxxun Interactive* (http://www.blaxxun.com/), currently operates *Cybertown* (http://www.cybertown.com/) in which human users and their autonomous *bots* possess *avatars* that interact with each other within a 3D virtual world.

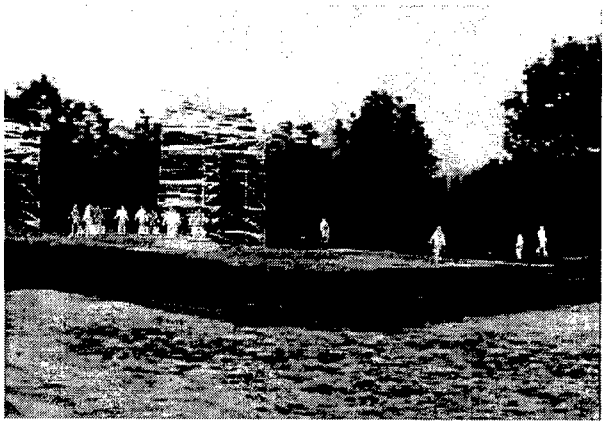


Fig. 1. Photo of agent visualization provided courtesy of *ReyesWorks.* Reprinted with permission.

C. *Object-Oriented Thinking and Design*

In part, the object-oriented approach to software design and construction is concerned with modeling. Specifically, it suggests that before any textual code is written, a model of the software s functionality should be created which will guide and inform the rest of the process in a *top-down* fashion – from abstract/general to concrete/specific [29]. In this effort, modularization is a primary design criterion [29]. In a broad sense, and as suggested by the term *object-oriented*, these modules may be viewed as individual objects, each with their own functions, interfaces, attributes, and allowable operations. These objects confer many benefits on the programmer, as they facilitate the easy management of what otherwise might be significant complexity, simplify the debugging process, and create the potential for code re-use, to name a few [29].

It is not difficult to see the parallels between object-oriented design and the world in which we live, both in an abstract and a physical sense. In fact, a common way used by instructors to teach novices to understand the object-oriented approach is to draw parallels between it and the real world. For example, when we are patrons of restaurants, we deal with the waitstaff, and not directly with the kitchen. In an object-oriented world, the waiter or waitress is an object, we are objects, and the kitchen is an object. The waitstaff serves as our interface to the kitchen. We would violate the modularity enforced by the commonly termed *client-server* relationship if we attempted to deal directly with the kitchen as patrons. In fact, we do not even need to know that the kitchen exists; the server can simply deliver our food without our knowledge of where the food came from or exactly how it was made. This example also embodies the closely related ideas of *information hiding* (i.e., the process of hiding details of an object or function) and *encapsulation* (i.e., the inclusion of a thing X within another thing Y so that the presence of X is not apparent) [29].

This suggests a form of design and programming in 3D, where these parallels can be surfaced visually during the design and programming process. At the moment, programmers typically may use pencil and paper to make sketches of program modules and the relationships between them, use visual programming tools, visualize these modules mentally, or do none of the above. Some may proceed directly to text coding. However, a characteristic of all the methods mentioned above is that they are implemented in two dimensions or less. If we suppose that we are physical creatures that live in a 3D world, it may follow that a better approach to design and programming lies in the utilization of a 3D design and programming environment. Such an environment would allow for the incorporation of geometry and behavioral dynamics into object-oriented design.

Object-oriented design currently has no specific relationship with the use of 3D metaphors in programming. However, a frequent occurrence during the design process is that one or more abstract data types or functions are created. If we must attempt to visualize an abstract function such as a sorter, we may arbitrarily decide to visualize it as a green pyramid. However, if it is possible to visualize the sorter as an animated person sorting boxes, we have a way to create a visual metaphor for the sorter. If we use the animated sorting person, we have made an analogy between the abstract sorter and the concrete, real-world person. If the sorter is visualized as a person in this fashion, there is no need to memorize the previous mapping of the green pyramid to the sorter. The visualization of the person and the metaphor introduced now provide this mapping implicitly. In effect, the visualization provides a semantic clue as to the object s function. This sort of use of metaphor, then, may find great utility during design, debugging, and code maintenance. It would be difficult to support the argument that the green pyramid visualization is preferable to the animated sorting person on grounds other than the additional effort it would take to produce the animated person. Finally, we predict that the extra effort required to produce a metaphor visualization should decrease to a minimal level with time and advances in technology, so that a cost/benefit analysis should eventually become favorable.

D. Software and Information Visualization

Software Visualization is primarily concerned with using computer graphics and animation to illustrate programs, processes, and algorithms [39]. Software Visualization systems are sometimes used as teaching aids and also during program development as a way to help programmers understand their code [39]. Information Visualization is primarily concerned with visualizing data as an aid in interpreting and understanding the information [5]. In general, the focus of Software and Information Visualization is on visualizing programs or data after they have been created using traditional means.

Several problems worthy of study have been encountered during visualization research. These include problems of abstraction of operations, data, and semantics, levels of detail, scaling, and user navigation [39], [5]. Additionally, visualization may occur in two dimensions, three dimensions, or a combination of these. Obviously, there are myriad ways to visualize programs and data, and perhaps one challenge of visualization research is to find the most usable and efficient ways of accomplishing this task. Any two-dimensional (2D) or 3D programming method or environment must tackle many of the same issues encountered in visualization research. This suggests that visualization is a sub-problem of any such development system, either implicitly or explicitly.

E. Visual Programming

The field of visual programming integrates research from other fields such as computer graphics, programming languages, and human-computer interaction (HCI). One aim of visual programming is to reduce the amount of text-based programming and increase the use of image-based programming. Two speculated benefits of such a migration would be increased productivity of computer users, and greater accessibility of computers to a wider general audience. These speculations are based on the observation that many people think and remember things in terms of images. People relate to the world in an inherently graphical way and use imagery as a primary component of creative thought [36].

For many people, textual programming languages can be difficult to learn and awkward to use. Programmers are often faced with the problem of translating visually formulated ideas into textual representations. Visual programming could ease the language learning process and

help in the translation of visual ideas to the computing environment by keeping ideas, to a degree, in the visual domain. Also, visual development methods seem intrinsically appropriate for the creation of interactive simulations and scientific visualizations.

A visual programming language system may be purely visual or may be a hybrid text and visual system. Purely visual languages are characterized by their exclusive reliance on visual techniques throughout the programming, debugging, and execution process. An example of a commercially successful and completely visual system is *Prograph* [7]. In one type of hybrid system, programs are created visually and then translated into a high-level textual language. An example of this sort of hybrid system is *Rehearsal World* [10].

Although visual programming emphasizes an effort toward visual interaction in programming, this effort is not likely to eliminate the use of text for practical reasons. While it is possible to represent all aspects of a program visually, such programs are less transparent than those that occasionally use text for labels and atomic operations. Further, limited use of text should not detract from visual representations.

F. Work in VRML and Other 3D Authoring Tools

Modelers must choose the number of dimensions they prefer to use when representing the operation of their models and programs. Often, they decide to make use of two dimensions or less. What are the advantages of using the third dimension? First and foremost, we appeal to the "real world" arguments that we have been using to justify the synthetic human agent metaphor. Humans exist in a 3D world – why shouldn t agents? Economic arguments concerning the high cost of 3D rendering hardware are becoming unfounded due to the blistering pace of technological advancement and the resulting rapid drop in hardware prices. Second, the third dimension simply offers more space and freedom than two dimensions. Many of the scalability and complexity problems associated with 2D visualization can be overcome by the layering that is made possible through the use of 3D visualization [42]. Of course, 3D visualization introduces an additional set of problems, such as managing user navigation, orientation, and viewpoint [42]. However, as opposed to 2D visualization, the problems associated with 3D visualization seem to be solvable given appropriate engineering effort.

A more practical argument against 3D visualization is that it may introduce unnecessary complexity, especially in cases when the modeling scenario is simple. We do not resist this assertion. In certain cases, 2D visualization or no visualization may suffice to meet modeling and design goals. However, we suggest that 3D visualization can provide intrinsic benefits for novice modelers and in large or complex modeling scenarios. In other instances, 3D visualization can serve as a valuable augment to more traditional visualization methods [23].

Many 3D geometry-authoring tools have been developed, such as *CosmoWorlds* and *3D Studio MAX* (Kinetix). These tools are appropriate for creating 3D environments and objects, and also for modeling relatively simple behaviors of these environments and objects. Occasionally, environments and objects created in this fashion are endowed with the ability to interact with the user. Artists are one community that make frequent use of these tools in their work, as it provides them a relatively cheap, reusable, flexible, and unique medium. Geometry, sound, light, color, movement, and user interaction can be combined through the use of these tools.

VRML (Virtual Reality Modeling Language) is a 3D language that is based on the concept of a tree hierarchy of nodes. Some reasons for its popularity include the fact that it is non-proprietary, portable, and many browsers for it are freely available. Another reason is the proliferation and accessibility of the Internet. An approximation of *generic* and *object-oriented* programming can be achieved through the use of a type of VRML node called `PROTO` (short for *prototype*) and the use of a type of structure called `ROUTE` that can be conceptualized as a *pipe* that connects structures to events. Code for behaviors of objects can be written in ECMAScript, which is an ISO standard for the more commonly known JavaScript. These attributes, among others, make it attractive for an early attempt at development of a 3D programming environment. VRML isolates the user from the details of specifying and rendering 3D geometry and thus it is more attractive for novices than using, for example, the Java3D API. Also, VRML seems to have commercial and developer momentum in the X3D and XML initiatives that Java3D seems to somewhat lack. This is not to assert that Java3D is not a viable platform – some interesting work incorporating VRML and Java3D in distributed agent simulations is proposed in [21]. Despite its advantages, the use of VRML is not without problems and difficulties, especially concerning the implementation of data structures and complex behaviors that are typical of modern programs. Specific drawbacks of VRML will be discussed in more detail in Section III, Part E, Sub-part 2.

An important feature of VRML, from the perspective of *rube*™, is the ability to create custom prototype nodes that can be reused. Another important feature is the ability to incorporate scripts inside these custom nodes that imbue them with a great range of physical and/or functional behaviors that are not native to the VRML language. Lastly, VRML provides `ROUTE`s for communicating the occurrence of events throughout the world. These events drive behaviors of objects in the world.

Since *rube*™ is primarily implemented in VRML and ECMAScript, we provide a short tutorial below. The tutorial is a more in-depth treatment of the language for

those that might wish for one, and can be safely skipped (to Section II, Part G) by readers who are more interested in the abstract ideas behind the *rube*™ development process. Note that this tutorial is not extensive enough to allow a novice user to begin programming in VRML and ECMAScript. However, it should suffice to give the reader general ideas that capture the structure, operation, and interaction of VRML and ECMAScript.

A VRML world is made up of nodes, which are types of object. Nodes contain fields, which are properties of the object. Fields can be anything, from a size of a box, to another node inside the first. Some nodes have a `children` field, which can contain other nodes. Nodes can be nested inside one another in this way, which yields a kind of hierarchy tree of nodes.

The specification of geometry in VRML is paramount, so we give a generic geometry declaration below. There are a few nodes shown, such as `Appearance` and etc., but these can be ignored for present purposes. To add a shape to the world, we might add the following to a VRML file:

```
Transform
  {
  children
    [
    Shape
      {
      appearance Appearance
        {
        material Material {}
        }
      geometry Box {}
      }
    ]
  rotation 1 1 0 0.785
  }
```

The above adds a `Shape` node, which contains two fields for describing an object. These are the `appearance` and `geometry` fields, which describe the look and shape of the object, respectively. The `geometry` field specifies a `Box` with default values and colors (default values are assumed by the VRML interpreter when field values are omitted or left empty by the programmer). Fig. 2 shows the box we added to the world, slightly rotated about the x and y axes.

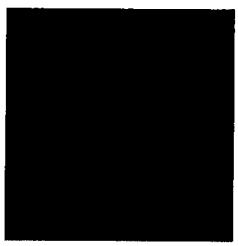


Fig. 2. A VRML box rotated about the x and y axes.

A common type of VRML node is the `Transform`, whose partial language specification is given below:

```
Transform
  {
  eventIn      MFNode  addChildren
  exposedField SFVec3f center 0 0 0
  field        SFVec3f bboxSize -1 -1 -1
  }
```

The interesting features of this node are `eventIn`, `exposedField`, and `field`. An `eventIn` of any node may be thought of as a *hook* for incoming events. In the `Transform`, the `eventIn addChildren` accepts input of type `MFNode`. An `MFNode` is one of several primitive data types available in VRML. When an `MFNode` arrives at the `eventIn addChildren`, its contents are added to the `Transform`s children. If the incoming `MFNode` contains a `geometry` node of some sort, this will then appear in the world, whereas it may not have appeared in the world at all until the event arrived. If the `Transform` node moves later, its new child will move along with it. A second notable item in the `Transform` node specification is the `bboxSize` (bounding box size) `field`. A `field` is simply a value holder. Since it is not an `eventIn`, the value of a `field` is set when it the node is declared and cannot be changed afterward, unless it is coupled with an `eventIn` through scripting (described shortly). Also, the VRML interpreter does not make the value of a `field` accessible outside of the node. An `exposedField` can be thought of as an integration of an `eventIn`, `field`, and `eventOut` (to clarify, `eventOut` is not an explicit element of the `Transform` node specification). The VRML interpreter makes values of `exposedFields` accessible outside of a node. An `exposedField` can be altered in a way similar to an `eventIn`. This is accomplished by specifying the `exposedField` to be altered with a `set_` prefix, (e.g., `set_center`). When an event arrives at an `exposedField`s `eventIn`, its corresponding `field` value (e.g., `center`) is changed. As soon as the `field` value is changed, the `exposedField` sends an `eventOut` to the `field` name followed by a `_changed` suffix (e.g., `center_changed`). The `eventIn` and `eventOut` field names (`set_`*xyz* and *xyz*`_changed`) are implicitly implemented for all `exposedFields` by the VRML interpreter, so they need not be explicitly declared upon node construction, for example when the user is specifying custom prototype nodes (`PROTO`s).

An example "toy" `PROTO`, `FOO`, is shown below to aid the discussion. In the declaration portion (header), the `eventIn set_number` will receive integers of type `SFInt32`, the `number` field will hold a value of `SFInt32`, and `eventOut number_changed` will send integers of type `SFInt32`. The `exposedField` `dummyval` operates like any other `exposedField`. To

demonstrate the implementation of custom behaviors, we include a script in the FOO PROTO:

```
PROTO FOO
  [
  eventIn       SFInt32  set_number
  field         SFInt32  number  0
  eventOut      SFInt32  number_changed
  exposedField SFInt32  dummyval 9
  ]

{
Transform
  {
  children
    [
    DEF MyscriptS Script
      {
      eventIn SFInt32 set_number IS
                                set_number
      field SFInt32 number IS number
      eventOut SFInt32 number_changed IS
                            number_changed

      url "javascript:
        function set_number(value, ts)
          {
          number = value;
          number_changed = value;
          }"
      }
    ]
  }
}
```

This simple PROTO introduces some new concepts to our discussion. The first of these is the DEF statement (i.e., DEF <ArbitraryNodeName> <node type>). DEF (short for *define*) names a node for future reference and for reuse in the same world. A second concept introduced is the Script declaration. Note the use of identical field names in the Script declaration as those of the PROTO declaration. In fact, the use of the IS modifier makes these fields identical to the fields of the PROTO. For example, if an event arrives at FOO s eventIn set_number, the event is also passed to MyscriptS s corresponding eventIn of the same name. Also, if MyscriptS produces an event for the eventOut number_changed, FOO also sends this event to its eventOut number_changed. A third concept introduced is the JavaScript function set_number(value, ts), where value is the SFInt32 value that is passed in to the function, and ts is the VRML timestamp for the event. The set_number function has the same name as the eventIn set_number of MyscriptS. The effect of this is that when an event arrives at the eventIn set_number of MyscriptS, the set_number function is called; the identical mapping of the function name to the name of the script s appropriate eventIn is essential. A fourth concept introduced is the use of the assignment (=) operator in the script. In the case of number = value, we have assigned the number field of the script a new value. In the case of number_changed = value, we have caused MyscriptS to produce an eventOut number_changed, that will make its way to FOO s eventOut number_changed.

Now all that remains to be described is a method for "piping" events into and out of nodes. In VRML, this is accomplished through the use of ROUTEs. An example implementation of ROUTEs in a short, top-level VRML file might look like this:

```
EXTERNPROTO FOO
  [
  eventIn       SFInt32  set_number
  field         SFInt32  number
  eventOut      SFInt32  number_changed
  exposedField SFInt32  dummyval
  ] "foo.wrl"

{
DEF F1 FOO {}
DEF F2 FOO {}
}

ROUTE F1.number_changed TO F2.set_number
ROUTE F1.dummyval_changed TO
                              F2.set_dummyval
```

In the above, we have first declared to the VRML interpreter to look for the FOO PROTO node definition externally (EXTERNPROTO) in a separate file (specified by "foo.wrl"). The skeleton of FOO is provided as a sort of declaration in a similar manner to the standard C convention for declaring function signatures at the head of a file. Next, we create two instances of FOO – F1 and F2. F1 and F2 have independent existences from each other (e.g., their fields can take on different values from each other). Finally, we use ROUTEs to connect F1 eventOut fields to F2 eventIn fields. When F1 performs its functions, it will send events to F2 that will initiate functions in F2. Unfortunately, the above program will not perform any function for at least one reason: We have not specified any structure or node that will initiate an event in F1 to start the chain of events we have constructed. However, the foregoing should help pull together most of the concepts that are relevant to understanding the structure and operation of a good deal of VRML PROTO and script design, and thereby *rube*™ s implementation methodology. If the reader would like further VRML tutorial, several sites on the WWW can be found for such purpose by conducting a search using one of the popular WWW search engines. For a more in-depth treatment, two good VRML references are [6] and [27].

Finally, VRML worlds are viewed with special browsers, such as *Blaxxun Contact 4.3.* and *CosmoPlayer 2.1.1.* These browsers allow the user s avatar to navigate through the 3D world and examine its contents. A prototypical browser *dashboard* (*CosmoPlayer 2.1.1.*) is shown at the bottom of Fig. 3.

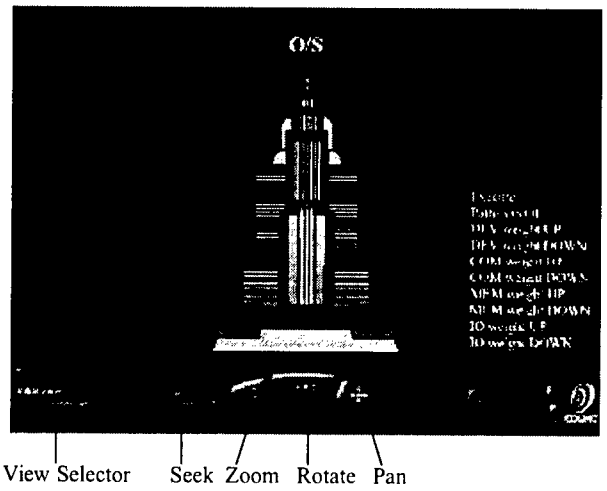


Fig. 3. Browser controls of CosmoPlayer 2.1.1.

G. 3D Programming

Two goals of 3D programming are to visualize program execution with 3D graphics [24], and to enable the user to program using executable 3D graphics. A brief review of two approaches to this problem is given below.

1) CUBE

Marc Najork s CUBE is an example of a 3D visual programming language that has a 3D syntax. In this language, programs consist of an arrangement of 3D shapes instead of a linear stream of text. CUBE supports recursion, function mapping, type checking, and user-defined types. One aim of the language is that of supporting virtual-reality based editing of programs for the user [31].

The CUBE language is divided into two fragments. The first fragment consists of predicate definitions and logic formulas, and the second fragment consists of type definitions and type expressions. 3D cubes represent terms and atomic formulas in the logic fragment. Pipes connect the arguments of predicate applications (i.e., atomic formulas), forming a data flow diagram. The third dimension is used to group various 2D diagrams together. Sets of base types and primitive predicate definitions are provided initially. The prototype implementation of CUBE functions by translating visual programs into a text format, and then operates on the text format [31].

CUBE does not make extensive use of visual metaphor. However, it does make use of the *data flow metaphor* on a high level and as a guiding paradigm for the representation of the language [31]. As described above, this metaphor also happens to extend into the 3D realm. From this, it can be seen that the use of metaphor need not be limited to its typical, everyday senses, but also can be used as a guiding framework for a form of representation.

2) Programming by Example/Demonstration

Programming by Example and Programming by Demonstration (PBE/PBD) are synonymous terms for a method of programming described by the following phases: 1) A user performs a series of steps on an example of some problem, 2) the computer records the steps, and 3) the computer applies these steps on some new, similar example. The programs generated in the recording step may not be applied exactly as they were recorded, but may adapt according to differences in future examples [25]. This sometimes requires such programs to be able to generalize the steps to new instances, and to perform inference in varying degrees, from no inference to a great deal of inference [30].

One focus of PBE is that of catering to the novice or non-programmer. For a novice programmer, there is usually a significant difference between the computer s representation of a problem and his or her own. As programmers become more experienced, they become more conditioned to think of a given problem in the context of the computer s representation, since the problem must ultimately be expressed in this fashion and there has been no efficient way to communicate the programmer s representation to the computer. PBE would bridge this gap by "bringing the computer closer to the programmer" rather than vice-versa [37].

PBE attempts to lower the barriers between novice users and computer programming by eliminating, as much as possible, the idiosynchracies of traditional programming language syntax. It attempts to provide visual, interactive, intuitive, and functional interfaces for program construction, and performs inference (to varying degrees) to reduce the amount of inference that must be performed by the programmer [30]. A recent trend in the PBE community has been to extend into the 3D programming language arena [22], [38]. This extension is natural since we are conditioned to think of the world around us in three dimensions, and so this is one way to bring the computer closer to the programmer. Attempts may be made, so far as possible, to keep programs and constructs in concrete domain terms rather than abstract computer terms [34]. In short, a goal is to let an end-user program without realizing that he or she is programming.

Ken Kahn s *Toon Talk* is a PBD system designed for children in which "the programmer is a character in an animated virtual world in which programming abstractions are replaced by their tangible analogs." Data structures are

given 3D geometry, such as a box with holes, into which different objects may be placed. Birds and nests perform *send* and *receive* operations. Robots are guarded clauses that are trained to perform actions when they receive boxes. The programmer may generalize robots by using a vacuum cleaner to remove details from the robots guard clauses [22]. It can be seen here that *Toon Talk* makes use of visual metaphor.

H. Comparison of the Current Research with Related Research

The current research, with its focus on extending programs and their components to resemble real-world objects (in an abstract, non-visual sense), has many parallels with object-oriented design approaches to programming. Since these parallels are somewhat obvious, they will not be elucidated here. However, the current research also seeks to give these objects 3D representations that convey meaning through visual metaphor. In this way it is unlike, and could be considered an extension of object-oriented design.

Software and Information Visualization research and the current research share several problems in common. These include abstraction vs. concretion, levels of detail, navigation, and many others. Without doubt, solutions to these problems that have arisen and will arise during the course of either line of research should easily translate into the other line. However, in the previous description of Software and Information Visualization research it was stated that a general focus of that research is on visualizing programs and etc. after they have been created using traditional means. Therefore, this research tends not to focus on the immediate aspects of program development, such as the potential for creating a 3D programming interface for users. Furthermore, metaphor is not commonly considered during such research. In these ways, the current research is unlike, and could again be considered an extension of Software and Information Visualization.

Work in 3D authoring tools and the current research are alike inasmuch as objects are identified by their geometry and have specific physical and behavioral attributes. The current research was carried out in part within a 3D authoring tool (*CosmoWorlds*) and in large part within a 3D language (VRML). However, the current research is distinguished from typical 3D authoring since most of the work was done in a text editor (for VRML coding), and a great deal of scripting was performed to make the behaviors of the objects in the world complex, dynamic, and interactive. The heavy use of PROTO nodes and ROUTEs in the current research also distinguishes it from straightforward VRML environments.

The current research should be considered as visual programming research – it is most like CUBE and PBE, but is significantly unlike these examples of visual programming in several respects. We did not use a 3D editor significantly during the programming effort. The "dragging and dropping" performed was on 3D objects that had only geometric representation. Unlike CUBE, there was no formal metaphor that informed a 3D syntax, nor was a set of 3D representations of primitive data types and operations available for use. Also unlike CUBE, type checking was performed exclusively by the programmer, and the text program was translated to the visual program. The use of metaphor and geometric shape in the current research was more free-form than that of CUBE. The current research is unlike PBE inasmuch as the computer did not record the programmer's steps or perform any generalization or inference. Also unlike PBE, the use of metaphor in the current research was user-defined.

I. Rube™ s Relationship to OOPM

In previous research, we have designed and implemented an object-oriented multimodeling and simulation application framework [8], [11]. OOPM (Object-Oriented Physical Multimodeling) is a system that resulted from this research. OOPM extends object-oriented program design through visualization and a definition of system modeling that reinforces the relation of *model* to *program.* One feature of OOPM is its graphical user interface (GUI), which captures model design, controls model execution, and provides output visualization. Another feature is its Library, which is a model repository that facilitates collaborative and distributed model definitions, and that manages object persistence. Some dynamic model types that OOPM models are composed of include Finite State Machine, Functional Block Model, Equation Constraint model, and Rule-based Model. Model types are freely combined through multimodeling, which glues together models of same or different type [8].

OOPM research evolved into research with *rube*™, and OOPM and *rube*™ are closely related inasmuch as they have nearly identical goals. They both make use of the listed dynamic model types within a multimodeling paradigm. The characteristics of *rube*™ may be viewed as the extension of the characteristics of OOPM into the third dimension. The extension includes an implementation of a web-based toolkit written using VRML. A few significant differences between *rube*™ and OOPM are that 1) *rube*™ is implemented in a 3D environment, while OOPM is implemented in a 2D environment, 2) *rube*™ implementation technology is still in its formative stages (there is no full-fledged GUI for *rube*™ as yet), and 3) the use of metaphor as a design aid (and otherwise) will play a much larger role in the development of *rube*™ than it played in the development of OOPM.

What is the utility of visual modeling and simulation, and what is the connection between it and programming? Visual modeling and simulation in 2D is fairly standard in

most simulation domains, especially for manufacturing and digital circuit design. On the issue of the relation of programming to modeling, there has long been a convergence of program to model [13], with the Unified Modeling Language (UML) taking center stage as a frequently-employed approach to software design. Thus, programs can be viewed as models, and so visual programs can be viewed as visual models.

III. EXAMPLE

Here, we demonstrate the practical application of the current research through the presentation of an example program. The program represents a greatly simplified operating system task scheduler that is synthesized utilizing metaphors and 3D visualization. Several issues that arise during the programming effort are discussed in Part E. Based on this example it is found that, though not without difficulty, this approach to programming is viable and worthy of further research.

A. A First Attempt: A Simplified Operating System Task Scheduler/Queuing System

Since we make the claim that 3D metaphor can be used as a high-level design aid, we set out to choose for our example a complex program that should be familiar to our audience. Programs meeting these criteria should best demonstrate the utility of 3D metaphor as a design aid. Though there are many choices that meet the familiar and complex criteria, we focus here on the operating system as a generic type.

In our example, we envision a MINIX-like OS, which can be characterized by hierarchical, vertical layers (top to bottom): 1) user processes (e.g., word processor), 2) server processes (e.g., file system), 3) I/O tasks (e.g., disk), and 4) process management (e.g., scheduling and communication). A graphical representation of these layers can be seen in Fig. 4. In our example, we focus on the process management portion of the OS, and specifically upon a hypothetical task scheduler in that layer. In an abstract sense, the task scheduler may be viewed as a complex queuing system, or QNET.

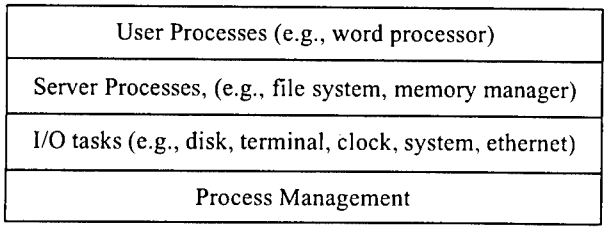


Fig. 4. Graphical representation of the vertically layered structure of a MINIX-like OS.

B. Metaphor Approach and Mappings

The sort of vertically layered structure of the OS described above lends itself naturally to an architectural metaphor, where the OS itself may be visualized as a building, as pictured in Fig. 3. The layers of the OS then correspond to the different floors of the building, as shown in Fig. 6. This is the metaphor we choose. The value of thinking of agents living in the building is at least as great as the value of using the 2D diagram that expresses the OS level hierarchy (Fig. 4), with the added benefit that design ideas may be suggested by the architectural metaphor (e.g., there could be doors between level to provide security). Besides floors, a couple of things that can typically be found inside buildings are people and furniture.

Keeping these things in mind, we now proceed to model the operations of the layers of the OS. The primary interactions between the layers of our hypothetical OS occur through message passing. Although we do not immediately focus on a choice of metaphor for these interactions, we choose a prospective metaphor and set it aside temporarily in order to concentrate on the construction of a single layer. The metaphor we choose for message passing is that of a person transporting a briefcase that contains the message from one place to another, as depicted in Fig. 7. Note that the briefcase metaphor for message passing follows naturally from the synthetic human agent metaphor and does not overlap or conflict with the architectural metaphor.

First, we decide to model the task scheduler portion of the process management layer. Our example task scheduler is a non-preemptive, dynamic priority scheduling system that contains tasks, four priority queues, and the following five types of physical devices with their associated queues: 1) CPU, 2) DEV (i.e., external device), 3) COM (i.e., communication, such as via the parallel, serial, and USB ports), 4) MEM (i.e., memory load/store), and 5) I/O (i.e., input/output, such as disk read/write). Expanding our previous metaphors, we map a task to a *person* (see Fig. 8) and a device to a *service facility* which is a person behind a desk (see Fig. 9). The priority and device queues map directly to physical space as *waiting lines* (see Fig. 10). Persons can travel over paths between the devices and queues. Taking stock of our progression so far, overall we have at least a minimal way to represent the significant portions of our hypothetical task scheduler.

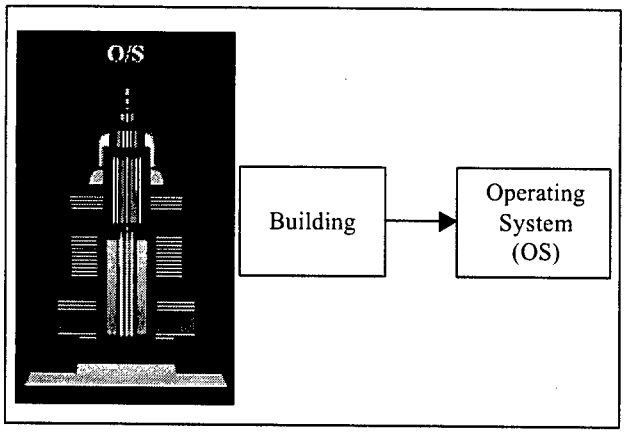


Fig. 5. Metaphor mapping of the building to the OS.

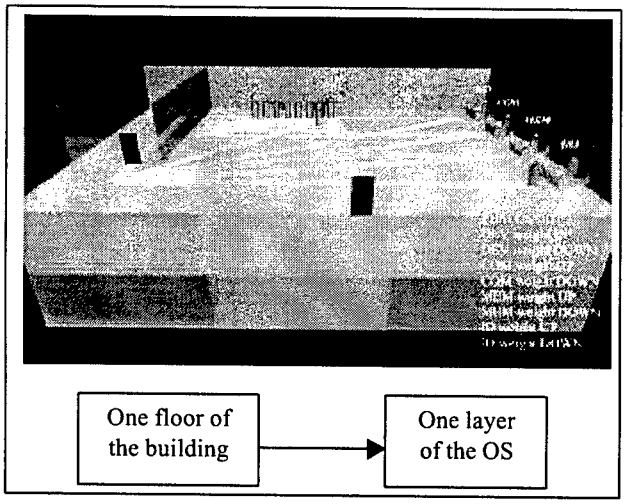


Fig. 6. Metaphor mapping of the floors of the building to the layers of the OS.

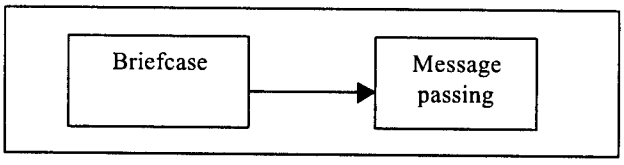


Fig. 7. Metaphor mapping of briefcase to message passing. Picture of mapping not shown because this aspect of the OS has not yet been implemented.

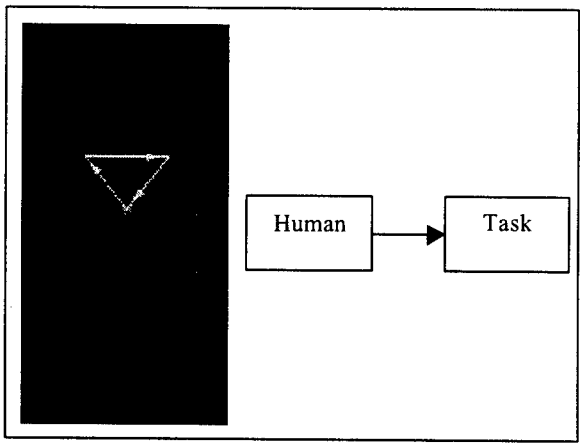


Fig. 8. Metaphor mapping of a human to a task.

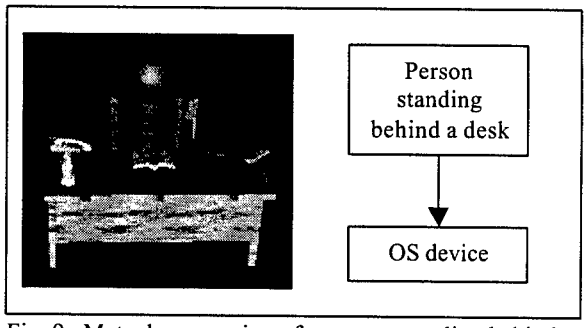


Fig. 9. Metaphor mapping of a person standing behind a desk (service facility) to an OS device.

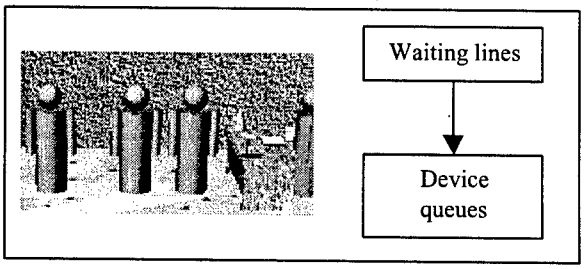


Fig. 10. Metaphor mapping of a waiting line to a device queue.

C. *Implementation*

We are now concerned with the details of implementing the above task scheduler along with its metaphors in a 3D environment. In the context of VRML, we create PROTOs for each of the items in our task scheduler that will be used again and again. A VRML convention for naming PROTOs is to fully capitalize the PROTO name, e.g. MYPROTO. We create the following PROTOs: 1) PERSON, 2) CPUFACILITY, 3) FACILITY, 4) QUEUEMANAGER, and 5) PATH. PERSON is described below. The CPU has special management functions, so it is given its own PROTO. All devices besides the CPU will be instances of the generic FACILITY. When a PERSON reaches a

FACILITY, they are held there for some random amount of time and then released. Instances of QUEUEMANAGER will manage each of the priority queues and the queues leading to each of the service facilities. PATHs connect the devices together physically and manage the movement of persons throughout the world.

Each of the PROTOs mentioned has a physical geometry node included inside that will give it an identifiable and appropriate physical presence in the 3D environment. Some of these physical geometry nodes are modeled in a 3D authoring tool prior to their inclusion in the PROTOs. For example, instances of PERSON look like a stylized version of an actual person, the geometry of which is modeled in *CosmoWorlds*. These appearances are according to the metaphors we are using.

Some items of note within the PROTOs are the attribute fields they include in their declarations. For example, the PERSON PROTO shown below has the fields 1) task type (one of CPU, DEV, COM, MEM, or I/O), 2) CPU run time left (only relevant if this is a CPU task), and 3) task priority (from 1 through 4). The implication here is that each instance of PERSON in our task scheduler carries around its own state information, which can be accessed and modified by other components of the scheduler. Other instances of PROTOs have similar ways of utilizing attributes. Two example PROTO specifications used in the present research (those for PERSON and FACILITY) are given below:

```
PROTO PERSON
  [
  exposedField SFString task_type
  exposedField SFInt32 task_cpu_run_time
  exposedField SFInt32  task_priority
  eventIn      SFBool   running
  eventIn      SFBool   transit
  eventIn      SFBool   waiting
  ]

PROTO FACILITY
  [
  field SFString facility_name "NoName"
  field SFNode   facility_geometry NULL
  field SFInt32  min_random_service_time
  field SFInt32  max_random_service_time
  field    SFFloat  sound_intensity 1
  eventIn  SFTime   time_step
  eventIn  SFNode   task_arrival
  eventIn  SFTime   start_music
  eventIn  SFBool   input_queue_1_empty
  eventIn  SFBool   output_queue_1_full
  eventOut SFFloat  set_sound
  eventOut SFBool
             get_task_from_input_queue_1
  eventOut SFNode
             send_task_to_output_queue_1
  ]
```

After the objects of the task scheduler have been physically arranged in the environment, we have a static representation of the example task scheduler. The next job is to make the task scheduler function dynamically and in an animated fashion. This is accomplished in VRML through the occurrence and utilization of events. Events are generated and sent through ROUTEs to their destinations.

Note that PATH is a metaphor in the following respect: it has a parallel function as that of a ROUTE. PATH physically transports *movers* through the world in a like manner as ROUTE transports events through the world. Fig. 11 shows an example mapping of a ROUTE structure to a use of PATH for transporting a task from a queue to a service facility. PATHs are given physical geometry that can be turned on or off by the user.

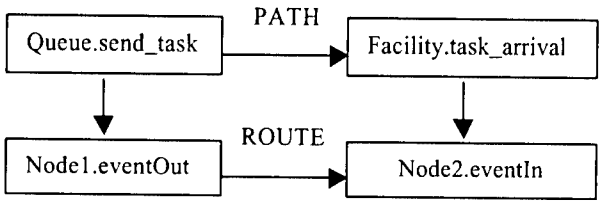


Fig. 11. Metaphor mapping of PATH to ROUTE.

Two more PROTOs are created to aid the implementation: 1) MFACTORY and 2) RANDOMIZER. MFACTORY generates people (tasks) dynamically, and RANDOMIZER randomly sets the fields of the persons to specific task types and assigns them a CPU run time if necessary. The MFACTORY s and RANDOMIZER s logical and functional location in the 3D world is inside a *void* area where tasks are created and their attribute fields are set. However, these two PROTOs have little to do with our initial metaphors and merely serve as *hooks* for the future addition of other components. As such, it is not necessary for them to have a physical presence in the world, and therefore they have no geometry associated with them. If desired, geometry could be added to them later.

Next, we physically arrange the components of the OS task scheduler in the VRML world. An overhead schematic of the task scheduler layout is given in Fig. 12.

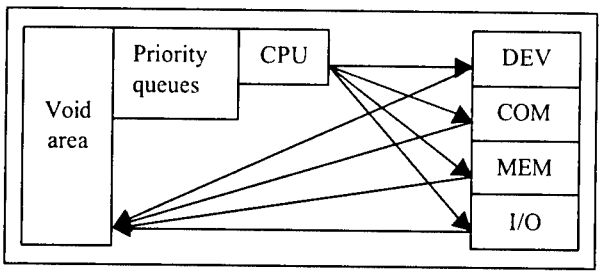


Fig. 12. Overhead schematic of task scheduler component layout. Arrows represent paths and queues for tasks.

Lastly, a finite state machine (FSM) is placed inside each person that reflects the associated task s current state: 1) *Waiting*, 2) *Running*, and 3) *In Transit*. These states roughly correspond to the familiar *waiting/running/ready* FSM for processes in an OS task scheduler, respectively. The user is able to zoom in on any particular person and see through the person s transparent skin into the person s chest, where the FSM is shown progressing through its various states, as pictured in Fig. 8. In addition to the FSM changes that can be viewed, the color of a person s skin changes to a color assigned to each state using a *traffic light metaphor*: 1) *Waiting* – red, 2) *Running* – green, and 3) *In Transit* – yellow. The addition of the internal FSM and the traffic light metaphor help demonstrate both the multimodeling and multimetaphor aspects of *rube*™.

D. Operation

When the program is initialized, `MFACTORY` generates 10 tasks inside the void area. Person/tasks have internal accounting abilities – they are individuals, and maintain their own task type, remaining run time, and priority information, which are utilized especially by the `RANDOMIZER` and `CPUFACILITY`. The 10 tasks move along a path to the `RANDOMIZER`, where the tasks fields are set. The randomizer begins with a 50/50 probability of assigning a task as a CPU task vs. another device task. Later, the user can alter these probabilities using a heads-up display (HUD). The initial task probability weightings are 1) CPU – 20, 2) DEV – 5, 3) COM – 5, 4) MEM – 5, and 5) I/O – 5. No control over CPU weight is given to the user from the HUD since changing the weights of the devices uniformly is equivalent to changing the weight of CPU tasks. If a task is a CPU task, it is also randomly assigned a CPU service time that the CPU will need to dispatch that task. Fields in the `RANDOMIZER` declaration, however, determine the randomization of CPU task service time, so the user has no control over it from the HUD. If a task is something other than a CPU task, it is assigned a service time of zero (that is, the CPU will send it on its way to the appropriate device immediately upon arrival at the CPU).

All tasks start out with a priority of (1) upon leaving the `RANDOMIZER` in the void area. When a task leaves the `RANDOMIZER`, it enters the first priority queue leading to the CPU. All queues, including the CPU priority queues, are managed by instances of the `QUEUEMANAGER PROTO`, which send out waiting tasks upon request from the CPU and devices.

When a task reaches the CPU, if the task *is* a CPU task, it gets service from the CPU. A backbeat soundtrack plays as the CPU is serving a task. The CPU determines how much time to spend servicing the task as follows: Priority 1 tasks are given 10 seconds, which is the priority 1 time quantum. If this quantum is expended before the total service time for the task is complete, the CPU assigns the task a priority of 2, updates the remaining total service time field inside that task, and sends it to the second priority queue. The other priority queues work the same way, except their CPU time quanta are 20, 30, and 40 for priorities 2, 3, and 4 respectively. Time quanta for each priority are determined by the `CPUFACILITY` declaration. If a task reaches priority 4, goes to the CPU, expends the 40 second time quantum, and is still not finished executing, the CPU leaves the task at priority 4 and sends it back to the fourth priority queue. When a CPU task's total service time is expended, it is sent back to the void area. The CPU determines which queue to draw tasks from based on the following rule: Draw a task from the highest priority queue that has a *waiting* task (by *waiting*, the task is meant to be at the head of the queue, and not traveling/in transit from the end of the queue to the head). Lower priority queues may starve, but this is a danger of any priority queuing system that is not implemented in a round-robin fashion. A round-robin system would be simple to implement here, if so desired.

When a task reaches the CPU, if the task *is not* a CPU task, it is sent along to the appropriate peripheral device queue. There is no priority queuing at the peripheral devices – these queues operate on a first-in-first-out basis. When a non-CPU task arrives at its device, the device decides on a random service time for that task. Fields in the `FACILITY` declarations determine the randomization of service time at peripheral devices. While a peripheral device is serving a task, a specific sound can be heard: 1) DEV – church bell, 2) COM – blowing wind, 3) MEM – running water, and 4) I/O – rain and thunder. When the task s service time is expended, the task is sent back to the void area.

In summary, proceeding in a top-down fashion, we first encounter the shell of the building of the OS as pictured in Fig. 3. Taking off the shell, we see a series of floors stacked one on the other as shown in Fig. 13. On one of these floors we see the task scheduler operating as pictured in Fig. 13 and Fig. 14. Inside the task scheduler, we see persons moving along paths and through queues toward service facilities, where they spend some random amount of time receiving service as shown in Fig. 15. As persons go through the *Waiting/Running/In Transit* states, their skin color changes and their internal FSM changes state as shown in Fig. 16. Service itself is associated with a unique type of sound that continues as long as that type of service is being provided. When their service at some facility is complete, persons leave the service facilities, enter a void area, and have their attributes reset to start the process again. This "recycling" of persons avoids destroying and regenerating them in order to continue the simulation. Lastly, a HUD is provided, which allows the user to turn the geometric representations of the paths on and off and alter the random probabilities of task type assignment, all while the simulation is operating.

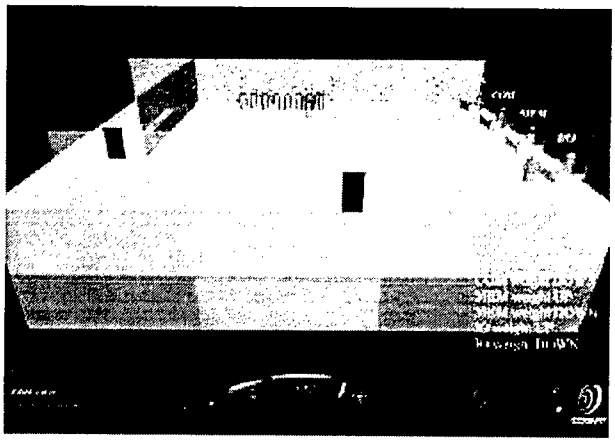


Fig. 13. The floors of the OS building with the task scheduler shown on the top floor.

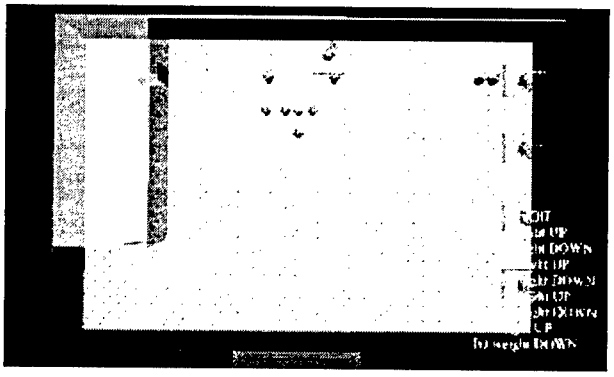


Fig. 14. Top view of the task scheduler. The CPU facility is shown at top center, with its priority queues just below and to the left of it. DEV, COM, MEM, and I/O service facilities are on shown on the right, top to bottom respectively, with queues to the left of each.

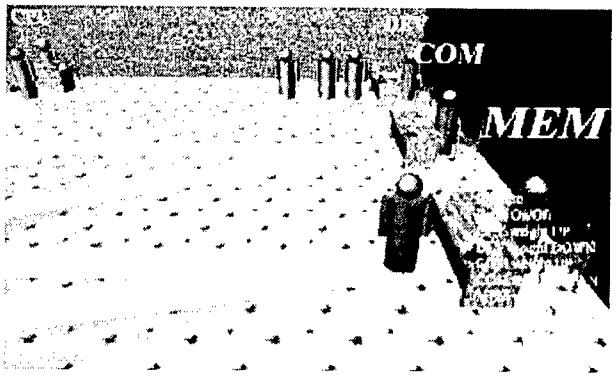


Fig. 15. Close-up view of task scheduler operation.

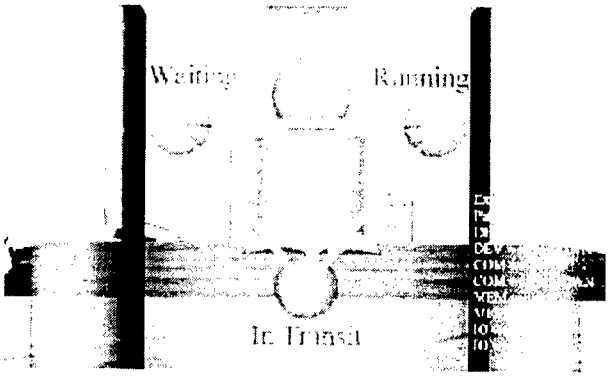


Fig. 16. A task s internal FSM with the In Transit, Waiting, and Running states. The task shown is being serviced at the CPU.

The actual implementation of the above-described 3D world/program is located at the following URL at time of writing:
http://www.cise.ufl.edu/~fishwick/rube/worlds/os.html

E. Issues

1) Modeling, Metaphor, and the "Real World"

The choice of the architectural metaphor for the OS may guide the rest of the design process, but it is not the only possible metaphor that can be chosen. This choice is left to the programmer, which coincides with the suggestion in [2] that the use of metaphor is most effective when the programmer has the freedom to produce his or her own metaphors. Neither is it necessary that the architectural metaphor exclusively dominate the rest of the design process. There is great freedom in subsequently choosing peer or sub-metaphors that may or may not have some relation to the architectural metaphor. These choices may lead to a *multi-metaphor* construction. The importance of this concept is that one need not necessarily come up with one over-arching metaphor that can include every possible circumstance that will need to be modeled in advance. Note here that we do not return to the person and briefcase message passing metaphor, although we must do so eventually. The point is that we do not need to expand on it immediately.

Note that we attempt to physically surface as many relevant attributes of the metaphor and object we are modeling as possible. This is key to the use of 3D metaphor. If it cannot be seen visually, the utility of the metaphor may be greatly reduced.

We do not claim that metaphor should be applied in all situations. It is unlikely that metaphor may be easily applied to every object or process that can be modeled. It is also doubtful that all modeling or programming tasks have a metaphor that is a "perfect fit." When a programmer must

deal with a program s atomic operations, it is possible that there may be no appropriate metaphor, or the use of metaphor may not contribute to the programmer s understanding of the situation. This barrier in the use of metaphor will remain until such time as programmers no longer need to deal with the details of traditional programming. Rather, we claim that the use of 3D metaphor can be of some aid and may simplify the programming task in many instances.

2) Implementation and Technical Issues

VRML is not a full-blown object-oriented language. VRML provides encapsulation of fields, and uses ROUTEs to provide circuit-like access to fields. However, this sort of implementation does not yield traditional or complete encapsulation. Additionally, there is no inheritance in VRML in an object-oriented sense. These factors and others make it difficult to achieve object-oriented design goals within VRML. PROTOs, ROUTEs, and scripts alone are not enough to satisfy object-oriented design goals, because there are few restrictions on their structure and function.

The standards for VRML are widely and differently interpreted, resulting in different and sometimes unpredictable behavior between browsers. This is especially evident for event ordering and handling. Overall, it becomes difficult to produce consistent behavior between browsers, and programmers of VRML may find themselves inconvenienced by the necessity of adapting their code depending on the type of browser being used.

There is no good 3D editing tool that allows one to "program" VRML using 3D graphics, although tools are available that export VRML and some VRML-specific geometry tools do exist. Ideally, we would like to be able to write a PROTO, place it in a *toolbox*, and then later gain drag-and-drop access to that PROTO through a 3D editing tool. Then we would like to connect ROUTEs like *pipes* to and from our different objects and PROTOs in the 3D editor. We did not use such an editor because none exists yet – it remains to be developed. However, the basic ideas described in this research will be easily and intrinsically applicable in making use of such an editor when one is finally developed.

One has to separate the interface used to create the 3D program from the structure and representation of the 3D program. Creating graphical user interfaces is highly time intensive, and our fairly small research group has chosen to focus on the representation and visualization issues, while using off-the-shelf 3D tools to aid in the creation of the geometry. Our creation of VRML need not be text-based. We use and promote tools such as *CosmoWorlds* and *Internet Scene Builder* (Parallel Graphics). However, these tools do not support the ability to assign one piece of geometry as a dynamic model of another, and so we have to manually encode these semantics using the VRML prototyping mechanism, which is poorly supported by the available VRML world generation tools. Moreover, the actual semantics must be coded in JavaScript, as they are not automatically generated from the 3D model objects. This is similar to the general state of the art in 3D game engines: the semantics must be separately applied using a scripting language even though the levels can be interactively created. This represents an important first step toward an eventual, complete 3D programming environment that addresses both the representational as well as the HCI issue of immersive and interactive "3D code development."

Lastly, VRML is an interpreted language. As complexity of a given world increases, there can be a noticeable drain on computer resources, even with very fast PCs. Also, there are no good debugging tools for VRML, as browser error messages can be cryptic. This has the effect of lengthening the design-debug cycle.

Thus, we do not support the idea that VRML should be used to implement a functional operating system or any other production system in which performance or programmer productivity is an issue. Further, VRML is not the only possible vehicle for demonstrating the *rube*™ methodology – the choice of language is incidental to the methodology. VRML was chosen for its adequacy in the task at hand and its practical expediency in producing relatively simple visual simulations, rather than for any great technical merit it may (or may not) possess as a programming language.

While the OS example may be interesting, the efficiency and expeditiousness of it for designing and implementing an actual, practical OS task scheduler is beyond discussion. This is because the design task (that of designing an OS and its task scheduler), minus the 3D metaphors, had already been largely completed prior to the beginning of the project by virtue of the fact that we had chosen MINIX as a rough model. Overall, we showed that we could *represent* an OS task scheduler using the *rube*™ paradigm. It is reasonable to assume that, during the representation process, we created the tools necessary to be able to *generate* an OS task scheduler design. However, the design and generation of a novel program by a novice would better demonstrate the utility of 3D metaphor in the design task. As far as the actual implementation of an OS task scheduler is concerned, the visualization itself would make the scheduler far too slow to perform the scheduling task efficiently. Eventually, the visualization would probably be stripped away to boost efficiency, perhaps only to resurface again during debugging efforts.

3) Aesthetic Issues

The current research brings out the issue of aesthetics in programming in a novel way [13]. At the moment, programmers may discuss the aesthetics of programs that exist as text code. These discussions may include the subjects of code commenting, brace indexing, overall

design efficiency, and etc. Sometimes, aesthetics may not be considered at all. With the introduction of 3D metaphor, aesthetics play a larger role, and the potential develops for programs to evolve into a type of visual art.

The increased use of metaphor in *rube*™ is predicated on the idea that aesthetics will continue to play a more important role in modeling and programming. The history of book-making lends us an analogy that strengthens this claim. The first books were expensive and made for a small readership. The goal was to create a book for one individual or a small group. When Gutenberg created the movable type press, books could be mass-produced for a much wider readership. It was no longer necessary to hand-create books and books became less costly to produce. The shift was away from a *one to one* relationship of producer to consumer to a *one to many* relationship. Now, with our increased technology, we are able to return to *one to one*, albeit with a much more efficient delivery mechanism. Media is returning to a more focussed transmission, away from broadcasting. The idea is to benefit the individual directly, without creating generally palatable media. The renewed focus on the individual promotes aesthetics and personal preference in modeling.

In the present example, some effort is made to have the program interact with and entertain the user. This is done with view changes, animation, and sound. These aspects of the research should be attractive to novices. It should be noted that the programmers are not artists, but rather use some creativity that might not otherwise have been called for. For programmers that would rather not consider this issue, it may be ignored, or it may prove to be a hindrance to using the method altogether. Others may view it as an opportunity to explore and pioneer a new art form.

4) Psychological Issues

The utilization of 3D metaphor necessitates that the programmer thinks in an object-oriented fashion. When the program and its operation will ultimately be represented in three dimensions with concrete and familiar objects, the programmer has little choice but to consider this during the programming process. Still, the programmer must make a mental effort to keep this in mind, and there is some extra work involved in invoking 3D metaphors. Additionally, a programmer must now be concerned with problems associated with visualization. Finally, since we are extending the concept of object-oriented design, a very slight paradigm shift is involved.

With the freedom to choose any type of metaphor, including a combination of metaphors to represent a program, we also introduce the possibility that there can be "too much freedom." In other words, great freedom could result in poor structure, or vacillation over what is "the best metaphor." We rely on the programmer to be disciplined enough to adhere to good design principles and be creative at the same time.

Ultimately, the utility of 3D metaphor will need to be judged from the results of empirical research. Such research might concern itself with both tangible and psychological factors. These may include such things as gains or losses in efficiency and correctness compared with traditional methods, length of design and debug cycles, programmer memory retention of program structures and design after varying periods of time, and ease and enjoyability of use. Historically, this kind of research has been hindered by the use of tools characterized by poor usability; poor usability interferes with the evaluation of more important aspects of the research. Hopefully, better tools are on the horizon. Two examples of related psychological studies are given in [3] and [33]. See [40] for an interesting empirical psychological study of users assessments of aesthetics and apparent usability in interface design.

5) Our Research Philosophy

We have not performed human experiments to justify our use of 3D, instead relying on the definition and deployment of a complete methodology. This approach is consistent with the usual dichotomies present that separate designer-builders from those performing statistical validation experiments. We recognize the importance of such experiments; however, due to the relatively small size of our research group, we lack the personnel resources necessary to perform these experiments ourselves. To mitigate this seeming deficiency in our method, we are in close contact with several researchers working in HCI and the psychology of user interfaces [40].

Currently, our mission is to express the *rube*™ paradigm and justify it by demonstrating how it achieves specific goals that are novel and unique, rather than immediately justify it through HCI-type research. This is because our research focus is more on modeling than on human-focused experimental research. We do not diminish the importance of such research (rather the opposite), but we do attempt to maintain our modeling focus in the face of the potentially significant distraction of HCI research. This perspective ensures that we progress in the area of model design and methodology. However, one of our goals is to monitor and make use of past and future HCI research and thus achieve a symbiosis.

IV. CONCLUSIONS

Through the creation of an example operating system task scheduler program using VRML, we have explored the utility and viability of the multimodeling and metaphor aspects of the *rube*™ paradigm. This approach involved modeling components of a simplified OS task scheduler as a QNET, and incorporated FSMs inside the tasks. Further, we used appropriate metaphors and their mappings to real-world objects in the design and visualization of the system. These metaphors included the mapping of a synthetic

human to an OS task, where the synthetic human may be viewed as an agent of the OS task as the task s representative to the task scheduler. We showed that the synthetic human agent metaphor may have application in agent systems, and may prove useful for novice modelers in particular. The issues, advantages, and disadvantages of the approach were discussed. We found that the method was both useful and viable for the example program and that the general method was worthy of further research and extension.

ACKNOWLEDGEMENTS

We would like to thank the following funding sources that have contributed towards our study of modeling and simulation: (1) Jet Propulsion Laboratory under contract 961427 *An Assessment and Design Recommendation for Object-Oriented Physical Modeling at JPL* (John Peterson and Bill McLaughlin); (2) Rome Laboratory, Griffiss Air Force Base under contract F30602-98-C-0269 *A Web-Based Model Repository for Reusing and Sharing Physical Object Components* (Al Sisti and Steve Farr); and (3) Department of the Interior under grant 14-45-0009-1544-154 *Modeling Approaches & Empirical Studies Supporting ATLSS for the Everglades* (Don DeAngelis and Ronnie Best). We are grateful for their continued financial support. We would also like to thank Milagros Lemos of *ReyesWorks* for allowing us to cite *ReyesWork* s agent development efforts and for granting us permission to reprint the photo in Fig. 1.

REFERENCES

[1] M. Auguston, "The V experimental visual programming language," Tech. Rep. NMSU-CSTR-9611, New Mexico State Univ., October 1996.

[2] A. F. Blackwell, *Metaphor in Diagrams.* Ph.D. dissertation, Darwin College, Univ. of Cambridge, Cambridge, U.K., September 1998.

[3] A. F. Blackwell and T. R. G. Green, "Does Metaphor Increase Visual Language Usability?" in *Proc. 1999 IEEE Symp. on Visual Languages*, pp. 246 – 253, 1999.

[4] R. Burkhart, "The Swarm Multi-Agent Simulation System," in *(OOPSLA) '94 Workshop on "The Object Engine,"* September 1994.

[5] S. K. Card, J. Mackinlay, and B. Shneiderman, Eds. *Readings in Information Visualization : Using Vision to Think*. San Francisco, CA: Morgan Kaufmann, 1999.

[6] R. Carey and G. Bell, *The Annotated VRML 2.0 Reference Manual.* Reading, MA: Addison-Wesley, 1997.

[7] P. T. Cox and T. Pietryzkowsky, "Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism," in E. P. Glinert, Ed. *Visual Programming Environments: Paradigms and Systems.* Los Alamitos, CA: IEEE Computer Society Press, 1990.

[8] R. M. Cubert, T. Goktekin, and P. A. Fishwick, "MOOSE: architecture of an object-oriented multimodeling simulation system," in *Proc. Enabling Technology for Sim. Sci., 1997 SPIE AeroSense Conf.*, pp. 78 – 88, April 1997.

[9] A. A. di Sessa, "Notes on the future of programming: breaking the utility barrier," in D.A. Norman and S.W. Draper, Eds. *User Centered System Design: New Perspectives on Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum, 1986.

[10] W. Finzer and L. Gould, "Programming by Rehearsal," in *BYTE*, vol. 9, no. 6, pp. 187-210, June 1984.

[11] P. A. Fishwick, "SIMPACK: Getting Started with Simulation Programming in C and C++," in *1992 Winter Sim. Conf. Proc.*, pp. 154 – 162, 1992.

[12] P. A. Fishwick, *Simulation Model Design and Execution*. Englewood Cliffs, NJ: Prentice Hall, 1995.

[13] P. A. Fishwick, "Aesthetic Programming," *Leonardo*, MIT Press, submitted for review September 2000.

[14] P. A. Fishwick, N. H. Narayanan, J. Sticklen, and A. Bonarini, "A Multi-Model Approach to Reasoning and Simulation," in *IEEE Trans. on Syst., Man and Cybern.*, vol. 24, no. 10, pp. 1433 – 1449, 1992.

[15] P. A. Fishwick, A. Urmacher, and B. Zeigler, Eds., "Agent Oriented Approaches in Distributed Modeling and Simulation: Challenges and Methodologies," in *Dagstuhl Seminar Report*, July 1999.

[16] P. A. Fishwick and B. P. Zeigler, "A Multimodel Methodology for Qualitative Model Engineering," in *ACM Trans. on Modeling and Comp. Sim.*, vol. 2, no. 1, pp. 52 – 81, 1992.

[17] S. Franklin and A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents," in *Proc. of the Third Int. Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1996.

[18] J. Hopkins and P. A. Fishwick, "On the Use of 3D Metaphor in Programming," in *Proc. of Enabling Technology for Sim. Sci., SPIE AeroSense 2000 Conf*, April 2000.

[19] W. Jansen and T. Karygiannis, "NIST Special Publication 800-19 – Mobile Agent Security," Gaithersburg, MD: National Institute of Standards and Technology, August 1999.

[20] N. R. Jennings and M. J. Wooldridge, Eds., *Agent Technology: Foundations, Applications, and Markets*. Berlin: Springer-Verlag, 1998.

[21] C. G. Jung, J. Lind, C. Gerber, M. Schillo, P. Funk, and A. Burt, "An architecture for co-habited virtual worlds," in *Proc. of the 1999 Virtual Worlds and Sim. Conf.*, January 1999.

[22] K. Kahn, "Generalizing by Removing Detail," in *Comm. of the ACM*, vol. 43, no. 3, pp. 104 – 106, March 2000.

[23] H. Lieberman, "Visual Programming: A Vision for the Future," in *Friend-21 Conf. on Human Interface Technologies*, September 1989.

[24] H. Lieberman, "A Three-Dimensional Representation for Program Execution," in *Proc. of the 1989 Workshop on Visual Languages*, October 1989.

[25] H. Lieberman, "Programming by Example," in *Comm. of the ACM*, vol. 43, no. 3, pp. 73 – 74, March 2000.

[26] A. B. Loyall and J. Bates, "Real-time Control of Animated Broad Agents," in *Proc. of the Fifteenth Annual Conf. of the Cognitive Sci. Society*, June 1993.

[27] C. Marrin and B. Campbell, *Teach Yourself VRML 2 in 21 Days*. Indianapolis, IN: Sams.net Publishing, 1997.

[28] M. Mateas, "An Oz-Centric Review of Interactive Drama and Believable Agents," Tech. Rep. CMU-CS-97-156, Carnegie Mellon Univ., June 1997

[29] B. Meyer, *Object-Oriented Software Construction Second Edition*. Upper Saddle River, NJ: Prentice Hall, 1997.

[30] B. A. Myers, R. McDaniel, and D. Wolber, "Intelligence in Demonstrational Interfaces," in *Comm. of the ACM*, vol. 43, no. 3, pp. 82 – 89, March 2000.

[31] M. Najork, *Programming in Three Dimensions.* Ph.D. dissertation, Univ. of Illinois at Urbana-Champaign, 1994.

[32] T. Nelson, "The Right Way to Think about Software Design," in B. Laurel, Ed. *The Art of Human-Computer Interface Design*, Addison-Wesley, 1990

[33] M. Petre and A. F. Blackwell, "Mental imagery in program design and visual programming," in *Int. Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 7 – 30, 1999.

[34] A. Repenning and C. Perrone, "Programming by Analogous Examples," in *Comm. of the ACM*, vol. 43, no. 3, pp. 90 – 97, March 2000.

[35] K. Schelderup and J. Ölnes, "Mobile Agent Security – Issues and Directions," *Lecture Notes in Comp. Sci.*, vol. 1597, pp. 155-167, 1999.

[36] D. C. Smith, *PYGMALION: A Creative Programming Environment.* Ph.D. dissertation, Stanford Univ., 1975.

[37] D. C. Smith, A. Cypher, and L. Tesler, "Novice Programming Comes of Age," in *Comm. of the ACM*, vol. 43, no. 3, pp. 75 – 81, March 2000.

[38] R. St. Amant, H. Lieberman, R. Potter, and L. Zettlemoyer, "Visual Generalization in Programming by Example," in *Comm. of the ACM*, vol. 43, no. 3, pp. 107 – 114, March 2000.

[39] J. Stasko, J. Domingue, M. Brown, and B. A. Price, Eds. *Software Visualization: Programming as a Multimedia Experience.* MIT Press, 1998.

[40] N. Tractinsky, "Aesthetics and Apparent Usability: Empirically Assessing Cultural and Methodological Issues," in *Proc. 1997 Conf. on Human Factors and Computing Sys.(CHI 97)*, pp. 115-122, March 1997.

[41] T. Veale, *Metaphor, Memory and Meaning: Symbolic and Connectionist Issues in Metaphor Interpretation*. Ph.D. dissertation, School of Computer Applications, Dublin City Univ., Dublin, Ireland.

[42] P. Young, *Visualising Software in Cyberspace*. Ph.D. dissertation, Dept. of Computer Science, Univ. of Durham, Durham U.K., October 1999.

John F. Hopkins was born in Elizabeth, NJ on November 20, 1969. He enlisted in the US Navy in 1987 and performed the duties of electronics technician and nuclear reactor operator on board a Los Angeles class submarine. He received the Bachelor of Science degree in Psychology from the University of Florida, Gainesville, FL, in 1998. He is currently pursuing the Master of Computer and Information Science degree at the University of Florida and expects to graduate in 2001.

Paul A. Fishwick is Professor of Computer and Information Science and Engineering at the University of Florida. He received the PhD in Computer and Information Science from the University of Pennsylvania. He also has six years of industrial/government production and research experience working at Newport News Shipbuilding and Dry Dock Co. (doing CAD/CAM parts definition research) and at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation modeling and analysis methods for complex systems. He is a senior member of the IEEE and a Fellow of the Society for Computer Simulation. Dr. Fishwick founded the comp.simulation Internet news group (Simulation Digest) in 1987, which now serves over 20,000 subscribers. He has chaired workshops and conferences in the area of computer simulation, and will serve as General Chair of the 2000 Winter Simulation Conference. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990). Dr. Fishwick's WWW home page is **http://www.cise.ufl.edu/~fishwick** and his E-mail address is **fishwick@cise.ufl.edu**. He has published over 40 journal articles, written one textbook, co-edited two Springer Verlag volumes in simulation, and published six book chapters.

Web-Based Simulation: Revolution or Evolution?

Ernest H. Page
The MITRE Corporation
1820 Dolley Madison Blvd.
McLean, VA 22102

Arnold Buss
Operations Research Department
Naval Postgraduate School
Monterey, CA

Paul A. Fishwick
Department of Computer
and Information Science Engineering
University of Florida
Gainesville, FL 32611

Kevin Healy
ThreadTec, Inc.
P.O. Box 7
Chesterfield, MO 63017

Richard E. Nance
Systems Research Center and
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

Ray J. Paul
Centre for Applied Simulation Modeling
Brunel University
Uxbridge, Middlesex, UB8 3PH, UK

Abstract

The nature of the emerging field of web-based simulation is examined in terms of its relationship to the fundamental aspects of simulation research and practice. The presentation, assuming a form of debate, is based on a panel session held at the first International Conference on Web-Based Modeling and Simulation which was sponsored by the Society for Computer Simulation during 11-14 January 1998 in San Diego, California. While no clear "winner" is evident in this debate, the issues raised here certainly merit ongoing attention and contemplation.

Categories and Subject Descriptors: I.6.5 [**Simulation and Modeling**]: Model Development – *modeling methodologies,* I.6.8 [**Simulation and Modeling**]: Types of Simulation – *distributed*

Additional Key Words and Phrases: Digital objects, distributed modeling, Java

1 Preface

The emergence of the world-wide web (WWW) has produced an environment within which many disciplines are being re-evaluated in terms of their inherent approaches, techniques and philosophies. The disciplines concerned with computer simulation are no exception to this phenomenon; the concept of "web-based simulation" has been introduced and is currently the subject of much interest to both simulation researchers and simulation practitioners. As an area of scholarly endeavor, web-based simulation debuted as a 3-paper session at the 1996 Winter Simulation Conference (WSC) and was, by far, the most well-attended session within the modeling methodology track of that conference. This success was repeated at WSC 97, and in January 1998 the first conference dedicated to the topic of web-based simulation was held as part of the annual Society of Computer Simulation (SCS) Western Multiconference [9].

This paper stems from a panel convened for WEBSIM '98. The charter for the panel was to examine the *fundamental* nature of web-based simulation and explore its relationship to the body of theory and practice in simulation modeling methodology that has evolved over the past forty years [14]. One goal for the panel was to distill the essential and differentiating aspects of web-based simulation, if any, from amongst the mountains hype that tend to surround the WWW. The central question was this: does web-based simulation represent a revolutionary change or an evolutionary change? We posed the question because the nature of change would seem to bear some relationship to the proper directions and focus of web-based simulation research and practice.

The panel composition was structured in an effort not only to portray all sides of the issues being addressed but also, hopefully, to engender controversy and stimulate the participation of the audience. Arnold Buss, Paul Fishwick and Kevin Healy are active in web-based simulation research and development. Dick Nance and Ray Paul represent the "traditional" simulation modeling methodology community. Kevin

Healy presents a view from the commercial world, the rest of the panel hails from academe. Arnold, Dick, Paul and Kevin provide a U.S. perspective. Ray serves as international representative.

The remainder of the paper is organized as follows. Section 2 establishes the framework for debate. The panelists responses are captured in Section 3. We attempt to portray the essence of the dialogue that occurred during the session through focus on a few key points of dispute in Section 4. Section 5 contains a concluding summary.

2 What are the Modeling Methodological Impacts of Web-Based Simulation?

Simulation modeling methodology deals with the creation and manipulation of models over the lifetime of their use. Motivated by the recognition that the manner in which a simulation model is conceived, developed and used can have a significant impact on the ability of the model to achieve its objectives, modeling methodology has been an active research area since the inception of digital computer simulation.

Over the past forty years the practice of simulation model creation has evolved from coding in general-purpose languages, to model development in special-purpose simulation languages, to model design using higher-level simulation model specification languages and formalisms, to comprehensive theories of simulation modeling and holistic environment support for the modeling task. Thematic in much of the modeling methodological work to date has been the recognition of Dijkstra's principle of the "separation of concerns" which argues for the separateness of specification and implementation [4]. In many cases, this philosophy has been tempered by the pragmatic observations of Swartout and Balzer [17], who observe that separation is a worthy goal but not achievable in totality since any specification, S, may be viewed as an implementation of some higher-order specification, S'.

Another argument in favor of the intertwining of specification and implementation is that technological advancements may enable new approaches to accomplishing a task—approaches that were not even conceivable prior to the advent of the technology. Consider, for example, the advent of the assembly line in manufacturing. The potential of the WWW as such a technology push is cited in an article that describes the application of the WWW within the manufacturing process [5], "Our initial experiments at putting engineering, design and manufacturing services on the Web are so successful that we believe we should rethink the traditional approaches and tools for coordinating large, distributed teams." With respect to simulation, a similar revolution seems plausible. Web technology has the potential to significantly alter the ways in which simulation models are developed (collaboratively, by composition), documented (dynamically, using multimedia), analyzed (through open, widespread investigation) and executed (using *massive* distribution).

Is the web, in fact, such an elixir, demanding that we radically alter our modeling philosophies and approaches? Or is web-based execution merely another implementation detail that can, and should, be abstracted from the model development process?

3 Responses

The following sections contain responses from the panel members regarding the central question.

3.1 Web-Based Simulation Modeling (Arnold Buss)

The explosion of computer networks have created an environment for computer modeling in general, and simulation modeling in particular, that is revolutionary. In order to properly exploit these developments the nature of modeling must change.

Simulation models have been traditionally monolithic in design. The advent of Object-Oriented Programming has resulted in more elegantly designed monoliths. Simulation models for both industrial and military applications have been mostly designed for models running on a single machine. For such models the network offers little. Using the full power of the network offers potentially substantial benefits to modeling and simulation, but only if models are designed differently.

Use of up-to-date data by dynamically interacting with databases across the network speeds up the modeling and decision-making cycle by an order of magnitude. The integration of computer models running with systems has great potential for military analysis and training.

In nutshell, web-based simulation models must accommodate:

- applications that expect to receive data across the network from a database that will be dynamically determined,
- applications that will expect to receive new classes and data unforeseen at the time the model was started, and
- applications using components that are loosely bound, rather than tightly coupled.

The Java programming language, together with the related cluster of Java Technologies, have substantially extended the capabilities of program-level tasks. Java classes can open sockets across a network, perform database queries, and encrypt data streams for secure transmission. New classes may be dynamically incorporated as the program is running, thus enabling dynamic extensibility. Objects on one computer may be serialized and sent to another, where thay are immediately incorporated into that computer's model. Objects on another computer may be invoked through Remote Method Invocation.

The capabilities of programming languages have outstripped our knowledge of how best to write programs exploiting these capabilities. Software design principles for procedural and even object-oriented programs are well-known. It is not yet known how software should be designed using these tools. It is also not clear how best to exploit the tremendous possibilities offered by the network.

3.2 Distributed Modeling Using the Web as an Infrastructure (Paul A. Fishwick)

One of the most critical problems in the field of computer simulation today is the lack of published models and physical objects within a medium—such as the World Wide Web—allowing such distribution. The web represents the future of information sharing and exchange, and yet it is used primarily for the publication of documents since the web adopts a "document/desktop metaphor" for knowledge. In the near future, we envision an "object metaphor" where a document is one type of object. A web predicated on digital objects is much more flexible and requires a knowledge in how to model physical phenomena at many different scales in space-time.

If a scientist or engineer (i.e., model author) works on a model, places the model inside objects, and constructs a working simulation, this work occurs most often within a vacuum. Consider a scenario involving an internal combustion engine in an automobile, where the engine is the physical object to be simulated. The model author's task is to simulate the engine given that a new engineering method, involving a change in fuel injection for example, is to be tested. By testing the digital engine and fuel injection system using simulation, the author can determine the potential shortfalls and benefits of the new technique. This task is a worthwhile one for simulation, and simulation as a field has demonstrated its utility for objects such as engines.

Let's analyze the problems inherent in this example. There is no particular location that will help the author to create the geometry of the engine and its dynamics. Moreover, if the model author seeks reusable components on the web, who is to ensure the quality or accuracy of these components? It may be that other employees of the company have made similar engine models in the past, and that these models may be partially reused. If this is the case, the model author is fortunate, but even if such a company-internal model exists, it may not be represented in "model form." There may be other model authors who have already constructed pieces that our model author could use, but there if there is no reuse and no standard mechanism for publishing the model or engine object, then this is all for naught. The model author may also be concerned with creating a fast simulation. While algorithms for speeding simulations are important, by solving the reusability problem, we also partially solve the speed problem since published quality models of engines will battle in the marketplace for digital parts, and the best engine models and testing environments–involving very fast and efficient simulation algorithms–will win out in the end. Therefore, the problem of reuse of engine objects and components lies at the heart of the simulationist's dilemma. Fast, efficient

and quality models could be available at some point in the future, but today there is no infrastructure or agreed-upon standards. for true digital object engineering.

What if the model author of the engine creates a digital engine that operates differently than the actual one? The automobile company could provide full access to an invalid model. We must have quality control measures in place to help us with this situation. The physical metaphor provides some help. Many consumer groups and institutions exist to protect consumers from bad products. Digital products will require similar groups and testing procedures. If a company knowingly markets a bad digital product, they will ultimately pay for this error in the marketplace. The digital object must be treated with the same level of quality control as the physical counterpart. In some cases, a company might make a mistake in production and a part or entire vehicle must be recalled. This type of recall is made easier with the digital product. It behooves the model authors to create valid, quality objects. It may be that anyone can publish a digital object but this is true of physical objects as well. The situation is somewhat more acute with a digital automobile since to create an automobile in the first place, one must have invested a fair amount of time and resources; however, a digital engine could be created by the neighbor down the street. One must learn to trust certain sources more than others based on past performance of prior digital objects. Also, we must have ways of verifying our sources, developers and producers with methods such as digital signatures, watermarks and encryption.

3.3 Simulation Modeling Methodology and the WWW (Kevin J. Healy)

The World Wide Web was conceived as a set of simple Internet-based client/server protocols for transferring and rendering documents of a primarily textual nature. What distinguished the Web's mode of communicating information from other Internet-based tools that preceded it (e.g. electronic mail, electronic file transfer via ftp, and network newsgroups) was the provision for embedding hyperlinks that allowed users to easily navigate between related documents. The hyperlinking scheme allowed content providers to organize and present information in a natural hierarchical fashion. It also served to insulate users from the tedious details involved in identifying and retrieving a particular document. Since the development and rapid widespread adoption of these conventions, they have been extended and integrated with other new related technologies that provide for the delivery of content that is much more dynamic in nature. The most important of these related developments has been the introduction and rapid widespread adoption of the Java programming language as a standard for Internet-based computation.

The integration of the Web and Java represents a technological advancement that enables a fundamentally new approach to simulation modeling, one that makes possible the development of environments with coherent Web-based support for collaborative model development, dynamic multimedia-based documentation, as well as open widespread execution and investigative analysis of models. A key aspect to the approach is the role the Java language plays in both the specification and implementation of the model.

The evolution to high-level model specification languages and formalisms has been motivated by the desire to make simulation more accessible by eliminating the programming burden. However, such systems are often difficult to modify or extend because of an imposed separation between the specification system and its implementation. This can lead to models that poorly mirror system behavior and have no potential for distribution and reuse within an enterprise. The Java language is ideally suited to implementing an advanced simulation architecture whose features are readily accessible at the programming language level, special purpose simulation language level, and high-level model specifications.

Specifically, key features like the well-designed object-oriented nature of Java and native support for multithreaded execution allow special purpose simulation modeling features to be incorporated directly into the Java language in a natural way so that the underlying modeling and programming languages are the same. These relatively low-level but powerful modeling capabilities can in-turn be used to implement higher-level model specification systems via the JavaBeans component development model. The simple programming conventions that constitute JavaBeans allow Java-based software components to be assembled visually into applications using any of a growing number of sophisticated graphical programming environments including Symantec's Visual Café, Microsoft's J++, IBM's Visual Age, Sun's Java Workshop, Borland's Jbuilder, and Lotus's BeanMachine. When visually assembling predefined simulation modeling components, no programming is required; however, when necessary, the user has access to the underlying code and full power and

flexibility of the Java programming language. What's more, any Java environment can be used for model building and debugging. The modeling language capabilities and predefined component assembly capabilities can also be used in isolation or in combination to produce high-level standalone simulation applications that users interact with in predefined ways.

The hardware and operating system independent design of Java facilitates collaboration by allowing modelers to share language level or component level models independent of where they were developed. The documentation and deployment of modeling tools and end-user applications via the Web also serves to make open and widespread both the development and investigative analysis of models.

This vision of Web-based simulation is the motivation behind Thread Technologies' design of *Silk*™, a general purpose simulation language based on the Java programming language. *Silk* merges familiar process-oriented modeling structures with powerful object-oriented language features in an intelligent design that encourages model simplicity and reusability through the development and the visual assembly of *Silk* modeling components in JavaBeans-based graphical software environments. More generally, *Silk*'s openly extensible, scalable, and platform independent design represents the type of approach that is essential to keeping simulation modeling on track with other revolutionary changes taking place in Internet-based computing.

3.4 Simulation Modeling Methodology in the Wonderfully Webbed World (Richard E. Nance)

While modeling methodology has been with us since the inception of simulation, it remained indistinguishable from programming throughout the first two decades. Nevertheless, a few early researchers abstracted beyond the executable form to search for more significant semantic revelations. Lackner and Kribs [11] and Kiviat [10] are prominent examples, but Tocher's [18] wheel charts to assist in model specification and the IFIP proceedings on simulation programming languages [2] show that interest was widespread. Efforts to derive a theory of simulation [19] generated interest in model representation in the 1970s. The latter part of the decade ushered in the first specific focus on modeling methodology (model life cycle, model specification languages, the DELTA project) [12]. With the 1980s came the vision of model development environments [13] that are now a commercial reality. Is the subject of this panel session presaging the next major transition in simulation model development?

3.4.1 Modeling Methodology

Since "methodology" is both over-used and misused, a definitional explanation in this context is appropriate. Methodology, following the view of Arthur et al. [1, p. 4], should:

- organize and structure the tasks comprising the effort to achieve global objectives,
- include methods and techniques for accomplishing individual tasks (within the framework of global objectives), and
- prescribe an in which certain classes of decisions are made and the ways of making those decisions leading to desired objectives.

Key in the attainment of the objectives are the *principles* that form the foundational support of a methodology.

3.4.2 Influence of the Web

If the world wide web is to effect major changes in modeling methodology, then it must alter or abolish existing principles or introduce new principles. At this juncture the capability of the web to influence the technology of model building, model execution and model sharing is clear, and the degree of change appears significant. However, that the potential for influence extends into the principles – the foundational core – is less apparent.

3.5 Web-Based Simulation: Whither We Wander? (Ray J. Paul)

This panel contribution will discuss a variety of new technologies for software development and ways of working that will have an unpredictable effect on the future of simulation modelling.

3.5.1 Multi-media/Synthetic Environments

The ability to access multi-media on the web clearly introduces greater potential for the use of videos of problem scenarios, for interaction with stake-holders situated at remote locations (for example, when the running model hits an unknown combination of circumstances, an expert stake-holder might be able to determine the successful rules for advance) and sound. For example, on a recent visit made to a Hong Kong container terminal, I was shown a television control centre, computer-based, which had 100% video coverage of the terminal. Whilst its purpose was clearly for security and safety, it requires little imagination to visualise how a simulation of the terminal operation could call up the appropriate video camera when problem discussants get to the point of a simulation run where clarification is desired. I think that the rush to join the much-hyped band-wagon of Synthetic Environments, driven by technical extravagance and financial greed, is in great danger of neglecting or even forgetting those major simulation issues of ongoing concern over the years. These are the so far intractable problems of verification and validation. The current enthusiasm for Synthetic Environments is therefore in danger of creating more expensive mistakes to the detriment of the reputation of simulationists, analysts and operations researchers in general.

3.5.2 Natural Born Webbers

A large proportion of the current generation of students entering higher education in the developed countries are already familiar with the pastime of browsing the Web and playing computer games. Both of these activities might loosely be depicted as approaches based on "suck it and see". Browsing and adventure games encourage the participant to try out alternatives with rapid feed-back, avoiding the need to analyse a problem with a view to deriving the result.

Such web users, in order to use simulation, need and desire development tools that allow for fast model building and quick and easy experimentation. Furthermore, such web users should have a natural affinity to the use of simulation models as a problem understanding approach [15, 16]. Web-enabled simulation analysts will be opposed to classical software engineering approaches and methodologies. They will be seeking tools that will enable them to assemble rather than build a model. Some feel for the change of "culture" that we can expect from future generations of computer users can be gauged from a recent experience of mine on a visit to Taipei (Taiwan). A class of school children were using the local university's multi-media lab. A ten year old schoolboy was typing in HTML codes faster than I can and dynamically checking it by running a rather impressive text/video/sound demonstration system. The boy could not speak, read or write any English, everything was symbolic to him.

3.5.3 New Software Technologies

Some have predicted that the software industry will be divided into component factories where powerful and general components will be built and tested, and into component assembly shops where these components will be assembled into flexible business solutions. Such component based development, if it occurs, might give significant gains in productivity and quality as well as known obvious benefits to web-based software development.

3.5.4 Java

Java is now so ubiquitous that it might appear unnecessary to comment on it. For completeness the reader is reminded that simulation models in Java can be made widely available; an applet can be retrieved and run and does not have to be ported to a different platform or even recompiled or relinked; there is a high degree of dynamism because Java applets run on a browser; Java built-in threads make it easier to implement

simulation following the process interactive paradigms; Java has built in supports for providing sophisticated animations and Java is smaller, cleaner, safer and easier to learn than C++, allegedly.

3.5.5 Conclusions

For me, the aforegoing indicates a world of dynamic change, which I welcome, but where it is all going is a matter of conjecture that will be colored more by prejudice and opinion than evidence.

4 Reactions

In this section the authors respond to the points made in the previous section.

4.1 Ray Paul's Comments

Regarding the positions of Arnold Buss and Kevin Healy, it is arguable that Java is so good. We have experience of platform dependence, and of course the rate of enhanced releases which are not downward compatible outdates software rapidly. On the other hand, such fast adaptation of the language might encourage improved methods of release compatibility, to the benefit of the industry at large.

Regarding Paul Fishwick's position, it is arguable that quality control is necessary for software re-use. The traditional methods of building large models, which takes much time and money, and which in itself then leads to an expectation of repeated use, demand some sort of quality assurance. When it takes so long to get an answer(s), it is a bit limp to also admit that the model may be indeterminately wrong! However, if we can "glue" bits together fast and experimentally (Ray's crystal ball in action here), then maybe the emphasis will shift dramatically from "is the model correct?" to "is the analysis, albeit with unproven software, acceptable given the large experimentation that swift modelling has enabled us to carry out in a short space of time?" In other words, the search space has been dramatically reduced not by accuracy (the old way), but by massive and rapid search conducted by an empowered analyst (the new way).

Regarding Richard Nance's position, maybe our current principles are inappropriate for a web-based world. I have already argued some of this in the previous paragraph. Here I go further. We are in a period of rapid technical change (though some authors claim this will come to an end and life will settle down again – see [6]). Every attempt we make to use these technological advances adds to the opening up of new opportunities to make change. This is particularly noticeable in business, where new companies are emerging fast, old ones sinking daily, mergers, acquisitions, takeovers, etc. are prevalent. Even in the military sphere, the nature of the task to be faced changes quickly (war, peace-keeping, policing, training allies, reassessing threat as the political world moves on and so forth). Analysis needs to be fast, else the problem has moved on anyway. Methods that produce ballpark estimates quickly, enhanced with more accurate methods if time allows, are or will be the order of the day. Principles based on output analysis, rather than modelling analysis, are likely to be more appropriate. If the traditional analytical and academic communities try to maintain current principles, they will become historians, worthy of a footnote about Luddite Neanderthals in the next Millennium history.

4.2 Arnold Buss's Comments

Regarding Ray Paul's comments, he has indeed brought up some thought-provoking issues. With regard to Java, although it is good to be skeptical, it is clear at this point that the only thing that will derail its achieving true platform independence is willful destruction, to wit, Microsoft's attempts to make it Windows-specific. There is, in my opinion, simply too large a critical mass of developers and companies who are getting on board for this to happen.

Moreover, I believe that the Java component infrastructure (JavaBeans) will be precisely the platform on which to assemble large models from smaller components, so that the entire monolith does not have to be designed in one piece. I believe that component-based design will supplant OO design in a major way in the near future, in part fueled by network-based computing. On the network, you *must* be component-oriented or the thing is just too unwieldy. Designing distributed models in a reasonable manner pretty much forces

you into components. The design issues focus more on responsibilities and interoperability rather than class hierarchies, as in traditional OO design.

There is a somewhat subtle aspect of the Java language that turns out to be the real winner for component-based design, namely interfaces (vice classes). Interfaces enable components to interoperate and pass messages *without having to know the precise class or class hierarchy of each other.* The interface is simply a contract to implement certain methods, so they may be invoked with compiler-safe impunity. Interfaces allow you to replace one object with another of an entirely different class with no necessary implied "is-a" relationship.

The second really important element is enabled by interfaces: communication via events. Interfaces allow you to define a small handful of event sources and event listeners that can provide communications between objects that is much more flexible than ordinary method invocation. One object will register its interest in another's events (or, more likely, be registered by a third party). Whenever the event source's state changes to trigger an event, all objects listening are notified. The key is that neither event sources nor listeners need to be "aware" that any of this is happening. Objects can register and un-register their interest as the program evolves. Event communication is a powerful means of implementing distributed models. Remote objects may easily register as listeners by using a remote mechanism (RMI, CORBA, etc.) and the event sources need not know (or care).

I have enhanced Simkit to incorporate this kind of messaging, and am currently working with a student to extend it further. For example, we have generic entities that are nothing more than containers. Functionality is put on these entities by creating and adding components. To enable movement, for example, a Mover entity is thrown in. The kind of movement possible is entirely determined by the type of Mover. Add a sensor and you can detect other things (depending, of course, on the kind of sensor). If a Mover and a Sensor are put together in a container, then the movement is governed by the Mover. Basically, this is an extremely flexible type of composition. It is difficult to express in standard OO notation, since neither the Mover nor the Sensor are instance variables. Besides, Booch/UML diagrams just tend to confuse matters in my opinion. The point is that a generic component-based methodology needs to be developed for such modeling. Java is a perfect vehicle for doing just that.

4.3 Richard Nance's Comments

Regarding Ray Paul's comments, I do not accept the claim that "it is a bit limp to also admit that the model may be indeterminately wrong." A model is not reality and only a fool insists that a model be error-free (the same person who wants a world with no accidents). How do you propose to answer the shifted question above: "Is the analysis ... acceptable?" I see your only recourse as an after-the-fact conclusion, which advances us to the stage of relying on prophets—why bother with the unproven software, etc? Having returned us to the technology of 2000 years ago, what next is to be offered? How about a roulette wheel with labeled outcomes?

Since the good Dr. Paul does not provide quotation marks or a page number, I assume that these sentiments are not those of Fernandez-Armesto but his own. My reaction is that I do not live in the fast lane that engulfs Dr. Paul. My long-time technical sponsor, the US Navy, is working now and for some three years prior on the design of the next destroyer (2003). The models and simulations used in this task are time-consuming to develop and the analysis is conducted over years, not days, hours or seconds. The hull will be in service for some 30 years and undergo three major overhauls all of which will require parts of the existing models and still others that will be developed, again perhaps in months, but certainly not in seconds.

I do not think our modeling methodology principles have been altered at all by the web. The capabilities of the tools based on the principles have changed and are changing, but that is the way technical progress is made.

My thanks to the good Dr. Paul for his agitating expressions of these misguided views. May he never fly on an aircraft developed with his analysis/prophet approach to decision making.

4.4 Paul Fishwick's Comments

Everyone on the panel makes valid points. There is a need to tie together some of the views to make a whole. At the same time, I'll express my own perspective on "where it all is going." Ernie Page's reference of Dijkstra's principle is one where we must separate specification from implementation. In general, this separation is one where we talk of "level of model." A piece of code, a mathematical expression, a Petri network, and a 3D cylinder are all examples of models. We may choose one specific model to represent some aspect of the system that we are studying. The code may represent the same dynamics as the Petri net, while the cylinder may represent either an abstraction of the system shape or, perhaps, a state of the system. The interpretation that we foster is the essence of modeling. Modeling is an art in this respect.

Arnie Buss and Kevin Healy speak kindly of Java. Java does show promise for its intended function: a computer language meant to migrate over a large area network to promote distributed computing. At the same time, Java is a textual computer language so its primary purpose is to represent dynamics at a fairly low level of abstraction. I'll submit that source code written in Java is a model, and that one can "think in Java" about system behavior. On the other hand, many people will not find this metaphor as appealing as one that is visual and graphical. Models must serve the user's view and way of thinking. There is no one correct modeling language. Ultimately, models are shared metaphors. If I am part of the Petri network or System Dynamics community, I think in these specific icons. The models color my thinking about dynamics. If we can all agree that we have many modeling types or languages, and that we can form translations among models (from Petri networks to Java, for example), then all forms of modeling become germane to the discussion. With Java at the lowest level of translation, our task of distributed execution of models is enhanced, and so research and development of Java is good for web-based modeling and simulation. Let's just keep in mind that Java is one of many nodes in a vast modeling network with models as nodes and translations as arcs. I know very few scientists or engineers who would prefer Java over their pet modeling methodologies.

Dick Nance and Ray Paul speak of two opposite poles in terms of quality in modeling. If I attend an art exhibition and buy a modern sculpture made of electronic home appliances—such as toasters, can openers, and mixers—I will most likely not use this artwork for engineering purposes. When special effects companies in Hollywood create models of the Titanic and of New York City, their objective is to foster entertainment and not to create statistically valid behavior. Therefore, both views are supported. Quality must be maintained where it is required, and to the degree that it is required depending on overall objectives of the simulation. There is nothing wrong with the Taiwanese schoolboy (Paul's example) who grabs objects left and right to create a new experience. Some of these objects, like the toaster in the sculpture, may be based on high resolution models (both structural and dynamic). It is the way in which the objects are used that determines the outcome, and all outcomes are fair game.

Luckily, the future is bright for simulation and web-based modeling and simulation. Imagine the Taiwanese schoolboy unchained from abstract languages such as HTML. Instead, a new world-wide marketplace of digital objects yields the digital equivalent of everything you see around you. The web is no longer fettered with documents. Documents are but one kind of physical object. The schoolboy will be creating complex games and simulations for his friends who will later join him in a multiplayer extravaganza. Meanwhile, the Navy is testing out a new class of submarine using objects delivered by its contractors. This delivery occurs well before the physical submarine components are manufactured. Some of the Navy objects will be the same used by the schoolboy just as the toaster can be used in more than one way. The objectives of the Navy and schoolboy are different, but the digital object marketplace is common to both of them. I think that Dick Nance is right about a change in modeling methodology. It is happening now and web-based simulation is the catalyst. The purpose of physical objects is to achieve a singular objective, but the global objective is left open to the end-user. This is a departure from Arthur et al. [1] We should not limit our models by global objectives. Objectives and models are orthogonal. I use the web to locate objects, and I use these objects to create models of every variety. Like the manufacturer of the toaster, I create the best digital toaster possible and let the consumer make the choice as to its utility. I would not be at all surprised to go to Taiwan in ten years and find the schoolboy playing a "multi-player deathmatch" inside the confines of a greatly enlarged toaster within a Dali-esque landscape. Meanwhile, the Navy is modeling the high-level dynamics of a towed-array sonar using a circuit of light bulbs from General Electric's web site, with bulbs representing states.

5 Summary

The era of web is certainly upon us. There seems to be no escaping that fact. The confluence of the web and simulation offers an opportunity to change the way we approach modeling. How much should we embrace such change? The panelists disagree on this point. If the world of digital objects appears—and if publishing models on the web becomes profitable, digital objects will proliferate—will the modeling process become enhanced or impaired? Certainly the act of model construction would be simpler—assuming sufficiently powerful search engines. It should be much easier to "plug" models together than to build models from scratch. But then what? How will models be validated in this environment? Unless the open source movement achieves ubiquity, model validation may be one big exercise in black-box testing. In areas where validation is critical, this situation can only represent a step backwards.

But perhaps an engineering analogy is useful here. No one would argue, for example, that bridges are a bad idea. Although occasionally failures do occur (and such failures can be catastrophic) for the most part bridges are engineered for safety. Where possible, pathological situations are considered and accounted for in bridge design. Worst-case capacities (and then some) are accommodated. The opportunity to misapply the science and mathematics that support bridge design exists, but the engineering profession actively seeks to limit such opportunities.

Technology marches on. Modeling is central to technological advancement. But advancing technology impacts the modeling process as well. As simulation becomes a desktop commodity, it will be available to masses. This ubiquity is a mixed blessing. Having access to such a powerful problem-solving technique is potentially quite valuable. On the other hand, to the untrained user—a user with a what-you-see-is-what-you-get perspective—the potential to misapply the technique is great. As responsible engineers of the future, should those enabling the web-based simulation revolution also shepherd the safety of the technique?

References

[1] Arthur, J.D., R.E. Nance and S.M. Henry. 1986. "A Procedural Approach to Evaluating Software Development Methodologies and Associated Products." SRC-87-007, Systems Research Center, Virginia Tech, Blacksburg, VA.

[2] Buxton, J.(ed). 1968. *Proceedings of the IFIP Working Conference on Simulation Programming Languages,* North-Holland.

[3] Cubert, R. and Fishwick, P.A. 1997. "MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework" accepted for *Simulation,* July 1997. Further information on MOOSE can be found in `http://www.cise.ufl.edu/ ~fishwick/moose.html`

[4] Dijkstra, E.W. 1976. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ.

[5] Erkes, J.W.; Kenny, K.B.; Lewis, J.W.; Sarachan, B.D.; Sobolewski, M.W; and R.N. Sum, Jr. 1996. "Implementing Shared Manufacturing Services on the World-Wide Web." *Communications of the ACM,* 39, no. 2 (Feb.):34-45.

[6] Fernandez-Armesto, F. 1995. *Millennium: A History of the Last Thousand Years.* Touchstone, Inc. New York, NY.

[7] Fishwick, P.A. 1995. *Simulation Model Design and Execution,* Prentice-Hall, Englewood Cliffs, NJ..

[8] Fishwick, P.A. 1996. "Web-Based Simulation: Some Personal Observations" In: *Proceedings of the 1996 Winter Simulation Conference,* (San Diego, CA, Dec. 8-11). Association for Computing Machinery, New York, 772-779.

[9] Fishwick, P.A., Hill, D.R.C. and Smith, R., Eds. *Proceedings of the 1998 International Conference on Web-Based Modeling and Simulation,* SCS Simulation Series **30**(1), San Diego, CA, 10-14 January 1998.

[10] Kiviat, P.J. 1967. "Digital Computer Simulation: Modeling Concepts." RAND Memo RM-5883-PR, Santa Monica, CA.

[11] Lackner, M.R. and P. Kribs. 1964. "Introduction to a Calculus of Change." System Development Corp., TM-1750/000/01.

[12] Nance, R.E. 1979. "Model Representation in Discrete Event Simulation: Prospects for Developing Documentation Standards." In *Current Issues in Computer Simulation,* N. Adam and A. Dogramaci (eds), Academic Press, 83-97.

[13] Nance, R.E. 1983. "A Tutorial View of Simulation Model Development." In: *Proceedings of the 1983 Winter Simulation Conference,* (Arlington,VA, Dec 12-14). IEEE, Piscataway, NJ, 325-331.

[14] Page, E.H., Buss, A., Fishwick, P.A., Healy, K.J., Nance, R.E. and Paul, R.J. 1998. "The Modeling Methodological Impacts of Web-Based Simulation," In: *Proceedings of the 1998 SCS International Conference on Web-Based Modeling and Simulation,* pp. 123-128, San Diego, CA, 11-14 January.

[15] Paul R.J and D.W. Balmer 1993. *Simulation Modelling.* Lund, Sweden: Chartwell-Bratt Student-Text Series.

[16] Paul R.J. and V. Hlupic 1994. "The CASM Environment Revisited." In: *Proceedings of the 1994 Winter Simulation Conference,* (J.D. Tew, S. Manivannan, D.A. Sadowski and A.F. Seila, Eds.) (11-14 December 1994, Orlando). Association for Computing Machinery, New York, 641-648.

[17] Swartout W.; and R. Balzer. 1982. "On the Inevitable Intertwining of Specification and Implementation." *Communications of the ACM,* 25, no. 7(July):438-440.

[18] Tocher, K.D.T. 1966. "Some Techniques of Model Building." In: *Proceedings, IBM Scientific Symposium on Simulation Models and Gaming,* pp. 119-155, White Plains, NY.

[19] Zeigler, B.P. 1976. *Theory of Modelling and Simulation,* Wiley, New York.

A Modeling Strategy for the NASA Intelligent Synthesis Environment

Paul A. Fishwick

Department of Computer & Information Science & Engineering
University of Florida
Gainesville, Florida 32611, U.S.A.

July 13, 1999

Abstract

We overview the goals of NASAs Intelligent Synthesis Environment (ISE) from the perspective of *system modeling.* Some of the problems with present day modeling are discussed, followed by a suggested course of action where models as well as their objects are specified in a uniform representation based on the Virtual Reality Modeling Language (VRML). Existing dynamic modeling techniques tend to be 2D in form. The Rube methodology and application provides a 3D modeling framework where model components are objects, and all objects are defined in such a way that they can be easily defined within web documents. This approach suggests the formation of reusable *digital objects* that contain models. **Keywords: Modeling, Metaphor, Abstraction, Simulation**

1 Modeling the Future

NASA is reinventing itself with respect to new challenges that will culminate in more frequent, less expensive missions. A recent paper by Goldin, Venneri and Noor [10] covers some of the sweeping changes that are to engulf NASA and project it well into the next century. The Intelligent Synthesis Environment (ISE) represents a key piece of the new NASA. How can we begin with a conceptual design for a new spacecraft and take this design through the stages of analysis, testing and fabrication while maintaining the highest level of quality? We enable ourselves to step through sections of the gauntlet with ease if we can generate effective *modeling* methods. Modeling represents a significant part of ISE since it is with modeling that synthesis of spacecraft is made manifest. ISE is divided into five elements. While the general role of modeling is pervasive in all areas, it is strongest in the ISE elements *Rapid Synthesis and Simulation Tools* and *Collaborative Engineering Environment.* It is certainly

cheaper to build a virtual spacecraft for Cassini or the Deep Space missions than to construct the actual hardware. And yet, modeling is not without its problems. Modeling can be extraordinarily complex—both in representational schemes and in the iterative procedures required to evolve models over time. My goal is to focus on the modeling aspect of ISE to recommend specific changes in how we design dynamic models that blend seamlessly with the 3D objects being modeled. Models do not have material components. They are ethereal and live inside the computer. It is through the efficient practice of modeling that NASA will jumpstart itself into a more efficient future.

NASA Centers are embracing the goals of ISE. Kennedy Space Center [4] has the *virtual Shuttle operations model* to support ground processing. JPL is improving the approach to engineering spacecraft from design to fabrication. The Develop New Products (DNP) initiative has generated significant research in methods for improving engineering design and the processes associated with design. Smith [20] and Wall et al. [22, 21] define approaches to modifying the existing NASA engineering design practices through model-based means. They point out that much of what exists today in NASA reflects a "document-based" approach to design. A model-based approach is a significant step toward a more manageable process. A related problem is where code is used instead of models. The recognition that we need to surface models will naturally lead to more effective and cost-efficient simulations, where the code is automatically compiled, translated from the model's structure. The DNP design cycle is typically divided into the following processes: Mission and System Design (MSD), Design, Build and Test (DBAT), Validate, Integrate, Verify and Operate (VIVO) and Project Leadership and Planning (PLP). Rather than being sequential, these are concurrent and hierarchically related processes with PLP being on the top and proceeding downward to the lower levels of administrative detail as follows: $PLP \Rightarrow MSD \Rightarrow DBAT \Rightarrow VIVO$. The mission is the top-most concern of any NASA process after a project is created. The mission defines what tasks are to be done, and in what order. Sample-based missions involve the collection of material from a comet or a planet's surface. A mapping-based mission would map the surface of a planet or its satellites. Most missions are multi-facetted. For example, Cassini involves flybys and mapping of planets, a moon of Saturn as well as instrumentation for atmospheric experiments for the released Titan probe. The DNP goal is to build cross-cutting (XCUT) models that span all aspects of the mission. If a mission begins with a modeled mission and modeled spacecraft then there will be easier and more effective collaboration among designers, engineers and manufacturing staff. Off the shelf commercial software for data flow diagrams and state-based diagrams have been used recently for elaborating modeled spacecraft subsystems.

The ISE Goal is to "develop the capability for personnel at dispersed geographic locations to work together in a virtual environment, using computer simulations to model the complete life-cycle of a product/mission before commitments are made to produce physical products" [15]. This is an ambitious goal, but it is on target with the increasing use of modeling and simulation to improve the efficiency by which we design and manufacture

components, machines, aircraft carriers, process plants and spacecraft. It also builds upon existing NASA projects (i.e., DNP) that attempt to steer engineering beyond paper and documents to digital representations of objects. In short, we need to use today's cheap computer technology to manufacture virtual equivalents of what we buy and sell.

One of the problems with DNP is that it uses a centralized parameter database, around which programs are situated so that each program reads-from and writes-to the database. This central hub-spoke approach is an improvement over having separate programs each with separate data files and repositories; however, a cleaner approach is to create an object-oriented scene where all data are associated with the relevant objects. The central database approach [9] was also used for the NASA Integrated Programs for AeroSpace Vehicle Design (IPAD) Project [8]. IPAD used a relational database to store structure-based parameters to be used by CAD and Finite Element programs at NASA Langley Research Center. This was a dramatic improvement over separate data files, but it suffered from the fragmentation of connecting data and model to the encapsulating object. With the design and creation of a spacecraft, the scientists and engineers will interact and focus on the physical item—the spacecraft itself. If we can create a process where we build a completely digital spacecraft, then we will maintain this necessary collaboration among all programmers, modelers and engineers. Parameters of a high-gain antenna should be made available to the engineer who touches the antenna; the parameters need to be stored within the objects they define. Moreover, all information about the spacecraft should be so oriented so that we also get to programs and models via the digital spacecraft. The spacecraft itself becomes the primary interface for all related models, programs, and data. Higher level abstract concepts such as the *mission* can be materialized into objects that remind us of the basic mission elements.

2 Problems in Modeling

There are a number of problems that must be addressed once we begin to model. These problems are by no means intrinsic to NASA. They are general problems of the larger modeling community. Even though, a large segment of the engineering community acknowledges the importance of modeling, the overall process modeling is not without its share of defects. Significant changes need to be instrumented if we are to make modeling effective, for if it is not a truly economic enterprise, modeling and simulation will always be seen as choices of last resort, or "to be performed only when time and resources permit." Let's highlight important modeling issues:

- *Modeling Freedom:* Many types of modeling exist, with mathematical models being only one type. We need to reemphasize that models are for humans—not computers. Therefore the models must appeal to the human senses to be effective.

- *Modeling vs. Validation:* The aspects of modeling that we discuss are based on design principles. Even though models are said "to be good" when they validate physical phenomena, all models are flawed in this sense—the model shows us a window of valid behavior of an object and we use it to augment our intellect and otherwise mathematical methods. The Bohr billiard ball model of the atom is still very useful when used correctly even though we realize that billiard balls are not to be taken literally [11, 1]. Validation is separate from modeling but is to be used in conjunction with it. Modeling is what we do to understand and reason about a thing. Validation is taking a model and comparing the model's prediction with experiment.

- *Code vs. Model:* It is all too frequent that when one speaks of "a model" that one is referring to an abstract representation that bears little or no formal relation to the computer code that is supposed to *represent the model.* While code can be viewed as a model in its own right, more common model forms are based on both equational and highly-visual structures. It is essential to have *generated code* be driven from the model and so all interaction is directly through the model so that we can best forget the code, since code represents the cement whereas the model represents the multi-tiered building created from the cement.

- *Programs vs. Models:* How do computer programs and models relate? The differences between programming, as we generally learn it in Universities, and modeling reflect a gradual change in our software and hardware technologies. Computer Science and Engineering stands out as being separate from other Engineering disciplines in the sense that everyone else talks about matter and physics and computer scientists talk of data structures, procedures, relations and objects. Programming has evolved with a heavy bias toward mathematical representation. The problem is that this sort of representation bears little direct connection to physics or to other engineering disciplines. Fortunately, movements are underway in many computer science areas that suggest alternate, more physical, representational structures [17, 2, 19].

- *Integration:* NASA is in need of truly integrated virtual, 3D environments where the objects to be modeled, as well as their models, live in the same space. To determine the dynamics of the Cassini probe destined for Titan, one need only touch the 3D probe (attached to the orbiter), activate its *behavior field* object and then navigate the dynamics that are surfaced in a 3D form. Achieving this means that we have to free the process of dynamic modeling from its two-dimensional home where it has been imprisoned. Humans better understand and reason with environments that are similar to those found in every day life. Data defining parameters of spacecraft science instrumentation, for example, should be co-located with the virtual instruments. Parameters are part of objects and the engineer wants to reason and work with these parameters through the virtual objects that the data represent or modify. It may well be that a very low-level underlying database schema supporting such interaction is still

needed, but it is critical to maintain the virtual connections to the data through the physical spacecraft components. This might be seen as an issue of *visualization* or *user interface* and it is. The act of modeling is all about developing and fostering sensory appeal of the human to the modeled object. Thus, it becomes impossible to separate the discipline of human-computer interaction from the task of modeling. They are one and the same. The relational or hierarchical database should disappear from view since it bears no relation to the spacecraft.

3 The Nature of Modeling

One physical object captures some information about another object. If we think about our plastic toys, metal trains and even our sophisticated scale-based engineering models, we see a common thread: to build one object that says something about another—usually larger and more expensive—object. Let's call these objects the *source object* and the *target object.* Similar object definitions can be found in the literature of metaphors [12] and semiotics [18]. The source object *models* the target, and so, modeling represents a relation between objects. Often, the source object is termed *the model* of the target. We have been discussing scale models identified by the source and target having roughly proportional geometries. Scale-based models often suffer from the problem where changing the scale of a thing affects more than just the geometry. It also affects the fundamental laws applied at each scale. For example, the hydrodynamics of the scaled ocean model may be different than for the real ocean. Nevertheless, we can attempt to adjust for the scaling problems and proceed to understand the larger universe through a smaller, more manipulable, version.

Later on in our education, we learned that modeling has other many other forms. The mathematical model represents variables and symbols that describe or model an object. Learning may begin with algebraic equations such as $d = \frac{1}{2}at^2 + v_o t + d_0$ where d, v and a represent distance, velocity and acceleration, and where d_0 and v_0 represent initial conditions (i.e., at time zero) for starting distance and initial velocity. These models are shown to be more elegantly derived from Newton's laws, yielding ordinary differential equations of the form $f = ma$. How do these mathematical, equational models relate to the ones we first learned as children?

To answer this question, let's first consider what is being modeled. The equations capture attributes of an object that is undergoing change in space (i.e., distance), velocity and acceleration. However, none of the geometrical proportions of the target are captured in the source since the structure of the equations is invariant to the physical changes in the target. A ball can change shape during impact with the ground, but the equations do not change their *shape.* If a ball represents the target, where is the source? The source is the medium in which the equations are presented. This may, at first, seem odd but it really is no different than the toy train model versus the actual train. The paper, phosphor or blackboard—along with the

Figure 1: Painting by Rene Magritte. Is it a pipe or a *model* of a pipe?.

medium for the drawing, excitation or marking—has to exist if the equations are to exist. In a Platonic sense, we might like to think of the equations as existing in a separate, virtual, non-physical space. While one can argue their virtual existence, this representation-less and non-physical form is impractical. Without a physical representation, the equation cannot be communicated from one human to another. The fundamental purpose of representation and modeling is communication. Verbal representations (differential air pressure) are as physical as those involving printing or the exciting of a phosphor via an electron beam. Figure 1 displays a painting by the French surrealist artist Magritte, which captures the essence of *semiotics* and reminds us that source and target objects both must exist. The painting includes a phrase in French "This is not a Pipe." The object is a painting representing a pipe, or more accurately, it is a piece of paper representing a painting that, in turn, represents a pipe. In the same sense as Magritte's painting isn't a pipe, likewise, the equations are (source) objects that we interpret as attributes of other (target) objects. We see an equation and think of the target's attributes. This leaves us with the wonderful thought that when we model, regardless of the type of model, we use different objects to represent the attributes of other objects. It takes some serious practice to imagine that strange ink impressions on paper might actually represent the position of a ball, train, or horse but that is part of the wonder of modeling and of our ability to perform *abstraction*: any object can be used as a surrogate for another object's attributes. In this sense, the more abstract a source object in its relation to the target, the fewer attributes will be found to be in common: a scale model of a train preserves geometry under the right scale transformations whereas the paper and ink (representing equations) preserves none of this geometry. The equations are said to be more abstract than the scale model. There is one thing to keep in mind regarding mathematical and event 2D image-based models. We use them so frequently because of economic reasons

and not because they reflect the best and most natural ways to model. Creating a scale model of the ocean is much easier than using the real ocean. But using a piece of paper or a blackboard is even easier. What if one could create virtual 3D spaces with ease on a portable digital assistant (PDA) device? In the far future, we may even approach the environment of the Holodeck as demonstrated in Star Trek: The Next Generation. The Holodeck is a physical space where humans enter fully immersive and interactive 3D simulations. What will modeling be like in such an environment? Will we still draw things on paper or will we gesture to each other while forming 3D worlds that appear before our eyes? The ultimate goal of modeling is not that different than what we did in the sandbox. The difference is that now we can make a virtual sandbox.

4 Rube: Building the Infrastructure

Since 1989 at the University of Florida, we have constructed a number of modeling and simulation packages. We'll begin with some early packages and proceed toward our development of the Rube environment. The web will become a repository for objects as well as documents. The first package was a set of C programs called *SimPack* [6]. SimPack is a collection of C libraries and programs to allow the student to learn how to effectively simulate discrete event and continuous systems. Discrete event simulation involves irregular leaps through time, where each leap is of a different duration. Discrete event simulation requires scheduling, event list data structures, and an ability to acquire resources and to set priorities. Continuous simulation involves stepping through time using equal-sized time intervals, and is most often associated with systems based ultimately on physical laws. SimPack began as a library for discrete event handling and grew to support continuous modeling (with difference, ordinary and delay-differential equation editors). Fully interactive programs were built upon the core routines and inserted into the SimPack distribution. SimPack is widely used by a number of sites worldwide.

By the early 90s, object-oriented programming was becoming increasingly common in simulation. This suggested that we re-engineer part of SimPack to address the advantages afforded by encapsulation, class hierarchies and re-use. In 1994, we announced OOSIM. OOSIM development started with the event scheduling library in SimPack and expanded upon it to make it more robust using C++.

Both SimPack and OOSIM were found lacking in the user-interface area. Most model types used by scientists and engineers are visual. While we can encode such models in text files, the user doesn't really get a good feel for a model unless it is surfaced in a visible form. In 1997, we began development on a fully visual and interactive multimodeling system, OOPM (Object Oriented Physical Modeler) [5]. Multimodeling [7] is the practice of creating a model at one level of abstraction where each model component can be refined at a level below into a model of a different type than the one at the level above it [14]. For example,

the state components of a finite state machine can be refined into differential equations (a different model type). OOPM is based on OOSIM and has a large amount of Tcl/Tk code to support the graphical user interface (GUI). A distributed simulation executive (DSX) has also been constructed for allowing functional block model components to be distributed over the net, where each block represents a legacy code responsible for an individual simulation. This system has recently been completed. During OOPM development, we learned a number of lessons. The first lesson was that even though *multimodeling* had been explained with several formal examples, we lacked an implementation and we had to carefully work out how the scheduling of "multimodel trees" was to done. The second lesson learned was that GUI development was extremely time consuming. Although everyone wants to use a GUI, one must recognize the significant software engineering effort involved in creating a robust interface. What may appear to be very minor problems from the software engineer's viewpoint turn out to be critical errors from the standpoint of a human-computer interface. We found out that it is often better to have a primitive text-based interface that is robust than a more complex GUI that has even a very small number of user interface anomalies. Users must develop trust in an application if they are to use it with confidence.

In late 1998, we started designing Rube, named in dedication to Rube Goldberg [16], who produced many fanciful cartoon machines, all of which can be considered *models* of behavior. The procedure for creating models is as follows:

1. The user begins with an object that is to be modeled. For JPL, this can be the Cassini spacecraft with all of its main systems: propulsion, guidance, science instrumentation, power, and telecommunication. If the object is part of a larger scenario, this scenario can be defined as the top-most root object.

2. A *scene* and interactions are sketched in a story board fashion, as if creating a movie or animation. A scene is where all objects, including those modeling others, are defined within the VRML file. VRML stands for Virtual Reality Modeling Language [3], which represents the standard 3D language for the web. The Rube model browser is made available so that users can "fly though" an object to view its models without necessarily cluttering the scene with all objects. However, having some subset of the total set of models surfaced within a scene is also convenient for aesthetic reasons. The modeler may choose to build several scenes with models surfaced, or choose to view objects only through the model browser that hides all models as fields of VRML object nodes.

3. The shape and structure of all Cassini components are modeled in any modeling package that has an export facility to VRML. Most packages, such as Kinetix 3DStudioMax and Autodesk AutoCAD have this capability. Moreover, packages such as CosmoWorlds and VRCreator can be used to directly create and debug VRML content.

4. VRML PROTO (i.e., prototype) nodes are created for each object and component.

This step allows one to create *semantic attachments* so that we can define one object to be a behavioral model of another (using a *behavior* field) or to say that the Titan probe is part of the spacecraft (using a *contains* field), but a sibling of the orbiter. Without prototypes, the VRML file structure lacks semantic relations and one relies on simple grouping nodes, which are not sufficient for clearly defining how objects relate to one another.

5. Models are created for Cassini. While multiple types of models exist, we have focused on dynamic models of components, and the expression of these components in 3D. Even textually-based models that must be visualized as mathematical expressions can be expressed using the VRML text node. Models are objects in the scene that are no different structurally from pieces of Cassini—they have shape and structure. The only difference is that when an object is "modeling" another, one interprets the object's structure in a particular way, using a dynamic model template for guidance.

6. Several dynamic model templates exist. For Newell's Teapot (in Sec. 5), we used three: FBM, FSM, EQN and for Cassini (in Sec. 6), we used one: FSM. These acronyms are defined as follows: FSM = Finite State Machine; FBM = Functional Block Model; EQN = Equation Set. Equations can be algebraic, ordinary differential, or partial differential.

7. The creative modeling act is to choose a dynamic model template for some behavior for Cassini and then to pick objects that will convey the meaning of the template within the scenario. This part is a highly artistic enterprise since literally any object can be used. In VRML, one instantiates an object as a *model* by defining it: `DEF Parthenon-Complex FSM {...}`. In other words, a collection of Parthenon-type rooms are interconnected in such a way that each Parthenon-Room maps to a state of the FSM. Portals from one room to another become transitions, and state-to-state transitions become avatar movements navigating the complex.

8. There are three distinct types of roles played modelers in Rube. At the lowest level, there is the person creating the *model templates* (FSM,FBM,EQN,PETRI-NET). Each dynamic model template reflects an underlying system-theoretic model [7]. At the mid-level, the person uses an existing model template to create a *metaphor*. A Parthenon-Complex as described before is an example of an architectural metaphor. At the highest level, a person is given a set of metaphors and can choose objects from the web to create a model. These levels allow modelers to work at the levels where they are comfortable. Reusability is created since one focuses on the level of interest.

9. The simulation proceeds by the modeler creating threads of control that pass events from one VRML node to another. This can be done in one of two ways: 1) using VRML Routes, or 2) using exposed fields that are accessed from other nodes. Method 1 is familiar to VRML authors and also has the advantage that routes that extend

from one model component to an adjacent component (i.e., from one state to another or from one function to another) have a topological counterpart to the way we visualize information and control flow. The route defines the topology and data flow semantics for the simulation. Method 2 is similar to what we find in traditional object-oriented programming languages where information from one object is made available to another through an assignment statement that references outside objects and classes. In method 1, a thread that begins at the root node proceeds downward through each object that is role-playing the behavior of another. The routing thread activates Java or Javascript Script nodes that are embedded in the structures that act as models or model components for the behaviors.

10. Pre- and Post-processing is performed on the VRML file to check it for proper syntax and to aid the modeler. Pre-processing tools include wrappers (that create a single VRML file from several), decimators (that reduce the polygon count in a VRML file), and VRML parsers. The model browser mentioned earlier is a post-production tool, allowing the user to browse all physical objects to locate objects that model them. In the near future, we will extend the parser used by the browser to help semi-automate the building of script nodes.

Rube treats all models in the same way. For a clarification of this remark, consider the traditional use of the word "Modeling" as used in everyday terms. A model is something that contains attributes of a target object, which it is modeling. Whereas, equation and 2D graph-based models could be viewed as being fundamentally different from a commonsense model, Rube views them in exactly the same context: everything is an object with physical extent and modeling is a relation among objects. This unification is theoretically pleasing since it unifies what it means to "model" regardless of model type.

5 Example 1: Newell's Teapot

In the early days of computer graphics (c. 1974-75), Martin Newell rendered a unique set of Bézier surface spline patches for an ordinary teapot, which currently resides in the Computer Museum in Boston. The teapot was modeled by Jim Blinn and then rendered by Martin Newell and Ed Catmull at the University of Utah in 1974. More recently, Fish produced the image of the teapot in Fig. 2, which has the nice property of showing the internal and external teapot shape. While at this late date, the teapot may seem quaint, it has been used over the years as an icon of sorts, and more importantly as a benchmark for all variety of new techniques in rendering and modeling in computer graphics. The Teapot was recently an official emblem of the 25th anniversary of the ACM Special Interest Interest Group on Computer Graphics (SIGGRAPH).

One of our goals for Rube was to recognize that the Teapot could be used to generate

Figure 2: Newell teapot rendering by Russ Fish, Copyright © 1995, University of Utah

another potential benchmark—one that captured the entire teapot, its contents and its models. The default teapot has no behavior and has no contents; it is an elegant piece of geometry but it requires more if we are to construct a fully *digital teapot* that captures a more complete set of knowledge. In its current state, the teapot is analogous to a building façade on a Hollywood film studio backlot; it has the shape but the whole entity is missing. In VRML, using the methodology previously defined, we built TeaWorld in Fig. 3 As in Fig. 2, we have added extra props so that the teapot can be visualized, along with its behavioral model, in a reasonable contextual setting. The world is rendered in Fig. 3 using a web browser. *World* is the top-most root of the scene graph. It contains a *Clock*, *Boiling_System*, and other objects such as the desk, chairs, floor and walls. The key fields in Fig. 4 are VRML nodes of the relevant field so that the *contains* field is refers to multiple nodes for its value. This is accomplished using the VRML *MFNode* type. The hierarchical VRML scene graph for Fig. 3 is illustrated in Fig. 4. The scene contains walls, a desk, chair and a floor for context. On the desk to the left is the teapot which is filled with water. The knob controlling whether the teapot heating element (not modeled) is on or off is located in front of the teapot. To the right of the teapot, there is a pipeline with three machines, each of which appears in Fig. 3 as a semi-transparent cube. Each of these machines reflects the functional behavior of its encapsulating object: *Machine1* for *Knob*, *Machine2* for *Water* and *Machine3* for *Thermometer*. The *Thermometer* is a digital one that is positioned in *Machine3*, and is initialized to an arbitrary ambient temperature of 0° C. Inside *Machine2*,

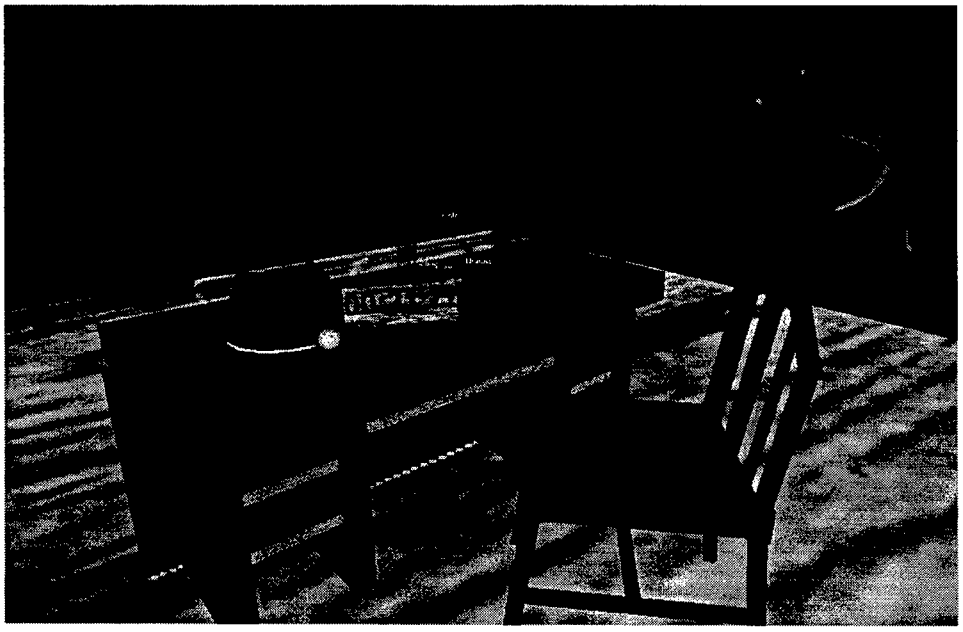


Figure 3: Office scene with Newell Teapot, dynamic model and props.

we find a more detailed description of the behavior of the water as it changes its temperature as a result of the knob turning. The plant inside *Machine2* consists of *Tank1*, *Tank2*, *Tank3*, and four pipes that move information from one tank to the next. Inside of each tank, we find a blackboard on which is drawn a differential equation that defines the change in water temperature for that particular state. The following modeling relationships are used:

- *Pipeline* is a Functional Block Model (FBM), with three functions (i.e., machines).
- *Machine* is a function (i.e., semi-transparent cube) within an FBM.
- *Plant* is a Finite State Machine (FSM) inside of Machine 2.
- *Tank* is a state within a FSM, and represented by a red sphere.
- *Pipe* is a transition within a FSM, and represented by a green pipe with a conical point denoting direction of control flow.
- *Board* is a differential equation, represented as white text.

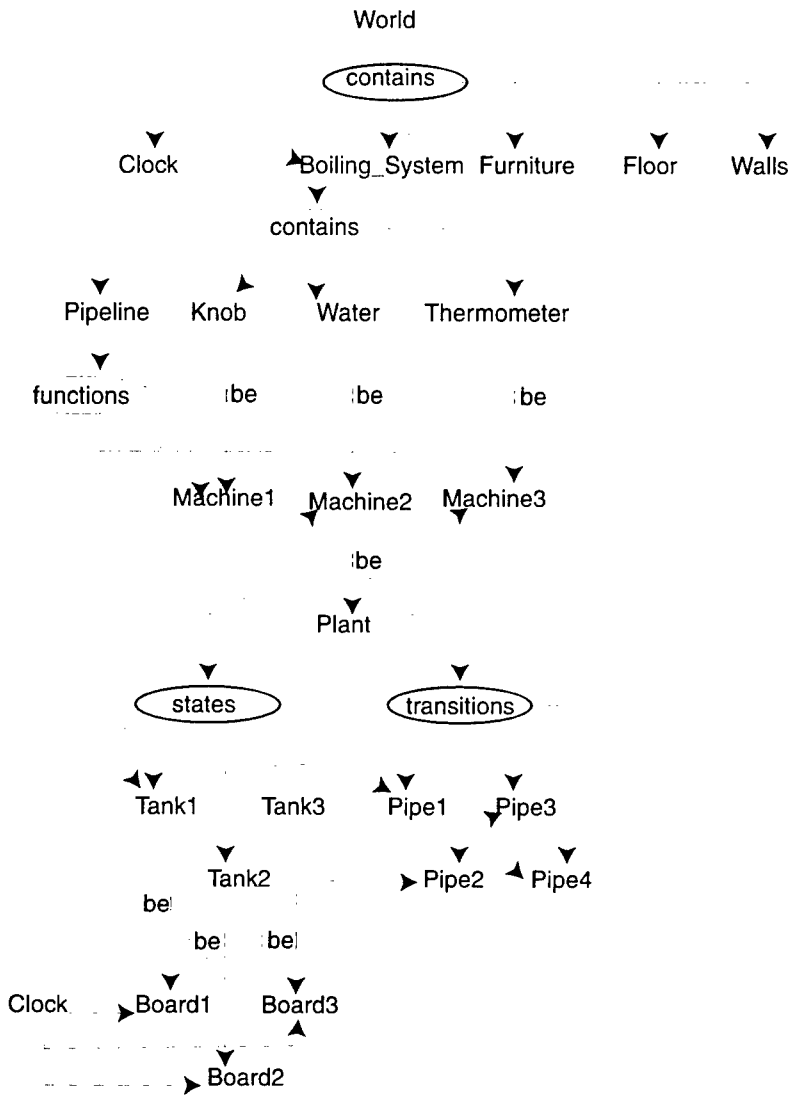


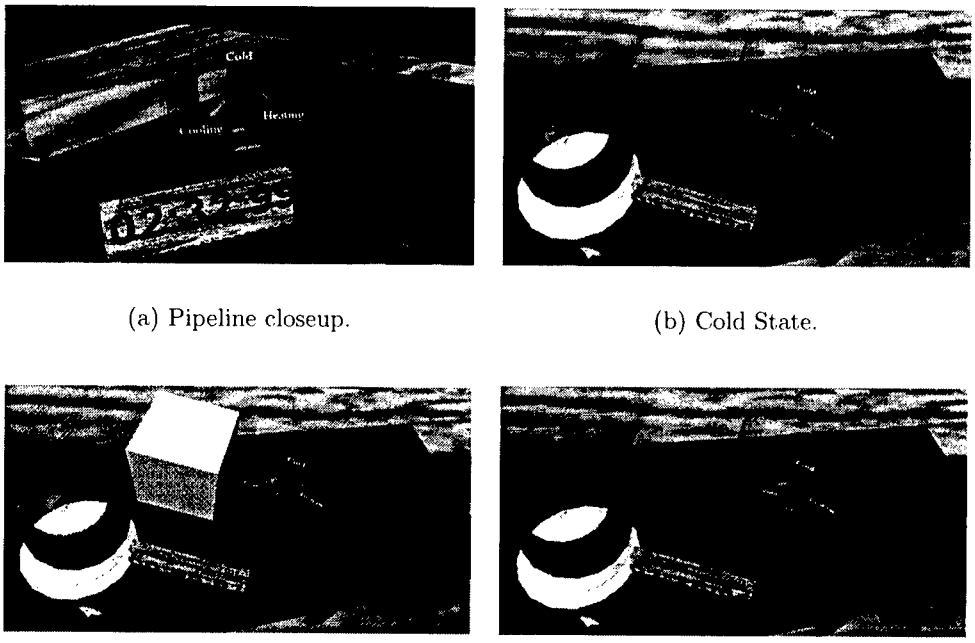
Figure 4: VRML Scene Graph for the Teapot and its models.

The following metaphors are defined in this example. The three cubes represent a sequence of machines that create a pipeline. One could have easily chosen a factory floor sequence of numerically controlled machines from the web and then used this in TeaWorld to capture the information flow. Inside the second machine, we find a plant, not unlike a petroleum plant with tanks and pipes.

The *Pipeline* and its components represent physical objects that can be acquired from the web. For our example, we show simple objects but they have been given meaningful real-world application-oriented names to enforce the view that one object models another and that we can use the web for searching and using objects for radically different purposes than their proposed original function. The overriding concern with this exercise is to permit the modeler the freedom to choose *any* object to model *any* behavior. The challenge is to choose a set of objects that provide metaphors that are meaningful to the modeler. In many cases, it is essential that more than one individual understand the metaphorical mappings and so consensus must be reached during the process. Such consensus occurs routinely in science and in modeling when new modeling paradigms evolve. The purpose of Rube is not to dictate one model type over another, but to allow the modelers freedom in creating their own model types. In this sense, Rube can be considered a meta-level modeling methodology.

The simulation of the VRML scene shown in Fig. 4 proceeds using the dashed line thread that begins with the *Clock*. The clock has an internal time sensor that controls the VRML time. The thread corresponds closely with the routing structure built for this model. It starts at *Clock* and proceeds downward through all behavioral models. Within each behavioral model, routes exist to match the topology of the model. Therefore, *Machine1* sends information to *Machine2*, which accesses a lower level of abstraction and sends its output to *Machine3*, completing the semantics for the FBM. The FSM level contains routes from each state to its outgoing transitions.

Fig. 5(a) shows a closeup view of the pipeline, that represents the dynamics of the water, beginning with the effect of the turning of the knob and ending with the thermometer that reads the water temperature. Figs. 5(b)-(d) show the pipeline during simulation when the knob is turned on and off at random times by the user. The default state is the cold state. When the knob is turned to the on position, the system moves into the heating state. When the knob is turned again back to an off position, the system moves into the cooling state and will stay there until the water reaches ambient room temperature at which time the system (through an internal state transition) returns to the cold state. Temperature change is indicated by the color of *Water* and *Machine3*, in addition to the reading on the *Thermometer* inside of *Machine3*. The material properties of *Machine1* change depending on the state of the knob. When turned off, *Machine1* is semi-transparent. When turned on, it turns opaque. Inside *Machine2*, the current state of the water is reflected by the level of intensity of each *Plant*. The current state has an increased intensity, resulting in a bright red sphere. The dynamics of temperature is indicated at two levels. At the highest level of the plant, we have a three state FSM. Within each state, we have a differential equation. The



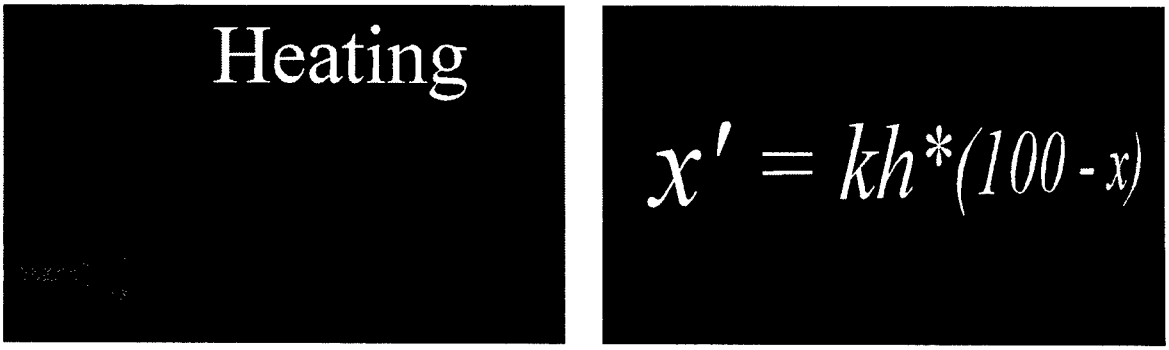
(a) Pipeline closeup.

(b) Cold State.

(c) Heating State.

(d) Cooling State.

Figure 5: The pipeline behavioral model and the behavioral FSM states defining the phase of the water.



(a) Outside of Heating phase.

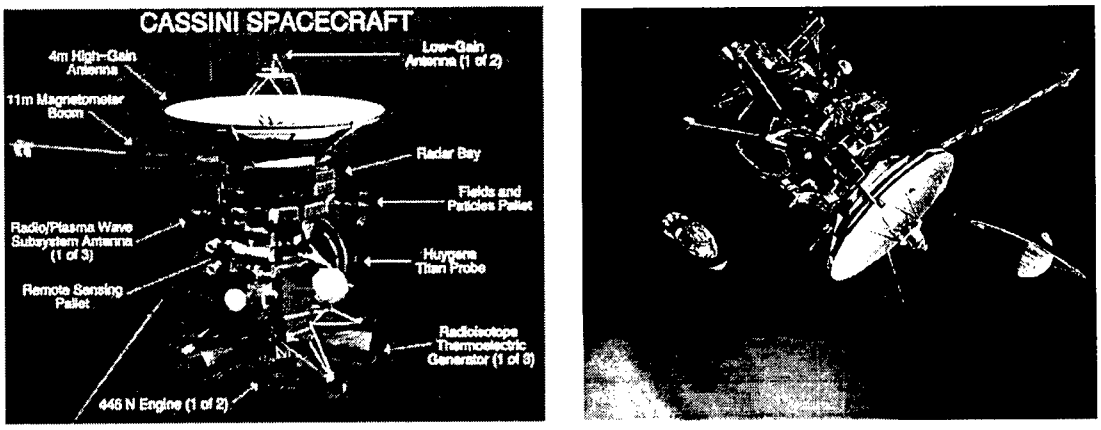
(b) Inside of Heating phase.

Figure 6: Zooming into the heating phase (Tank2).

equation is based on Newton's Law of Cooling and results in a first order exponential decay and rise that responds to the control input from the knob. The visual display of temperature change confirms this underlying dynamics since the user finds the temperature changing ever more slowly when heating to 100°C or cooling back to the ambient temperature. Figs. 6(a) and 6(b) show the outside of the heating phase (i.e., red sphere), and the inside of the phase (i.e., blackboard with the first-order differential equation).

6 Cassini

At the time of this writing (June 1999), Cassini has made a Venus flyby. It was launched in October 1997 and plans to make flybys of Venus, Earth and Jupiter on its way to Saturn. Part of the mission is to visit Titan, a moon of Saturn. Cassini, illustrated in Fig. 7(a), shows a schematic of the Cassini spacecraft while Fig. 7(b) shows an illustration of the Huygens probe separation from the Spacecraft. The probe descends through Titan's atmosphere and relays science instrument data back to the orbiter. We used the Cassini mission as a basis for a preliminary study on modeling techniques, and we decided to use an FSM dynamic model template to show three phases for the probe: 1) Separation from the spacecraft, 2) Descent, and 3) Impact. A scene was created by using an architectural metaphor for FSM states. In VRML, the user is located in a room that contains a free-floating model of Titan and Cassini. These models, as well as the model of the room, are visual, computer graphic models meant



(a) Spacecraft schematic.

(b) Release of Huygens probe.

Figure 7: Cassini mission to Saturn and Titan, Courtesy of the Jet Propulsion Laboratory.

to act as scaled-down replicas of the actual objects. Scales are non-uniform since Cassini would be much smaller with respect to Titan. The user can freely navigate this environment to view Cassini and Titan. Cassini is shown, with probe attached, making a circular orbit of the moon.

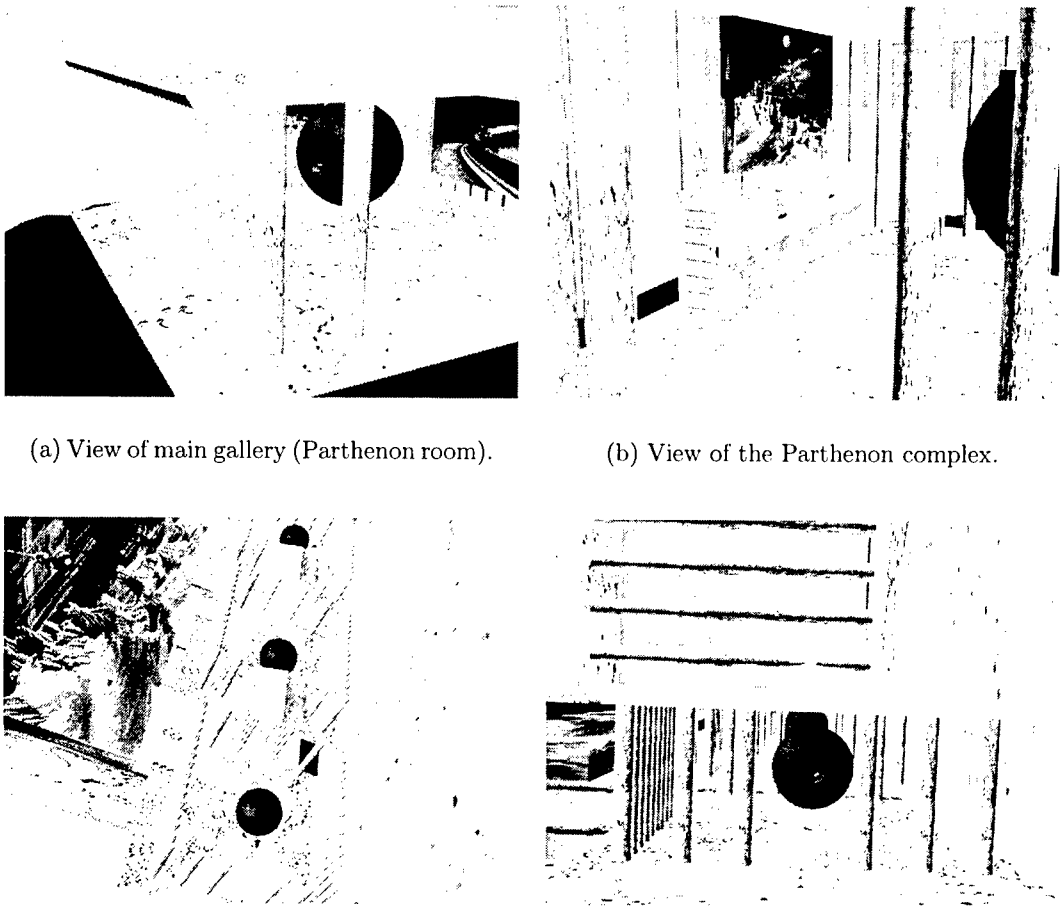
These sorts of visual, scale models are common in computer graphics but they represent a small piece of information about Cassini and its mission. Fig. 8 displays snapshots of the scene with Fig. 8(a) being the Parthenon room. On three of the four walls of this room, we find color posters relating to the mission. These posters can be clicked within the browser and the user is transported to an appropriate JPL web page identified by the poster content. Under the poster, in Fig. 8(b), we have the *Parthenon Complex*, which is an architectural metaphor for an FSM, showing the probe separation in 3 discrete phases. Fig. 8(c) shows three rooms (A, B, and C). with the following structure: $A \rightarrow B \rightarrow C$. The initial entry room and the three room environment were created from the Parthenon in Greece. This is an aesthetic aspect of this modeling practice where the modeler is free to choose any type of environment or metaphor. For Cassini, many other types of architectural metaphors come to mind, including the layout of a JPL building or the entire JPL complex (since this represents a common space well known to all JPL employees working on the Cassini project). Even within the confines of the architectural metaphor, there are an infinite number of choices. Within Room A, we may have an avatar that is positioned at the entrance to

the room (ref. Fig. 8(d)). There is also a scale model of Titan with Cassini performing the dynamics *associated with the phase* associated with Room A (i.e., probe separation from the spacecraft). Rooms B and C have similar 3D Titan models with dynamics being specified for those phases. The avatar's movement from Room $A \rightarrow B \rightarrow C$ maps directly to the dynamics of probe separation, descent and impact on Titan. The user is able to control the simulation, involving the execution of the FSM, from the main gallery or from inside the complex in Room A. Given this scenario for Cassini, there are some key issues which we should address:

- *Is it a visualization*? The work in Rube provides visualization, but models such as Cassini and Newell's Teapot demonstrate active modeling environments whose existence serves an engineering purpose and not only a post-project visualization purpose for outside visitors. This sort of modeling environment is needed from the very start of a mission—as an integral piece of the puzzle known as model design.

- *Is it economical?* Is this a lot of work just to create an FSM? Why go through the bother of creating the Parthenon, the complex and the avatar? All of these items are reused and so can be easily grabbed from the web. The concept of reuse is paramount to the Rube approach where the metaphor can be freely chosen and implemented. Without the web, Rube would not be possible. 3D object placement can be just as economical as 2D object placement, but object repositories are required not only for Cassini and Titan, but also for objects that serve to model the dynamic attributes of other objects (i.e., the Parthenon). Another economical aspect centers on the issue of computational speed for these models. Would creating a simulation in a more typical computer language would be more efficient? The structure of objects and their models within a VRML scene can be translated or compiled into native machine code as easily as source code; the 3D model structure becomes the "source code."

- *What is the advantage?* If we consider psychological factors, the 3D metaphor has significant advantages. First, 3D spatially-specific areas serve to improve our memory of the models (i.e., mnemonics). Second, graphical user interfaces (GUIs) have shown that a human's interaction with the computer is dramatically improved when the right metaphors are made available. Rube provides the environment for building metaphors. One should always be wary of mixed metaphors. We leave the ultimate decision to the user group as to which metaphors are effective. A Darwinian-style of evolution will likely determine which metaphors are useful and which are not. Aesthetics plays an important role here as well. If a modeler uses aesthetically appealing models and metaphors, the modeler will enjoy the work. It is a misconception to imagine that only the general populous will benefit from fully interactive 3D models. The engineers and scientist need this sort of immersion as well so that they can understand better what they are doing, and so that collaboration is made possible.



(a) View of main gallery (Parthenon room).

(b) View of the Parthenon complex.

(c) Removing the roof.

(d) Side view of complex.

Figure 8: Scene for Cassini and the Huygens probe dynamics.

- *Is this art or science?* The role of the Fine Arts in science needs strengthening. With fully immersive models, we find that we are in need of workers with hybrid engineering/art backgrounds. It is no longer sufficient to always think "in the abstract" about modeling. Effective modeling requires meaningful human interaction with 3D objects. So far, the thin veneer of a scale model has made its way into our engineering practices, but when the skin is peeled back, we find highly abstract codes and text. If the internals are to be made comprehensible (by anyone, most importantly the engineer), they must be surfaced into 3D using the powerful capabilities of metaphors [13, 12]. This doesn't mean that we will not have a low level code-base. Two-dimensional metaphors and code constructs can be mixed within the 3D worlds, just as we find them in our everyday environments with the embedding of signs. At the University of Florida, we have started a *Digital Arts and Sciences* Program with the aim to produce engineers with a more integrated background. This background will help in the production of new workers with creative modeling backgrounds.

7 Key Architectural Benefits of Rube

The following are novel features of Rube and represent reasons for choosing elements of this architecture:

- *An Integrated Environment*: There is no difference between objects modeling other objects and objects acting in their traditional roles. The modeling and object environments are identical. A pipe can be used in a petro-chemical factory or in a Petri net. Model components are chosen from the vast universe of VRML objects on the web. Components in models are dynamic as for any object. Models need not be static.

- *Modeling Freedom*: Any 2D or 3D package can be used to create models. There is no need for the Rube team to build a GUI for each model type; the model author can freely choose among drawing and modeling packages.

- *Model Design Flexibility*: There is no predefined modeling method. If a set of objects is to be interpreted as a model then one adds a small amount of "role playing" information to the objects. Any number of model types can be supported. A side-effect of this flexibility is the provision of natural *multimodeling* support.

- *VRML encapsulation*: VRML worlds can be stored anywhere over the web and positioned within an author's world through a URL. No new standards have been created outside of existing web standards and so Rube is built within the framework of VRML, but we can find expressive distributed modeling and simulation capability by "piggybacking" on the capabilities of the standard. The VRML file that contains prototypes with model fields is a *digital object*, the digital equivalent of the corresponding physical

object with all of its attributes. This encapsulation is possible due to the flexible syntax and architecture of VRML (i.e., with key nodes such as PROTO, EXTERNPROTO, Anchor nodes and Sensors being essential for the inclusion of modeling information). The average 3D file standard would leave little room for the definition of models. We propose our modeling methodology as a method for model construction with VRML. In the VMRL community, this has the potential to alter, for example, how behavior of objects are modeled. Java and selected behavior scripting languages are currently used, whereas Rube offers the capability for some of this behavior to be modeled and translated into Java using VRML, itself, to define behavior.

8 Reflections on the Art of Modeling

It is sometimes difficult to differentiate models used for the creation of pieces of art from those used with scientific purposes in mind. Models used for science are predicated on the notion that the modeling relation is unambiguously specified and made openly available to other scientists. Modeling communities generally form and evolve while stressing their metaphors. In a very general sense, natural languages have a similar evolution. The purpose of art, on the other hand, is to permit some ambiguity with the hopes of causing the viewer or listener to reflect upon the modeled world. Some of the components in worlds such as Fig. 3 could be considered non-essential modeling elements that serve to confuse the scientist. However, these elements may contribute to a more pleasing immersive environment. Should they be removed or should we add additional elements to please the eye of the beholder? In Rube, we have the freedom to go in both directions, and it isn't clear which inclusions or eliminations are appropriate since it is entirely up to the modeler or a larger modeling community. One can build an entirely two dimensional world on a blackboard using box and text objects, although this would not be in the spirit of creating immersive worlds that allow perusal of objects and their models.

It may be that a select number of modelers may find the TeaWorld room exciting and pleasing, and so is this pleasure counterproductive to the scientist or should the scientist be concerned only with the bare essentials necessary for unambiguous representation and communication? Visual models do not represent *syntactic sugar* (a term common in the Computer Science community). Instead, these models and their metaphors are essential for human understanding and comprehension. If this comprehension is complemented with a feeling of excitement about modeling, this can only be for the better. Taken to the extreme, a purely artistic piece may be one that is so couched in metaphor that the roles played by objects isn't clear. We can, therefore, imagine a kind of continuum from a completely unambiguous representation and one where the roles are not published. Between these two extremes, there is a lot of breathing space. Science can be seen as a form of consensual art where everyone tells each other what one object *means*. Agreement ensues within a community and then there is a mass convergence towards one metaphor in favor of another.

We are not proposing a modification to the VRML standard although we have found that poor authoring support currently exists in VRML editors for PROTO node creation and editing. We are suggesting a different and more more general mindset for VMRL—that it be used not only for representing the shape of objects, but all modeling information about objects. VRML should be about the complete digital object representation and not only the representation of geometry with low-level script behaviors to support animation. Fortunately, VRML contains an adequate number of features that makes this new mindset possible even though it may not be practiced on a wide scale. While a VRML file serves as the digital object, a model compiler is also required for the proper interpretation of VRML objects as models.

9 Summary

There is no unified modeling methodology, nor should there be one. Instead, modelers should be free to use and construct their own worlds that have special meaning to an individual or group. With Rube, we hope to foster that creativity without limiting a user to one or more specific metaphors. Rube has a strong tie to the World Wide Web (WWW). The web has introduced a remarkable transformation in every area of business, industry, science and engineering. It offers a way of sharing and presenting multimedia information to a world-wide set of interactive participants. Therefore any technology tied to the web's development is likely to change modeling and simulation. The tremendous interest in Java for doing simulation has taken a firm hold within the simulation field. Apart from being a good programming language, its future is intrinsically bound to the coding and interaction within a browser. VRML, and its X3D successor, represent the future of 3D immersive environments on the web. We feel that by building a modeling environment in VRML and by couching this environment within standard VRML content, that we will create a "trojan horse" for simulation modeling that allows modelers to create, share and reuse VRML files.

Our modeling approach takes a substantial departure from existing approaches in that the modeling environment and the object environment are merged seamlessly into a single environment. There isn't a difference between a circle and a house, or a sphere and a teapot. Furthermore, objects can take on any role, liberating the modeler to choose whatever metaphor that can be agreed upon by a certain community. There is no single syntax or structure for modeling. Modeling is both an art and a science; the realization that all objects can play roles takes us back to childhood. We are building Rube in the hope that by making all objects virtual that we can return to free-form modeling of every kind. Modeling in 3D can be cumbersome and can take considerable patience due to the inherent user-interface problems when working in 3D using a 2D screen interface. A short term solution to this problem is to develop a model package that is geared specifically to using one or more metaphors, making the insertion of, say, the Parthenon complex rooms a drag and drop operation. Currently, a general purpose modeling package must be used carefully

position all objects in their respective locations. A longer term solution can be found in the community of virtual interfaces. A good immersive interface will make 3D object positioning and connections a much easier task than it is today.

There are many unanswered questions concerning the Rube architecture and the affect it may have on the vast community of model authors. For example, many communities have their own internal standards for behavior representation. VHDL (Very High Level Hardware Description Language) is one such community. They have expended vast resources into the use of VHDL. Should they switch to VRML or is there a way that the two standards can relate to one another? We feel that conversion techniques between the VRML file and the other file-based standards will ameliorate the potentially harsh conditions associated with a migration of standards. Some standards such as HLA (High Level Architecture) do not include a direct provision for model specification since HLA is focused on the execution of distributed simulators and simulations regardless of how they were created and from what models they were translated. In such cases, Rube will provide a complementary technology to aid in the modeling process. UML (Unified Modeling Language) unifies select visual object-oriented formalisms for representing models of software. There is no reason why someone cannot build a complete 2D representation using a 2D modeler such as CorelDraw or AutoCAD and then construct a grammar to produce the necessary target language code segments needed for UML model execution. Therefore, Rube is a more general procedure for model translation than that provided by most metaphor-fixed visual formalisms. In this sense, the following analogy holds: Rube is to Modeling-Language-X as Yacc is to Computer-Language-Y. Rube is a general purpose model creation facility and Yacc is a compiler-compiler used to create compilers for arbitrary computer language grammars.

We will continue our research by adding to Rube and extending it to be robust. In particular, we plan on looking more closely into the problem of taking legacy code and making it available within the VRML model. This is probably best accomplished through TCP/IP and a network approach where the Java/Javascript communicates to the legacy code as a separate entity. We plan on extending the VRML parser, currently used to create the model browser, so that it can parse a 3D scene and generate the Java required for the VRML file to execute its simulation. Presently, the user must create all Script nodes. The model browser will be extended to permit various modes of locating models within objects. A "fly through" mode will take a VRML file, with all object and model prototypes, and place the models physically inside each object that it references. This new generated VRML file is then browsed in the usual fashion. Multiple scenes can be automatically generated.

Acknowledgments

My first thanks go to my students. They are making Rube and the 'virtual sandbox' come alive through their hard work and inventive ideas and solutions. In particular, I would like to

thank Kangsun Lee, Robert Cubert, Andrew Reddish, Tu Lam, and John Hopkins. I would like to thank the following agencies that have contributed towards our study of modeling and simulation, with a special thanks to the Jet Propulsion Laboratory where I visited for three weeks during June 1998: (1) Jet Propulsion Laboratory under contract 961427 *An Assessment and Design Recommendation for Object-Oriented Physical System Modeling at JPL* (John Peterson, Stephen Wall and Bill McLaughlin); (2) Rome Laboratory, Griffiss Air Force Base under contract F30602-98-C-0269 *A Web-Based Model Repository for Reusing and Sharing Physical Object Components* (Al Sisti and Steve Farr); and (3) Department of the Interior under grant 14-45-0009-1544-154 *Modeling Approaches & Empirical Studies Supporting ATLSS for the Everglades* (Don DeAngelis and Ronnie Best). We are grateful for their continued financial support.

References

[1] Ian G. Barbour. *Myths, Models and Paradigms.* Harper and Row Publishers, 1974.

[2] Grady Booch. *Object Oriented Design.* Benjamin Cummings, 1991.

[3] Rikk Carey and Gavin Bell. *The Annotated VRML 2.0 Reference Manual.* Addison-Wesley, 1997.

[4] National Research Council. Advanced Engineering Environments: Achieving the Vision, Phase I, 1999. http://www.nap.edu/catalog/.

[5] Robert M. Cubert and Paul A. Fishwick. MOOSE: An Object-Oriented Multimodeling and Simulation Application Framework. *Simulation*, 70(6):379–395, June 1998.

[6] Paul A. Fishwick. Simpack: Getting Started with Simulation Programming in C and C++. In *1992 Winter Simulation Conference*, pages 154–162, Arlington, VA, December 1992.

[7] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall, 1995.

[8] Paul A. Fishwick and Charles L. Blackburn. Managing Engineering Data Bases: The Relational Approach. *Computers in Mechanical Engineering (CIME)*, pages 8–16, January 1983.

[9] Paul A. Fishwick, Thomas R. Sutter, and Charles L. Blackburn. Prototype Integrated Design (PRIDE) System: Reference Manual, Volume 2: Schema Definition. Technical Report 172183, NASA, July 1983. NASA Contractor Report, Contract NAS1-16000.

[10] Daniel S. Goldin, Samuel L. Venneri, and Ahmed K. Noor. Beyond Incremental Change. *IEEE Computer*, 31(10):31–39, October 1998.

[11] Mary Hesse. *Models and Analogy in Science.* University of Notre Dame Press, 1966.

[12] George Lakoff. *Women, Fire and Dangerous Things: what categories reveal about the mind.* University of Chicago Press, 1987.

[13] George Lakoff and Mark Johnson. *Metaphors We Live By.* University of Chicago Press, 1980.

[14] Kangsun Lee and Paul A. Fishwick. A Methodology for Dynamic Model Abstraction. *SCS Transactions on Simulation*, 1996. Submitted August 1996.

[15] John B. Malone. Intelligent Synthesis Environment: Engineering Design in the 21st Century. Slide Presentation.

[16] Peter C. Marzio. *Rube Goldberg, His Life and Work.* Harper and Row, New York, 1973.

[17] Pierre-Alain Muller. *Instant UML.* Wrox Press, Ltd., Olton, Birmingham, England, 1997.

[18] Winfried Noth. *Handbook of Semiotics.* Indiana University Press, 1990.

[19] James Rumbaugh, Michael Blaha, William Premerlani, Eddy Frederick, and William Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[20] David B. Smith and L. Koenig. Modeling and Project Development. In *Fifth International Workshop on Simulation for European Space Programmes - SESP '98*, November 1998.

[21] S. Wall. Reinventing the Design Process: Teams and Models. In *International Astronautical Federation Specialist Symposium on Novel Concepts for Smaller, Faster and Better Space Missions*, Redondo Beach, CA, April 1999.

[22] S. D. Wall, J. C. Baker, J. A. Krajewski, and D. B. Smith. Implementation of System Requirements Models for Space Missions. In *Eighth Annual International Symposium of the International Council on Systems Engineering*, July 26-30 1998.

Author Biography

Paul A. Fishwick is Professor of Computer and Information Science and Engineering at the University of Florida. He received the BS in Mathematics from the Pennsylvania State University, MS in Applied Science from the College of William and Mary, and PhD in Computer and Information Science from the University of Pennsylvania in 1986. He also has six years of industrial/government production and research experience working at Newport News Shipbuilding and Dry Dock Co. (doing CAD/CAM parts definition research) and

at NASA Langley Research Center (studying engineering data base models for structural engineering). His research interests are in computer simulation, modeling, and animation. Dr. Fishwick is a Fellow of the Society for Computer Simulation (SCS), and a Senior Member of the IEEE Society for Systems, Man and Cybernetics. Dr. Fishwick founded the `comp.simulation` Internet news group (Simulation Digest) in 1987, which now serves over 15,000 subscribers. He has chaired workshops and conferences in the area of computer simulation, and will serve as General Chair of the 2000 Winter Simulation Conference. He was chairman of the IEEE Computer Society technical committee on simulation (TCSIM) for two years (1988-1990) and he is on the editorial boards of several journals including the *ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics, The Transactions of the Society for Computer Simulation, International Journal of Computer Simulation,* and the *Journal of Systems Engineering.* He has published over 100 refereed articles, authored one textbook *Simulation Model Design and Execution: Building Digital Worlds* (Prentice Hall, 1995), and co-edited three books.

Digital Object Multimodel Simulation Formalism and Architecture

R. M. Cubert and P. A. Fishwick

Department of Computer & Information Science and Engineering
University of Florida; CSE Building Room E301; Gainesville, FL 32611-6120 USA

ABSTRACT

The object-oriented approach known as heterogeneous behavior multimodeling has been developed, used, and reported elsewhere, to facilitate creation, modification, sharing, and reuse of object-oriented models and the simulations created from those models. The digital object extends multimodeling so that digital objects can be shared and combined in ways that ordinary multimodels cannot. We describe an abstract base multimodel and several derived instantiated multimodel types. We also describe a transformation which takes a digital object to a simulation program. We give formal definitions of multimodeling, digital object, and the transformation, then from these definitions prove correctness of execution sequencing of simulations created by applying the transformation to digital objects. Closure under coupling of digital objects follows as a corollary, subject to an assumption regarding experimental frame. We then construct an abstract base architecture for manufacture, flow, and persistence of digital objects. From the base architecture we derive and instantiate a suite of architectures, each targeted at a distinct set of requirements: one to operate locally, another with internet protocols, a third with web protocols, and a fourth to allow digital objects to interoperate with other kinds of simulations.

Keywords: Simulation, Object-Oriented Modeling, Model Abstraction, Heterogeneous behavior multimodel, Digital Object

1. INTRODUCTION

The concept of digital object extends heterogeneous behavior multimodeling. Heterogeneous behavior multimodeling has been developed, used, and reported elsewhere, to facilitate creation, development, modification, and reuse of object-oriented models and the simulations created from those models. Although Object Oriented Physical Modeling (OOPM) and its implementation of behavior multimodels provide an ability to manage complex patterns of behavioral abstraction in simulation modeling, heterogeneous multimodeling has suffered from lack of underlying formalism; and while multimodels could be readily refined with additional detail (downwards), they could not be readily combined with other multimodels (upwards), thus limiting their potential for sharing, reuse, and participation in a model repository. Digital objects overcome these limitations. In what we here term the "base paper"[1] the authors reported results of some work extending multimodeling at AeroSense 1999. In reporting new work during the past year, the present paper in effect extends and refines the base paper: work reported in the base paper is restated quite briefly and only as required to preserve logical continuity; accordingly, the present paper relies heavily on the reader's familiarity with the base paper. The base paper defines several types of multimodels and how to build digital objects as multimodels; it explores interfaces which conduct information between the outside world and a digital object, and traces which conduct information within a digital object; it shows how arrangement of what we term "constituents" of a multimodel imposes a partial order on execution sequence; and, it discusses heterogeneous multimodel hierarchy. It goes on to show how to transform a multimodel to a simulation; examines temporal and logical behavior sequences; elucidates a problem which we term "inversion", and shows how to resolve this problem without affecting correctness; it provides algorithms for transforming a digital object multimodel to a simulation program in such a way that the behavior sequence of such simulations is correct by construction; and, proves the correctness of the transformation as regards behavior sequence. It also provides the comprehensive "Teapot" example.

In Sect. 2, we define digital object including the input-output information tuple, and the "DOT" transform which takes a digital object to a multimodel; we present an abstract base multimodel and a modeling grammar, and discuss coupling and closure under coupling. In Sect. 3 we present a transformation from multimodel to simulation

Author E-mail information: rmc@cise.ufl.edu, fishwick@cise.ufl.edu

In *Enabling Technology for Simulation Science IV*, Alex F. Sisti, Editor,
Proceedings of SPIE Vol. 4026 (2000) • 0277-786X/00/$15.00

program, including a way to rearrange constituents, and explanation of the inversion elimination algorithm. In Sect. 4 we demonstrate the correctness of the transformation of Sect. 3, with additional definitions and a theorem, a special case in scheduling, and showing closure of multimodeling under coupling subject to an assumption regarding experimental frame. In Sect. 5 we report an abstract base architecture for manufacture, flow, and persistence of digital objects. From the base architecture we derive and instantiate four configurations, each targeted at a distinct set of requirements: one to operate locally, another with internet protocols, a third with web protocols, and a fourth to allow digital objects to interoperate with other kinds of simulations. Sect. 6 is the conclusion.

2. DIGITAL OBJECT

DEFINITION 2.1 (ABSTRACTION). *A human activity consisting of focusing on some aspects of a system, while ignoring other aspects which are not germane to a particular context or objective*[2]. *Related to two senses of the verb* abstract*: to consider apart from application to a particular instance; and, to summarize*[3-5].

DEFINITION 2.2 (OBJECT). *That which can be circumscribed as a distinct unit of existence and/or abstraction; having state, behavior, and identity; and which can be observed, manipulated, and/or affected.*

DEFINITION 2.3 (MODEL). *A system, either constructed or discovered, which, in a particular context, is a metaphor for another system and serves as surrogate for that other system.*

DEFINITION 2.4 (CORRESPONDENCE). *A metaphor, expressed as a pairwise relation: suppose system S1 is to be modeled, that object O1 is identified as a germane part of S1, and that there exists in the universe some object O2. The ordered pair* $C_i = \{O1, O2\}$ *denotes that in a model of S1, O2 stands for or represents O1.* C_i *is a correspondence. A model has a set* $\{C_i\}$ *of such correspondence relations.*

DEFINITION 2.5 (MODELING). *A human activity which uses abstraction and (usually numerous instances of) correspondence, to express an overall representation (the model) based on an overall metaphor in the mind of the author. The model serves as surrogate for a system, for the purpose of managing complexity and/or attaining insight.*

DEFINITION 2.6 (ATOM). *Typically, text representing source code in a programming language.*

DEFINITION 2.7 (HETEROGENEOUS MULTIMODEL). *A model represented as a graph*[6] *in which each vertex and each edge may be an atom or a multimodel of any of various types, combined arbitrarily irrespective of type. Multimodel types include Finite State Machines (FSM), Functional Block Models (FBM), Rule Based Models (RBM), and System Dynamics Models (SDM).*

DEFINITION 2.8 (CONSTITUENT). *A vertex or edge of a multimodel graph which exhibit behavior, and which can be refined into more multimodels. Constituents are: in an FSM, states* Q *and transitions* Δ*; in an FBM, blocks* $\mathfrak{B}$*; in an RBM, premises* Ψ *and consequences* K*; and, in an SDM, auxiliaries* $\mathcal{A}$.

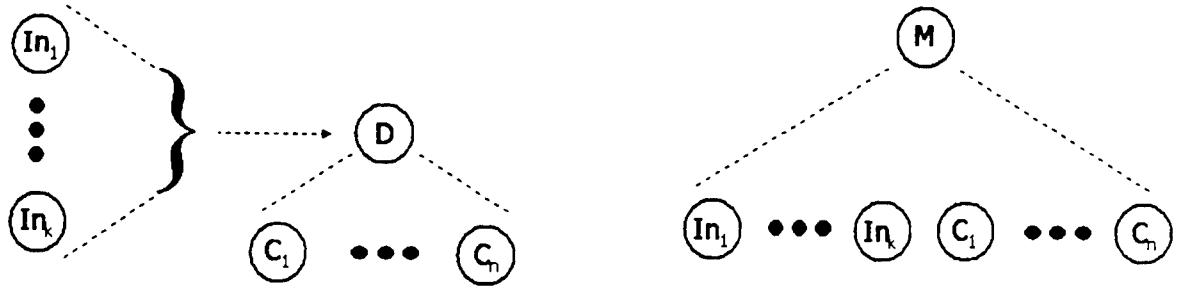
DEFINITION 2.9 (INPUT-OUTPUT INFORMATION TUPLE). *An* ***input-output information tuple*** $\hat{I}$ *is a tuple* $\{In, InOut, Out, R\}$*, where In is an ordered set of inputs, InOut is an ordered set of items which are both input and output, Out is an ordered set of outputs, and R is a return type. Each element of In and of InOut is an ordered triple* $\{T, N, V\}$*, and each element of Out is an ordered pair* $\{T, N\}$*, where T is a parameter type, N is a parameter name, and V is a(n optional) behavior returning a value. A parameter type may be a native type such as integer, real, or string; or, any abstract type. A parameter name is typically a character string.* $\hat{I}$ *ensures uniform treatment of inputs and outputs across all multimodel types and atoms.*

DEFINITION 2.10 (DIGITAL OBJECT INTERFACE). *A* ***digital object interface*** I *is a tuple* $\{N, H, G, \hat{I}, \mathcal{E}\}$*, where* N *is a name,* H *is descriptive hypertext URL,* G *is a graphical icon URL, and* $\mathcal{E}$ *is a behavior element.* N *is typically a character string, and all elements except* N *are optional. When a behavior element is called, the actual parameters in the call correspond to the formal parameter lists (In followed by InOut followed by Out) of Iotuple, except that an actual input parameter can be omitted if the corresponding formal parameter has a value* V*, in which case* V *is the default value.*

DEFINITION 2.11 (DIGITAL OBJECT). ***Digital object*** *extends heterogeneous multimodel with interface* I*. A digital object is a composition of subordinate multimodels and atoms. When arranged as a tree whose nonterminal vertices are multimodels and whose leaves are atoms, its hierarchy corresponds to levels of abstraction,*[3] *and any number of such levels is permissible.*

2.1. The DOT Transform

Digital objects are imported using the "DOT Transform," named after our Digital Object Tool which performs this transformation. Digital object $\mathcal{D}$ shown in Fig. 1 part (a) is to be imported into some larger digital object multimodel (not shown). Behavior elements of the inputs in its Interface are shown as $In_1 \cdots In_k$. Constituents of the multimodel at the root of D are shown as $C_1 \cdots C_n$. The effect of the transformation is shown in part (b), as multimodel $\mathcal{M}$ with $n + k$ constituents. This figure is the underlying *mechanism* which supports a capability to compose digital objects by drag-and-drop.



(a) dot-3a (b) dot-3b

Figure 1. DOT Transform.

When we import a digital object we make its multimodel an integral part of the overall multimodel representing the larger digital object. To do so, inputs (*Ins* and *InOuts*) of its interface must be resolved. Each input has a behavior element $\mathcal{E}$ which is internal to the digital object. This behavior element is used as the default. However, if something from outside is connected to this input, then that "something" is used instead. Importing digital object $\mathcal{D}$ whose interface has k inputs and whose root multimodel has n constituents involves the conversion shown in Fig. 1 into a multimodel with $n + k$ constituents.

2.2. Multimodel Types

DEFINITION 2.12 (TRACE). *A path conducting an object or a value of native type from a source to a destination. Let $\mathcal{M}$ be a multimodel with an interface including $In_s \in In \cup Inout$ and $O_d \in InOut \cup Out$. Let $\{C_i\}$ be the constituents of $\mathcal{M}$, with each C_i having an interface including $In_{i,a} \in$ the i^{th} element of $In \cup InOut$ and $O_{i,b} \in$ the i^{th} element of $InOut \cup Out$. Let $\aleph$ denote the attributes of $\mathcal{M}$. There are seven kinds of traces: (1) from In_s of $\mathcal{M}$ to $In_{d,a}$ of C_d; (2) from In_s of $\mathcal{M}$ to $\alpha \in \aleph$; (3) from $O_{s,b}$ of C_s to O_d of $\mathcal{M}$; (4) from $\alpha \in \aleph$ to O_d of $\mathcal{M}$; (5) from $\alpha \in \aleph$ to $In_{d,a}$ of C_d; (6) from $O_{s,b}$ of C_s to $\alpha \in \aleph$; and, (7) from $O_{s,b}$ of C_s to $In_{d,a}$ of C_d. However, if $\mathcal{M}$ is an atom, it has no constituents and its traces can only be of types (2) and (4), conducting data between the outside world and the atom.*

Abstract base multimodel is an abstract base class, embodying characteristics common to all multimodel types; we instantiate some particular subtypes of it; it also forms a basis for extensibility to future new subtypes. A base multimodel is a 3-tuple $\{\hat{I}, \aleph, T\}$. $\hat{I}$ is an input-output information tuple (see Definition 2.9). $\aleph$ is a set of attributes which may be of native type or abstract type, and which retain information in the multimodel. T is a set of traces. Multimodel types subsumed by this theory appear in Definition 2.7 and below. These (among others) are presented by Fishwick[6] and examples appear in the base paper. Denote the set of multimodel types as $\mathfrak{M}$ = {FSM, FBM, RBM, SDM}. For multimodel $\mathcal{M}$, let $\mathcal{MT}(\mathcal{M})$ denote its multimodel type. Then $\mathcal{MT}(\mathcal{M}) \in \mathfrak{M}$. Each multimodel type derived from the base multimodel has a distinct set of constituents.

Finite State Machine (FSM) is a tuple $\{I, Q, \Delta, q_0, q_c, \aleph, T\}$, where Q is an ordered set of states, Δ is a set of transitions, $q_0 \in Q$ is the start state, and $q_c \in Q$ is the current state. A transition $\delta_{i,j}$ joins the two states q_i and q_j. and is directed from q_i to q_j. Behavior of a finite state machine is the behavior of q_c followed by the behavior of some or all eligible transitions. Eligible transitions are those which originate at q_c, *i.e.*:, $\{\delta_{c,j}\ \forall j\}$. The behavior of $\delta_{c,j}$ is to decide whether to change q_c to q_j. Eligible transitions are explored in an ordering corresponding to the ordering of Q, until one of them decides to change q_c, or until no eligible transitions remain to be explored. Let $p_{i,j}$ denote

the (boolean-valued) predicate associated with the decision made by $\delta_{i,j}$. In a well-formed finite state machine, at most one predicate among those for transitions originating at q_c, that is of the set $\{p_{c,j}\ \forall\ j\}$, is true at one particular time. If all such predicates are false, the result is the equivalent of a reflexive transition associated with a predicate which is the complement of the disjunction of the other predicates.

Functional Block Model (FBM) is a tuple $\{I, B, \aleph, T\}$, where B is an ordered set of blocks. Behavior of a functional block model is the behaviors of all its blocks, in sequence, corresponding to the ordering of B.

Rule-Based Model (RBM) is a tuple $\{I, R, \aleph, T\}$, where R is $\{R_i,\ 1 \leq i \leq n\}$, where each R_i is the tuple $\{\psi_i, A_i, \kappa_i\}$, consisting of ψ_i a premise (lowercase Greek letter psi), A_i an arc, and κ_i (lowercase Greek letter kappa) a consequence. Behavior of a rule-based model is the behavior of all premises followed by the behaviors of none, some, or all, of the consequences. When there are multiple premises the sequence of premise behavior corresponds to the ordering of R. The behavior of each premise is to decide whether to excite the behavior of the associated consequence to occur. An arc conducts a decision from ψ_i to κ_i which excites κ_i to perform its behavior. When multiple consequences occur, they occur in a sequence corresponding to the ordering of R.

System Dynamics Model (SDM) is a tuple $\{I, S, L, R, A, F, C, \aleph, T\}$, where S is a set of sources, L is a set of levels, R is a set of rates, A is an ordered set of auxiliaries, F is a set of flow arcs, and C is a set of causal arcs. The metaphor for an SDM is a hydraulic system, with inlets (S), fluid reservoirs (L), and valves (R) which effect flow rates. Arcs are of two kinds: flow arcs and causal arcs. Flow arcs in a system dynamics model resemble pipes which carry fluid from sources, through valves to reservoirs, and from reservoirs through valves. Just as electrical wires carry control signals to solenoid valves in a hydraulic system, causal arcs in a system dynamics model carry information (typically from levels and auxiliaries) to control flow rates. A rate is specified to within a multiplicative factor by a causal arc connecting either a level or a source to the rate. A causal arc from an auxiliary to a rate provides the multiplicative factor. An auxiliary can be a constant, an attribute value, or a constituent. Behavior of a system dynamics model is the behavior of its constituent auxiliaries in a sequence corresponding to the ordering of A, followed by behavior of a set of difference equations derived from the way arcs relate rates, levels, and auxiliaries. In these difference equations, the state variables are levels (L) of the system dynamics model.

2.3. Constituent Partial Order and Hierarchy

In a simulation corresponding to a multimodel, it is necessary to order the occurrence of behavior of constituents to ensure conformance to the proper partial ordering imposed by dependencies associated with traces. Ordering can usually be determined by a topological sort based on traces and the nature of the constituents, carried out, in most cases, mechanically without human intervention. A constituent has a list of predecessors. Partial order is represented as a set of such lists. C_a precedes C_b in the partial order if and only if $C_a \in predecessors(C_b)$. Zeigler[7] at pages 108-112 discusses ways to determine partial order, as does Fishwick[6] at page 150. Our approach generates a correct sequence by choosing a total order from the partial order, but which one is chosen is of no consequence to simulation correctness.

DEFINITION 2.13 ($\Re$). *The set of all constituents which do not change the value of any attribute; typically from Δ and Ψ; similar to accessor methods. Correctness of behavior sequencing is unaffected by certain rearrangements of the order of occurrence of constituents which are* $\in \Re$.

We represent behavior of a digital object as a hierarchy, in particular a tree $\mathcal{T}$. We partition the set of all such trees into three classes: those of height 1 ($\{\mathcal{T}_1\}$), those of height 2 ($\{\mathcal{T}_2\}$), and all others ($\{\mathcal{T}_m\}$). $\mathcal{T}_1$ is of degree 0 and has an atom as its root. Standing alone, $\mathcal{T}_1$ represents a digital object written as monolithic source code, thus subsuming legacy code under our theory. $\mathcal{T}_1$ can occur within $\mathcal{T}_2$ or $\mathcal{T}_m$ as a leaf. $\mathcal{T}_2$ is of degree n and has root $\mathcal{R}$. $\mathcal{MT}(\mathcal{R}) \in \mathfrak{M}$, and $\mathcal{R}$ has n children. $\mathcal{R}$ represents a multimodel; each child of $\mathcal{R}$ is an atom which represents a constituent of that multimodel. Children of $\mathcal{R}$ are arranged, from left to right, in an order resulting from a sort whose primary key is the order of appearance of each constituent type in the definition of constituents above; and, whose secondary key is the position of each element within its ordered set. Thus in an FSM, states followed by transitions, $q_1, \ldots, q_n, \delta_{(1,2)}, \ldots, \delta_{(n-1,n)}$; in an FBM, blocks, $b_1, \ldots, b_n$; in an RBM, premises followed by consequences, $\psi_1, \ldots, \psi_n, \kappa_1, \ldots, \kappa_n$; and, in an SDM, auxiliaries, $a_1, \ldots, a_n$. $\mathcal{T}_m$ has properties similar to those of $\mathcal{T}_2$, and also has has refinement of constituents, and a mix of multimodels and atoms as constituents. Let generalized type $\mathfrak{G} = \mathfrak{M} \cup \{atom\}$. Let $\{\mathcal{C}_i\}$ be the set of all vertices in $\mathcal{T}_m$ except $\mathcal{R}$. Every $\mathcal{C}_i$ is a constituent of some multimodel, and $\mathcal{MT}(\mathcal{C}_i) \in \mathfrak{G}$.

Multimodel hierarchy tree $\mathcal{T}$ satisfies two **heterogeneity** requirements. Let $\mathcal{M}_0$ be a multimodel vertex $\in \mathcal{T}$, and let $\mathcal{M}_0$ have as its constituents multimodel vertices $\mathcal{M}_1, \ldots, \mathcal{M}_n$. We know that $\mathcal{MT}(\mathcal{M}_i) \in \mathfrak{M}, \forall i \in \{0,\ldots,n\}$. The first heterogeneity requirement is: the $\mathcal{MT}(\mathcal{M}_i)\ \forall i \in \{0,\ldots,n\}$ may all be chosen independently, as $n+1$ arbitrary choices. Let $\mathcal{M}$ be a multimodel vertex $\in \mathcal{T}$; and let the constituents of $\mathcal{M}$ be vertices $\mathcal{C}_1,\ldots,\mathcal{C}_n$. We know that $\mathcal{GT}(\mathcal{C}_i) \in \mathfrak{G}\ \forall i \in \{1,\ldots,n\}$. The second heterogeneity requirement is: $\mathcal{GT}(\mathcal{C}_i)\ \forall i \in \{1,\ldots,n\}$ may all be chosen independently of one another and independently of $\mathcal{MT}(\mathcal{M})$, as n arbitrary choices.

2.4. Coupling

Coupling is the joining of multimodels and constituents in a hierarchy. The two kinds of coupling are **intralevel** and **interlevel**. Given a set of behavior elements which are sibling constituents of a behavior multimodel, **intralevel coupling** is the mechanism for relating the sibling behavior elements to one another. Intralevel coupling is specified by six rules: (1) topology is that of a well-formed directed graph; (2) constituents are partitioned into sets, with elements of each set subject to rules governing behavior of that set; (3) like items are uniquely ordered; (4) corresponding items are placed into unique correspondence; (5) traces specify data dependencies; and, (6) partial ordering is determined by the traces. Given a digital object or any behavior multimodel hierarchy, **interlevel coupling** is the mechanism for relating a behavior multimodel or atom at one level to behavior multimodels or atoms at the levels above and below. Interlevel coupling is specified by three rules: (1) hierarchy, (2) heterogeneity, and, (3) adherence to multimodel semantics specified in Sect. 2.2. Our theory must exhibit closure under coupling, to permit us to freely combine models without concern for their various multimodel types or details of their internal structure, and to mix and match digital objects. Zeigler[8] at page 60 states:

> "A formalism is said to be *closed under composition* if any composite system obtained by coupling components specified by the formalism is itself specified by the formalism. ... The significance of such closure is that it facilitates hierarchical construction of models by recursive applications of the coupling procedure."

Although Zeigler's work does not contemplate heterogeneity in the composition he mentions above, his definition of closure is applicable to OOPM, with the understanding that in OOPM the "composition" mentioned above is heterogeneous multimodeling. With the definitions already given, Zeigler's requirements for closure under coupling are met for digital objects if we resolve two things: first, the **experimental frame** issue; and second, the **select** issue. The experimental frame issue has to do with conditions under which a simulation is valid for the system it represents. While the experimental frame issue is important, we are concerned here with verification rather than validation, so we leave the experimental frame issue as outside the scope of the present inquiry, and make the assumption that the experimental frames intersect. The select issue has to do with how to decide the order in which a number of simultaneously scheduled events should be made to occur. Thus, if we can show that transformation of a digital object to a simulation program results in all events (especially simultaneous events) occurring in a proper sequence, this will suffice to show closure under coupling of digital objects. Selecting from simultaneous events will be discussed in Sect. 3, with algorithms for transforming digital objects to simulation programs presented in Sections 3.2.1 and Sect. 3.3.1. These algorithms will be shown by an inductive proof in Sect. 4 to result in construction of a simulation exhibiting correct behavior sequence. Closure under coupling of digital objects in the context of simulations generated under this theory then follows as a corollary in Sect. 4.4.

With closure under coupling, multimodels may be extended upwards or downwards. An upward construction supports integration or reuse; for example, several existing digital objects are combined into a new larger digital object built as a multimodel. The preëxisting digital objects are the constituents of this multimodel. In such an upward extension for integration, new digital object $\mathcal{D}_0$ is created, having an interface, and a multimodel of any type $\in \mathfrak{M}$. Constituents of this multimodel are existing digital objects $\mathcal{D}_1,\ldots,\mathcal{D}_n$, each of which may in turn be further refined. In a more general upward construction, new digital object $\mathcal{D}_0$ is created, with a multimodel whose n constituents are $\{\mathcal{E}_i\}$ such that $\mathcal{GT}(\mathcal{E}_i) \in \mathfrak{G},\ \forall i \in \{1,\ldots,n\}$. Multimodels may also be extended downwards, to facilitate that style of project development, or to selectively refine a model in just those places where fidelity must be increased to meet objectives of the model author. In a downward extension, an atom is replaced by an arbitrary multimodel $\mathcal{E}$ such that $\mathcal{GT}(\mathcal{E}) \in \mathfrak{M}$. Multimodeling may itself be extended by defining new multimodel types; that is, by adding elements to $\mathfrak{M}$, subject to satisfying coupling requirements.

2.5. $\mathcal{G}_{\mathcal{M}}$, the Digital Object Multimodel Grammar

For a digital object, its hierarchy tree $\mathcal{T}$ captures the topology of every multimodel therein, specifying each multimodel's connections to its constituents. Information in $\mathcal{T}$ is, however, only part of what is needed to define the multimodel. We also need other information, from the conceptual model, interfaces, and traces. $\mathcal{G}_{\mathcal{M}}$, which appears as Grammar 2.1, is a modeling grammar described in extended BNF. Many nonterminal symbols are left undefined when the meaning is clear or when the particulars are not of central relevance.

GRAMMAR 2.1 (DIGITAL OBJECT GRAMMAR).

digital-object	⇒	conceptual-model interface
interface	⇒	name text icon io-info [behavior-hierarchy-tree]
io-info	⇒	{in}* {inout}* {out}* return-type
in	⇒	IN type name [default]
inout	⇒	INOUT type name [default]
out	⇒	OUT type name
default	⇒	$\mathfrak{G}$
behavior-hierarchy-tree	⇒	$\mathfrak{G}$
$\mathfrak{G}$	⇒	dynamic-multimodel | digital-object | atom
imported-digital-object	⇒	base-multimodel mm-type {constituent}$^+$ $\mathfrak{M}$
dynamic-multimodel	⇒	base-multimodel mm-type $\mathfrak{M}$
base-multimodel	⇒	io-info $\aleph$ {trace}$^+$
trace	⇒	TRACE from-type end-point to-type end-point
from-type	⇒	IN-PARAMETER | ATTRIBUTE | MMU-OUT-PARAMETER
to-type	⇒	MMU-IN-PARAMETER | ATTRIBUTE | OUT-PARAMETER
end-point	⇒	[constituent] parameter-id | attribute
$\mathfrak{M}$	⇒	fsm | fbm | rbm | sdm
mm-type	⇒	FSM | FBM | RBM | SDM
fsm	⇒	Q Δ q_0 q_c
fbm	⇒	$\mathfrak{B}$
rbm	⇒	rule-set
rule-set	⇒	Ψ K
sdm	⇒	S L R $\mathcal{A}$ $\mathcal{F}$ C
Q	⇒	{constituent}$^+$
Δ	⇒	{constituent}$^+$
$\mathfrak{B}$	⇒	{constituent}$^+$
Ψ	⇒	{constituent}$^+$
K	⇒	{constituent}$^+$
$\mathcal{A}$	⇒	{constituent}$^+$
constituent	⇒	$\mathfrak{G}$

3. TRANSFORMING A MULTIMODEL TO A SIMULATION

We specify a transformation to mechanically and unambiguously take behavior multimodels describing digital object $\mathcal{D}$ to simulation program $\mathcal{P}$. This capability confers benefits which have been enumerated by the authors elsewhere.[9,10] $\mathcal{P}$ exhibits correctness with regard to the sequence of occurrence of its behavior elements; the nature of the transformation is such as to ensure this correctness. Transformation of model hierarchy tree $\mathcal{T}$ thus corresponds to transforming $\mathcal{D}$ to $\mathcal{P}$ such that $\mathcal{P}$ has correct behavior sequence.

3.1. Behavior Sequences

Consider temporal and logical sequence of occurrence of behaviors of behavior elements in $\mathcal{P}$. Let $\{\mathcal{E}_i,\ 1 \leq i \leq n\}$ be the behavior elements of $\mathcal{D}$, with $\mathcal{MT}(\mathcal{E}_i) \in \mathcal{GT}\ \forall i$. $\{\mathcal{E}\}$ corresponds to vertices in $\mathcal{T}$, the hierarchy tree of $\mathcal{D}$. Typically $\mathcal{D}$'s behavior requires the behavior of each $\mathcal{E}_i$ to occur many times during an execution of the simulation, and there is a requirement that temporal and logical sequence of occurrence of the behaviors of $\{\mathcal{E}\}$ be correct. Simulation time is real-valued and monotonically increasing. Each occurrence of behavior element $\mathcal{E}_i$ is associated with some value t of simulation time. In $\mathcal{E} \times t$ space, in which we assign each behavior occurrence to a point, and where we denote a behavior occurrence as $\mathcal{E}_{i,t}$, each point in this space is occupied by at most one behavior occurrence. Sometimes each constituent requires its own event chain, separate from that of other constituents, and from that of the multimodel to which the constituents belong. This situation arises frequently in queuing models with multiple token sources. The result is a need to support multiple schedulable units. With this in mind, we state three behavior sequencing correctness requirements.

Temporal Sequence Let a and b be non-negative real numbers denoting values of simulation time, and let $\prec$ be the "happens before" operator. Then $a < b \rightarrow \mathcal{E}_{i,a} \prec \mathcal{E}_{j,b}$.

Occurrence of Simultaneous Called Behaviors There is a requirement for appropriate sequencing of the set of behavior elements occurring at time t; that is, $\{\mathcal{E}_{i,t}\ \forall\ i\}$. If $\mathcal{E}_i$ and $\mathcal{E}_j$ are constituents of $\mathcal{E}$, and the set of $\mathcal{E}$'s constituents are ordered in a linear sequence (as they are in all the multimodel definitions) and both $\mathcal{E}_i$ and $\mathcal{E}_j$ must occur at time t, then the second sequencing correctness requirement is: $i < j \rightarrow \mathcal{E}_{i,t} \prec \mathcal{E}_{j,t}$.

Occurrence of Simultaneous Scheduled Behaviors There is also a requirement for appropriate sequencing of simultaneous behaviors, but in a more general situation, reflecting multiple scheduling units. Above, behavior of a multimodel includes directly causing occurrence of the behavior of its constituents. We say the multimodel "calls" its constituent. But when constituent behaviors are independently event-scheduled, each with its own event chain, such as in a traditional queuing model, several such behaviors may be required to occur at the same simulation time t. The event scheduler must then cause (behaviors corresponding to) such events to occur in a correct order, yet we do not wish to burden the event scheduler with detailed knowledge about internals of multimodel structure. We use event priority to induce correct event scheduling when several events are scheduled to occur at the same simulation time. This approach hides the complexity of sequencing, mapping information about sequencing onto event priority, a well-known and common capability of many event schedulers. In every event scheduler, a future event list is sorted. The primary sort key is simulation time. In an event scheduler which understands event priority, the future event list secondary sort key is event priority. Define the scheduling priority $\wp(\cdot)$ of an event chain to be the same for every event in the chain, and let the scheduling priority be integer-valued with the convention that larger values mean higher priority. Thus if events $\{e_1, e_2, \cdots, e_n\}$ from different event chains are all scheduled to occur simultaneously at time t_1, and if each e_i has an associated scheduling priority $\wp(e_i)$, and if all the $\wp(\cdot)$ are unique, then the first event to occur will be $\{e_i \mid \wp(e_i) > \wp(e_j),\ \forall\ j \neq i\}$ and remaining events will occur in descending order of $\wp(\cdot)$. Let k be the scheduling priority of event e_k in the event chain. Thus, $\wp(e_k) = k$, and if $\mathcal{E}_i$ and $\mathcal{E}_j$ are constituents of $\mathcal{E}$, and $\mathcal{E}_i$ and $\mathcal{E}_j$ are scheduled, and $\mathcal{E}$'s constituents are ordered by priority $\wp$, and both $\mathcal{E}_i$ and $\mathcal{E}_j$ must occur at time t, then $\wp(\mathcal{E}_i) > \wp(\mathcal{E}_j) \rightarrow \mathcal{E}_{i,t} \prec \mathcal{E}_{j,t}$. The third sequencing correctness requirement then is: $i < j \rightarrow \wp(\mathcal{E}_i) > \wp(\mathcal{E}_j)$.

3.1.1. Sequencing Schedulable Units

Digital object $\mathcal{D}$ has multimodel hierarchy tree $\mathcal{T}$. Each vertex $\mathcal{E} \in \mathcal{T}$ corresponds to a behavior element. Every $\mathcal{E}$ except the root of $\mathcal{T}$ has a parent. Let $\mathcal{M} = parent(\mathcal{E})$, and the relation between $\mathcal{E}$ and $\mathcal{M}$ is that of constituent to multimodel. Let $\{C_i\}$ denote the constituents of $\mathcal{M}$. Then $\mathcal{E} \in \{C_i\}$. Every simulation of interest has at least one schedulable unit (SU), corresponding to an event chain which causes the behavior of the root of $\mathcal{T}$. A behavior element $\mathcal{E}$ which must have an independent event chain requires an SU separate from the SU of $\mathcal{M}$, and separate from the SU of every other C_i. A 3D representation will be presented for a collection of related SU's representing a simulation, with each SU_k lying in the x-y plane at $z = k$. A scheduling structure for a simulation may have one or many SU's. We shall show how to make the z-coordinate of an SU determine behavior sequence which is deterministic, temporally and logically sequenced, and hence correct.

3.2. Inversion, Rearrangement, and Promotion

All scheduled constituents occur before any called constituent occurs. If a multimodel has some constituents called and others scheduled, a problem may arise. The order of occurrence of constituents is the order of their appearance from left to right as siblings in the hierarchy tree. If in this sequence appears $\ldots C_i, C_{i+1} \ldots$ such that $CT(C_i) =$ *called* and $CT(C_{i+1}) =$ *scheduled* then an inversion exists, because all called constituents have the same priority as that of the multimodel to which the constituents belong, so there exists no priority between that of the called constituents and that of the parent multimodel; hence no way to schedule C_{i+1} to occur between C_i and their parent multimodel. Two ways to resolve inversion without affecting correctness are rearrangement and promotion. Rearrangement is preferred (because it is "free" in the sense that it imposes no runtime overhead on a simulation), but sometimes rearrangement cannot be used. Then we use promotion which is always available.

In $\mathcal{P}$, the order of occurrence of constituents must conform to the partial order imposed by dependencies associated with traces. Within partial order we are free to choose any total order. One specific rearrangement is of constituents within Q. The FSM definition requires the states to appear in order $q_1, \ldots, q_n$, but correct behavior sequence is satisfied by *any* ordering of $\{q_i\}$, because exactly one state occurs at one value of simulation time, by definition of an FSM, hence temporal order suffices, no matter what the ordering of sibling states. Another rearrangement is of those constituents which are $\in \Re$. Permitted rearrangements are within a constituent group only; they cannot cross a group boundary. For example, in an FSM, where the constituents are states and transitions, the invariant is $Q \prec \Delta$. We may rearrange within group Q and within group Δ, but we must *never* rearrange so as to result in the condition $\exists\ \delta_{i,j} \prec q_k$. In an FSM for which $\Delta \subset \Re$, any order of occurrence of $\{\delta_{i,j}\}$ will do, because at simulation time t, only the transitions in $\Delta_{qc} = \{\delta_{i,j} | i =$position of q_c in $Q\}$ can occur, the transitions in Δ_{qc} have no partial ordering imposed by traces because none of them alters any attribute used by any of the others (because $\Delta_{qc} \subset \Re$), and at most one of the predicates of Δ_{qc} can be true at any simulation time (by the FSM definition), hence the order in which we execute transitions $\in \Delta_{qc}$ will not affect which one (if any) we find to be true. In an RBM for which $\Psi \in \Re$, any order of occurrence of $\{\psi_i\}$ will do, because the premises in Ψ have no partial ordering imposed by traces because none of them alters any attribute used by any of the others (because $\Psi \subset \Re$), and a premise influences exactly one consequence, which is a consequence different from the consequence influenced by any other premise, i.e.: ψ_i affects κ_j if and only if $i = j$, and $\Psi \prec K$, so if we denote as K_t the subset of K which occurs at time t, then $K_t = \{\kappa_i | \psi_i = true\}$, so the order in which we learn which premise is true, and which consequences should occur, does not affect membership in K_t nor the order of execution of its elements, which is determined solely by the position within K of κ_i.

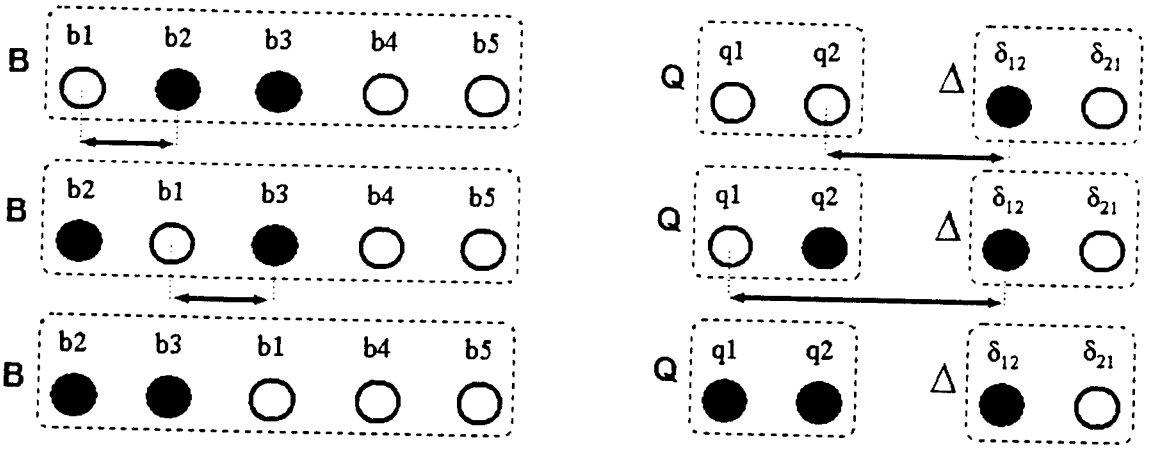
When we cannot rearrange to eliminate inversions, we **promote** inverted constituents from *called* to *scheduled*. A promoted constituent can then have any requisite scheduling priority, hence any order of occurrence, hence no inversion. However, promotion imposes additional runtime cost on the simulation, in the form of event scheduling overhead. Thus rearrangement is preferable.

3.2.1. Inversion elimination algorithm

The RESOLVEINVERSIONS($\mathcal{M}$) algorithm (Algorithm 3.1) resolves inversions in constituents of a multimodel $\mathcal{M}$ in two ways, *rearrangement* and *promotion*:

ALGORITHM 3.1 (RESOLVEINVERSIONS).

```
RESOLVEINVERSIONS(M)
1  while ∃ inversion in multimodel M (at constituents C_i and C_{i+1})
2      if  C_i ∈ Q ∧ C_{i+1} ∈ Q
3      or  C_i ∈ Δ ∧ C_{i+1} ∈ Δ ∧ C_i ∈ ℜ ∧ C_{i+1} ∈ ℜ
4      or  C_i ∈ Ψ ∧ C_{i+1} ∈ Ψ ∧ C_i ∈ ℜ ∧ C_{i+1} ∈ ℜ
5      or  C_i and C_{i+1} are FBM blocks  and  ¬ C_i ∈ predecessors(C_{i+1})
6      then swap C_i and C_{i+1}
7      else break
8  while ∃ inversion in multimodel M (at constituents C_i and C_{i+1})
9      CT(C_i) = scheduled
```



(a) Rearrangement

(b) Promotion

Figure 2. Resolving Inversions: Rearrangement preferred, promotion otherwise. Hollow circles are called constituents; filled circles are scheduled constituents; inversion may exist when a hollow circle is to the left of any filled circle.

The first `while` loop in lines 1 through 7 of Algorithm 3.1 performs rearrangement. It scans the siblings which are constituents of a particular multimodel. To see the operation of this loop, consider the top row of Fig. 2(a). For the sake of exposition, suppose b_2 and b_3 are scheduled. An inversion is found involving b_1 and b_2. Because $b_1 \not\prec b_2$ the condition in line 5 of the algorithm is true. As a result b_1 and b_2 are swapped by line 6 of the algorithm, and we have the situation shown in the second row of Fig. 2(a). The while loop condition this time finds the inversion involving b_1 and b_3, and the body of the while loop again executes. Because $b_1 \not\prec b_3$, the line 5 condition is again true, and line 6 again executes. The result is as shown on the third row of Fig. 2(a). This time when the while loop condition is tested, it is false because no more inversions exist, and the rearrangement loop ends. In this example the condition in line 8 is not met, so no promotion takes place. The algorithm terminates with no inversions remaining among the constituents of the multimodel.

The second loop in lines 8 and 9 of Algorithm 3.1 performs promotion. To see its operation, consider the top row of Fig. 2(b), whose constituents belong to an FSM. The condition in line 1 is true because there is an inversion involving q_2 and $\delta_{1,2}$. However, none of the conditions in lines 2 through 5 of the algorithm is true, so line 7 executes, terminating the loop. At line 8, the condition is true, so q_2 is promoted, and we have the situation shown on the second row of Fig. 2(b). At line 8 again, the condition is still true, and this time q_1 is promoted, leading to the situation shown on the third row of Fig. 2(b). Now at line 8, the condition is false because no inversions remain, so the loop terminates, and the algorithm terminates, with all inversions resolved. Finding an intergroup inversion immediately ends all attempts at rearrangement. While a more sophisticated algorithm could temporarily ignore intergroup inversions and keep trying to resolve (other) inversions by rearrangement until the only inversions remaining are those requiring promotion, such efforts are sometimes wasted because we may subsequently find we may need to promote the same constituents we just rearranged; hence we leave the algorithm as is.

3.3. Simulation Scheduling Structure $\mathcal{F}$

Suppose $\mathcal{D}$ has been constructed by a model author. $\mathcal{D}$ has a representation as a multimodel, corresponding to which is $\mathcal{T}$. We now elaborate a structure for generating correct behavior element sequence of a simulation program from $\mathcal{T}$. A principal feature of the structure is its representation of the schedulable units of the simulation, and the way it relates schedulable units to behavior elements. $\mathcal{T}$ is of course planar (two-dimensional). Each vertex appears as a circle. The root is at level 0. We construct $\mathcal{F}$, a 3D simulation scheduling structure which we show to have an interpretation which corresponds to a mechanical unambiguous generation of $\mathcal{P}$ having the property that temporal and logical sequence of the occurrence of the behaviors of its behavior elements is correct. The structure corresponds to, and preserves the information of, $\mathcal{T}$; indeed, $\mathcal{T}$ is a projection in 2D of the 3D $\mathcal{F}$. Let $\mathcal{E}$ be the root of any subtree

of $\mathcal{T}$, and let $\{\mathcal{C}_j,\ 1 \leq j \leq n\}$ be the n constituents of $\mathcal{E}$. Let CT be the set $\{scheduled,\ called\}$, and denote the causation type of $\mathcal{C}_j$ as $\mathcal{CT}(\mathcal{C}_j)$. Let $\mathcal{CT}(\mathcal{C}_j) \in CT\ \forall\ \mathcal{C}_j$. The only vertex not included in this constraint is the root of $\mathcal{T}$. Let $\mathcal{R}$ denote the root of $\mathcal{T}$. Let $\mathcal{CT}(\mathcal{R}) = scheduled$. Let $\mathcal{F}$ be the scheduling structure corresponding to $\mathcal{T}$. The number of vertices in $\mathcal{F}$ is the same as the number of vertices in $\mathcal{T}$. Information content of each vertex $\mathcal{E}_f$ in $\mathcal{F}$ is identical with that of the corresponding vertex $\mathcal{E}_t$ in $\mathcal{T}$, except for the addition of a new non-negative integer attribute $\wp$ in $\mathcal{E}_f$. Whereas $\mathcal{E}_t$ is a circle, $\mathcal{E}_f$is a cylinder, whose base is the circle which is vertex $\mathcal{E}_t$, located in the plane $z = 0$. The height of $\mathcal{E}_f$'s cylinder is the value of $\wp(\mathcal{E}_f)$. Thus the value $\wp(\mathcal{E}_f)$ is the distance from the $z = 0$ plane to the top surface of the cylindrical $\mathcal{E}_f$ vertex; $\wp(\mathcal{E}_f) \geq 0\ \forall\ \mathcal{E}_f$ denoting that the top of each vertex is in or above the $z = 0$ plane.

3.3.1. Elevator Algorithm

Given $\mathcal{D}$ and a corresponding $\mathcal{T}_\mathcal{D}$, copy $\mathcal{T}_\mathcal{D}$ to (initially flat) scheduling structure $\mathcal{F}_\mathcal{D}$, with $\wp(\mathcal{E}) = 0\ \forall\ \mathcal{E} \in \mathcal{F}_\mathcal{D}$; thus, $\mathcal{F}_\mathcal{D}$ lies entirely in the plane $z = 0$ at this juncture. Then, recursive procedure Elevator$(\mathcal{E}, z)$ first resolves inversions, if any, then traverses $\mathcal{F}_\mathcal{D}$, setting $\wp$ of each vertex in such a way that $\mathcal{F}_\mathcal{D}$ takes on its proper 3D shape. Elevation of vertices of $\mathcal{F}_\mathcal{D}$ is performed by calling Elevator$(\mathcal{R}, 0)$, where $\mathcal{R}$ is the root of $\mathcal{F}_\mathcal{D}$, zero is the initial value of z; recursion does the rest. Traversal order is a variant of preorder: visit the root, then traverse its subtrees, starting with the rightmost, and proceeding from right to left. To elevate the structure rooted at $\mathcal{E}$ given non-negative integer z, we execute ELEVATOR$(\mathcal{E},\ z)$. The algorithm and detailed exposition of its operation appear in the base paper.

3.3.2. Special case for scheduled constituents $\in \Re$

In Section 3.1.1 we stated that every scheduled constituent must have a scheduling unit different from every other sibling constituent, and this is reflected in the Elevator algorithm. This requirement can be relaxed somewhat under a special circumstance. If $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ are (adjacent) sibling constituents of multimodel $\mathcal{M}$ and $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ are both scheduled and $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ are both in the same group (e.g., Δ), and $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ are both $\in \Re$, then $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ can both be assigned the same elevation. We leave it to the event scheduler to determine the order of occurrence, as that order does not affect simulation correctness. This relaxation is not necessary but can be used when a structure with minimal elevation is desired.

3.4. Summary

Starting with $\mathcal{D}$ and its corresponding $\mathcal{T}$, we constructed $\mathcal{F}$ from $\mathcal{T}$. $\mathcal{F}$ is in effect a 3D version of the 2D $\mathcal{T}$. We related a property of $\mathcal{F}$, namely the z-coordinate of the top surface of each cylindrical vertex in $\mathcal{F}$, via scheduling priority, to the behavior sequence of $\mathcal{P}$ corresponding to $\mathcal{D}$.

4. PROOF OF CORRECT BEHAVIOR SEQUENCING

4.1. Introduction

It remains to show that $\mathcal{F}$ and hence $\mathcal{P}$ exhibit correct behavior sequence. This is done with an inductive proof. Closure under coupling of digital objects follows as a corollary. Let b_i denote a vertex in $\mathcal{T}$ or $\mathcal{F}$, where i is an index set over $\mathcal{T}$ or $\mathcal{F}$. Let $e_{i,t}$ denote an associated event; that is, occurrence of the behavior b_i at simulation time t. The "occurs before" operator $\prec$ in the expression $e_{i,t1} \prec e_{j,t2}$ means that either: b_i occurs at simulation time t_1, b_j occurs at simulation time t_2, and $t_1 < t_2$; or, two behaviors occur at the same simulation time $t_1 = t_2$ and b_i precedes b_j in the execution sequence at this instant of simulation time. The "occurs before" operator is also used between two sets of behaviors, such as $A \prec B$, to mean $a_i \in A \prec b_j \in B\ \forall\ i, j$. Let $\mathcal{P}$ denote a simulation program. Suppose the code of $\mathcal{P}$ consists of code units $\{b_i\}$, and that each code unit b_i either implements some dynamic multimodel, *e.g.*, a functional block model, or is an *atom*. Suppose also that if b_i implements a behavior multimodel, then the code logic of b_i is such that the logical sequence in which b_i calls subordinate code units is fixed.

DEFINITION 4.1 (CALLING SEQUENCE). *Given simulation program $\mathcal{P}$ with a set of code units $\{b_i\}$, a **calling sequence** is defined at each code unit b_i as an ordered list whose elements are the code units (if any) which b_i calls, and whose sequence is the unique logical sequence in which the calls occur in b_i. The overall calling sequence for $\mathcal{P}$ is defined as the sequence produced by starting with the code sequence for the main level code unit of $\mathcal{P}$, inserting immediately before each element the calling sequence of that element, and continuing until no new insertions remain to be performed.*

DEFINITION 4.2 (CORRECT BEHAVIOR SEQUENCE). *Correct behavior sequence requires:*

(1) events occur in temporal order: $t1 < t2 \rightarrow e_{i,t1} \prec e_{j,t2}\ \forall\ i, j$;

(2) events which occur at the same simulation time occur in a sequence consistent with constituent groupings defined for each kind of multimodel:
for an FSM, $Q \prec \Delta$;
for an RBM, $\Psi \prec K$;

(3) behaviors which occur at the same value of simulation time occur in an execution sequence consistent with a partial order imposed upon sibling constituents of a multimodel by traces, i.e.:
$b_i \in predecessors(b_j) \rightarrow e_{i,t1} \prec e_{j,t1}$; *and,*

(4) constituents of a multimodel occur before the multimodel occurs, where "occurrence" of a multimodel means determining its outputs.

THEOREM 4.1 (CORRECT TRANSFORMATION OF DIGITAL OBJECTS). *Given well-formed digital object* $\mathcal{D}$ *with hierarchy tree* $\mathcal{T}$, *and given simulation scheduling structure* $\mathcal{F}$ *constructed by transforming* $\mathcal{T}$ *using* ELEVATOR *and* RESOLVEINVERSIONS *algorithms. Then a simulation program* $\mathcal{P}$

- *whose code units are in one-to-one correspondence with the vertices of* $\mathcal{F}$,
- *whose calling sequence is the same as the sequence produced by (left-to-right) preörder traversal of* $\mathcal{F}$, *and*
- *whose scheduling priorities are numerical values assigned so that the scheduling priority of a code unit of* $\mathcal{P}$ *equals the height of the cylinder of the corresponding vertex of* $\mathcal{F}$,

has correct behavior sequence.

The third requirement for correct behavior sequence is weaker in several ways than the ordering in multimodel definitions. We exploited this to perform rearrangement within Q, Δ, and Ψ. Note the absence above of any specification of semantics of atoms of $\mathcal{D}$ in Theorem 4.1. We make no claim that $\mathcal{P}$ is a correct program, only that it correctly implements multimodel semantics and that the behavior sequence of its behavior elements is correct. We combine atoms and small digital objects into larger digital objects in such a way that the combining action introduces no error. Progressively larger digital objects we build up will be correct in two respects: implementation of multimodel semantics, and behavior sequence. Although outside the scope of this paper, it follows that if the atoms can be shown to be correct, then $\mathcal{P}$ can be shown to be correct as well.

4.2. Special Case: $\mathcal{F}$ is Flat

We consider an important special case with some $\mathcal{D}$ and its corresponding $\mathcal{T}$. $\mathcal{D}$ has an arbitrary amount of refinement. Denote the root of $\mathcal{D}$ as b_0. $\mathcal{CT}(b_0) =$ *scheduled* in consequence of grammar production rules (reported in the base paper) for the scheduling structure. It often happens that $\mathcal{CT}(b_i) =$ *called* for all vertices other than b_0. The construction algorithms then build $\mathcal{F}$, without rearrangement, promotion, or elevation: $\mathcal{F}$ is flat, entirely located in the plane $z = 0$, and is identical to $\mathcal{T}$. $\mathcal{F}$ and hence $\mathcal{P}$ exhibit correct behavior sequence, shown as follows: temporal order of occurrence of $\mathcal{F}$ as a whole is determined by the event scheduler, satisfying requirement 1 of Dfn. 4.2. The logical ordering of behavior elements in a plane at any simulation time t is wholly determined by the structure of the code emitted at the time $\mathcal{P}$ is created. This order corresponds to (left to right) postorder traversal of $\mathcal{F}$, satisfying requirements 2 through 4 of Dfn. 4.2. Hence $\mathcal{P}$ exhibits correct behavior sequence.

4.3. The General Case

A preliminary version of the proof appears in the base paper; the full proof appears elsewhere.[11] A sketch of the proof follows. Consider some $\mathcal{D}$ and its corresponding $\mathcal{T}$. $\mathcal{F}$ is constructed from $\mathcal{T}$. $\mathcal{T}$ is in one of three groups (see Sect. 2.3) which partition the set of all model hierarchy trees.

$\mathcal{F} \in \{\mathcal{T}_1\} \cup \{\mathcal{T}_2\}$: In $\mathcal{T}_1$ the root is an atom. An atom is indivisible; hence, its behavior sequence is vacuously correct. In $\mathcal{T}_2$, children of the root are all atoms. Correct sequencing is shown by an argument which partitions the children into two sets (called and scheduled) and then examines the algorithms ELEVATE and RESOLVEINVERSIONS with regard to the height of each vertex of $\mathcal{F}$, which is also its scheduling priority. The behavior sequence of the scheduled children is shown to be from left to right, before any called sibling constituent, and before the root, and thus is correct. All constituents are thus in correct behavior sequence relative to one another and to the multimodel which they constitute.

$\mathcal{F} \in \{\mathcal{T}_m\}$: $\mathcal{T}$ has an arbitrary amount of refinement. $\mathcal{F}$ is constructed from $\mathcal{T}$ following the construction algorithms. An inductive proof shows that $\mathcal{F}$ and hence $\mathcal{P}$ exhibit correct behavior sequence. Induction is on induction variable η (lowercase Greek *eta*). A value of η is a property of every vertex in $\mathcal{T}$, and hence of $\mathcal{F}$. η is a measure related to the length of the path distance of a vertex from the root of $\mathcal{T}$: if we place $\mathcal{T}$ with it root down and its leaves up, then η of the highest leaf is zero, and η increases as we move downward toward the root. An algorithm for calculating η is given in the base paper. The proof starts by considering $\eta = 0$. All such vertices are atoms, whose behavior sequence is vacuously correct. Therefore behavior sequence of every vertex for which $\eta = 0$ is correct. The proof continues with $\eta = 1$. Some vertices for which η is 1 may be atoms because $\mathcal{T}$ in general is not a complete tree, and so has leaves at a number of different levels. A vertex for which η is 1 which is not an atom is the root of a model tree of height 2. Trees of height 2 were previously shown to have correct behavior sequence. Hence all subtrees for whose root $\eta = 1$ have correct behavior sequence. Next comes the inductive step, in which the inductive hypothesis is that behavior sequence is correct for any subtree of $\mathcal{F}$ for whose root $\eta < k$. It is required to show behavior sequence correct for any subtree of $\mathcal{F}$ for whose root $\eta = k$. The argument is similar to that above for a tree of height 2. The inductive step completes the proof.

4.4. Closure of Multimodeling under Coupling

Consider digital object $\mathcal{D}$ built as a behavior multimodel, for which we construct a model hierarchy tree $\mathcal{T}$ which has an arbitrary amount of refinement. $\mathcal{F}$ is constructed from $\mathcal{T}$. The behavior sequence of the simulation program $\mathcal{P}$ generated from $\mathcal{F}$ has been shown to be correct. Coupling and closure under coupling were introduced in Sect. 2.4. **Intralevel coupling** is specified by six rules: (1) topology is that of a well-formed directed graph; (2) constituents are partitioned into sets, with elements of each set subject to rules governing behavior of that set; (3) like items are uniquely ordered; (4) corresponding items are placed into unique correspondence; (5) traces specify data dependencies; and, (6) partial ordering is determined by the traces. The first four conditions above are met by multimodel specifications in Sect. 2.2. The fifth and sixth conditions are met by the specification of traces in Sect. 2.3. **Interlevel coupling** is specified by three rules: (1) hierarchy (see Sect. 2.3); (2) heterogeneity (see Sect. 2.3); and, (3) adherence to multimodel semantics specified in Sect. 2.2. These are captured in a recursive definition in the behavior multimodel grammar (Grammar 2.1 on page 6) which allows integration of multimodels, digital objects, and atoms into larger digital objects. As mentioned in Sect. 2.4 we make the assumption that all experimental frames intersect. We proved that transformation of $\mathcal{D}$ to $\mathcal{P}$ results in all events (especially simultaneous events) occurring in a proper sequence. Therefore digital objects exhibit closure under coupling.

5. ARCHITECTURE

5.1. Introduction

The purpose of the architecture is to support creation, modification, examination, combination, and reuse of digital objects. The base architecture is shown in Fig. 3. It is an abstract base class. Configurations are derived from it, and only these derived classes are instantiated. Digital objects created under one configuration can be used without modification under any other configuration. Seven *Intrinsic* components (blocks with heavy outline in Fig. 3) are creations of this research. Five *External* components (blocks with thin outline) are the work of others. The architecture is essentially a way of getting the external components (including the human model author) to coöperate in creating and sharing digital objects. Intrinsic components are the architecture's way of making this happen. Some components and paths are unused in some configurations, and some paths have bifurcations to indicate they play several rôles.

5.2. Components

Intrinsic components are Bridge, Translator, Engine, Scenario, Digital Object Warehouse, Digital Object Tool (DOT), and Plug-in/Applet/CGI. "Plug-in" means "Web browser plug-in". Plug-ins are often written as a C++ dynamically linked library (DLL). CGI is the Common Gateway Interface, and a CGI "script" may be, for example, a Perl script or a C++ executable. DOT is digital object tool, closely tied to Modeler, helping to find and reuse digital objects. Scenario is used to visualize simulation runtime output. Translator automates the mapping of a model to a computer simulation program. Engine carries out the simulation. Translator, Engine, and Scenario have been described by the authors elsewhere.[9] The warehouse stores and organizes model information, so that it can be retrieved by DOT. Bridge is the top level user interface for the model author. It provides access to the other components, such as DOT, Modeler, Translator, Engine, and Scenario: using Bridge, the model author can have Translator transform a

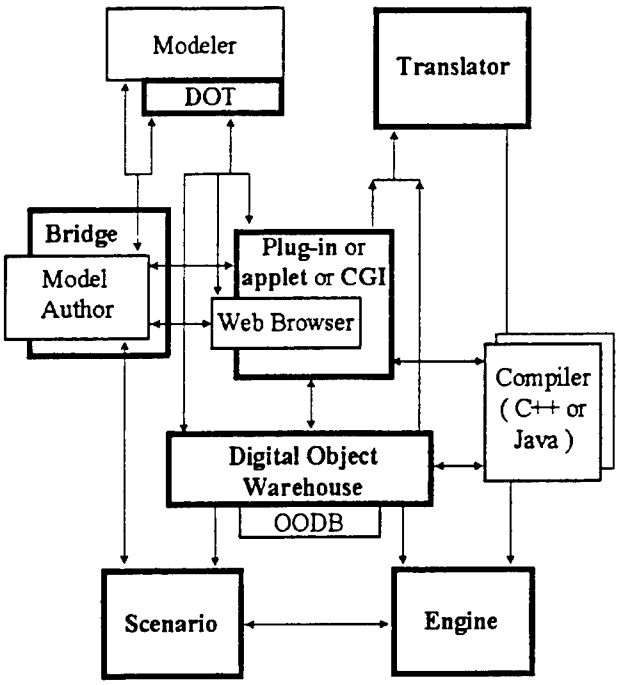


Figure 3. Base Architecture.

multimodel description, produced by Modeler, into a simulation program, and then activate Scenario, which in turn activates Engine and displays visualization of simulation output produced by Engine. DOT helps the model author to create, reuse, or combine digital objects, as well as to browse among digital objects. DOT attaches to the Modeler and the model author works with the modeler through DOT, and DOT sets things up for Modeler. This permits DOT to provide a context for modeling, to fetch all needed model representations, to resolve collisions, if any, and to make everything available to the Modeler. External components are the human model author, modeler, compiler, Web browser, and object-oriented database (OODB). Modeler is used by the human model author to build and change digital objects. Modeler has been describe by the authors elsewhere.[9] The OODB provides an optional back end at the Warehouse, for scalability of the architecture, and for replication of information and similar capabilities.

5.3. Configurations

Four configurations (local, internet, browser plug-in, and applet) appear as Fig. 4. They suit different needs, provide a migration path as needs change, and make the architecture extensible and tolerant of change. Some paths use TCP/IP on the Internet (among Modeler, Translator, and Warehouse); others use the Web–based HyperText Transmission Protocol (HTTP) with a Web browser such as Netscape or Microsoft Internet Explorer, plus a browser plug–in; or via XML. The language spoken on TCP/IP and XML pathways is Digital Object MultiModel Language (DMML).[11] These configurations by no means exhaust all possibilities; for example, a preliminary investigation was made into an XML configuration. This helped shape DMML as a markup language so that it can be defined by an XML Document Type Definition (DTD). The configurations and the components are described in detail elsewhere.[11] Model authors can use the local configuration to preview the work of others. The Internet configuration can be used to allow a workgroup to share digital objects and to engage in collaborative development. A platform–dependent plug–in configuration relies on a Web browser to interact with the model author and control and interact with the other components. An applet configuration can be used for the same purpose but with greater platform independence while eliminating the need to install development software locally. Finally, if DMML is widely accepted, the Digital object MultiModel language (DMML) used by the components can be defined to an XML–capable Web browser, and components can communicate in DMML directly via the Web browser, obviating the necessity for a browser-plug-in. Multimodel descriptions, expressed in DMML, are usable in any of these configurations, or other future configurations derived from the base architecture.

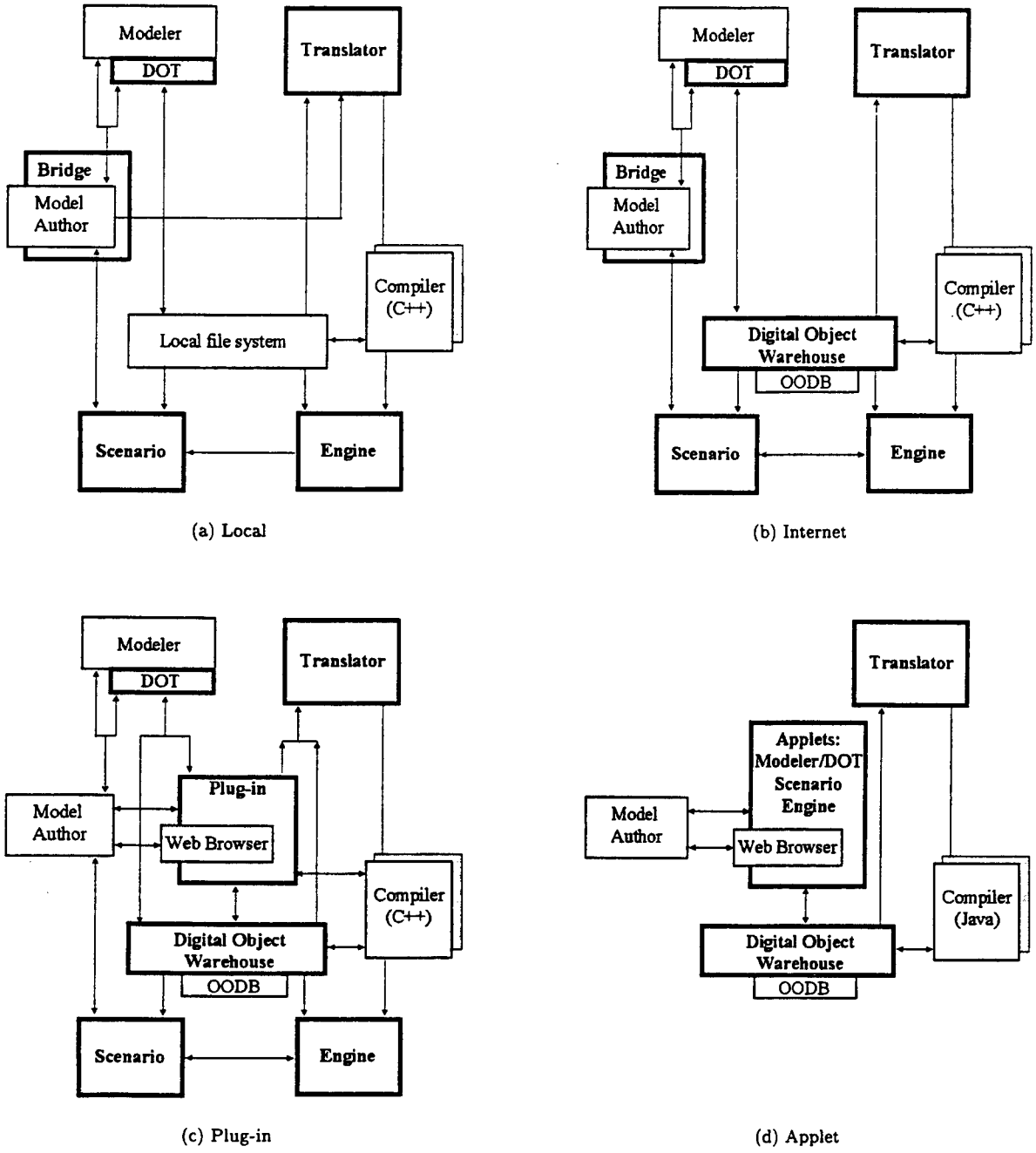


Figure 4. Configurations: Configurations derived from the abstract base architecture are shown above: local (a), internet (b), plug-in (c), and applet (d). Blocks with heavy outline are *intrinsic*; other blocks are *external*. DOT is digital object tool. OODB is object-oriented database. From (a) to (b), note replacement of the local file system with Warehouse; from (b) to (c), note replacement of Bridge with a browser plug-in; and, in (d), note that Modeler, Scenario, and Engine have all become applets.

Local configuration appears in Fig. 4(a). As its name suggests, this configuration is self-contained within a (typically single-user) computer system. Model descriptions are expressed in DMML. Model and other information is passed through the local file system. A directory tree organizes the collection of digital objects. Some flexibility if available by using symbolic links or shortcuts, but a complex collection of digital objects is beyond the intended use of this configuration. Model persistence is accomplished using the local file system. A digital object is defined in DMML in a set of flat ASCII files, all located in the same directory. Benefits of this approach include: such model definitions are simple, compact, portable, relatively easy to read, understand, and modify; model definition files get backed up as part of local system backups; models can be put on diskette, into a `.zip` archive, or FTP'd; as a software engineering tool, local files generated by hand can eliminate some development bottlenecks due to circular waits. But the implementation which uses the flat files for model persistence is stand-alone software, with no provision for sharing, thus limiting reuse; moreover, this software can be used on a machine only after it is obtained and installed on that machine. Reasons such as privacy, national security, performance, or proprietary software, may make this configuration preferable to the internet or Web–based configurations.

Internet configuration appears in Fig. 4(b). Two major features which distinguish this configuration from the local configuration: the presence of Warehouse and use of sockets and connection-based Transmission Control Protocol (TCP) over Internet Protocol (IP) for communication on several paths. Persistent digital object definitions reside within Warehouse rather than a local file system, resulting in several benefits: digital objects reside where they can be best be cataloged, shared, and reused; a digital object can be modeled on one machine and transformed to a simulation program on a different machine or at the warehouse, supporting the ability to build platform-specific Engines if needed; and, a digital object can be modeled on a machine with limited or no disk space. TCP/IP is used to communicate between DOT and Warehouse, permitting warehouse to be located on a machine different from model authors' machines, and allows model authors to share access to the warehouse. TCP/IP also provides bidirectional communication between Engine and Scenario with sockets, allowing Scenario to operate synchronously with Engine and to be located on either the same or a different machine. Because the connection is bidirectional, it allows not only the kind of operations which playing back the history file permits, but also allows Scenario to control Engine during the simulation execution, by changing simulation parameters. The TCP/IP socket output of Engine also affords a ready connection point for attachment to an external visualization or other analysis tool located anywhere on the Internet. The presence of Warehouse and use of TCP/IP do not *per se* make a "Web–based" environment. One reason is that Warehouse communicates with the other components with TCP/IP, which is an Internet protocol, rather than a Web–based protocol such as HTTP, XML, or CGI.

Plug–in configuration appears in Fig. 4(c). It orchestrates interaction among locally installed software components, such as a Modeler and a Translator, using a web browser plug–in. The plug–in component serves the same rôle as Bridge; however, the plug-in component runs as part of the Web browser, whereas Bridge runs as locally installed software. If a Web browser is present on the machine, and because the plug-in component can be delivered over the Web and installed automatically, this configuration requires less system administrator capability than a configuration using Bridge, which not only requires installation of Bridge itself but also, for example, a local copy of the Tcl/Tk interpreter. Both the plug-in configuration and the (about to be described) applet configuration are Web–based, because they rely on a Web browser to examine a URL or an HTML tag and based on that to activate Web–based software, such as the plug-in or an applet and because the Web browser (and its plug-ins and applets) form the primary user interface with the model author. The key characteristic which distinguishes the plug-in configuration from the applet configuration is the ability of the Web browser plug-in to activate a local process.

Applet configuration appears in Fig. 4(d). This is a Web–based configuration. Applets are probably Java applets, although Tcl/Tk also offers an applet, called a "Tclet." Applets are software which are automatically delivered over the Web with no need for local software installation. They do require that a Web browser be installed. Applets work within the Web browser with no separate software installation. A Java virtual machine is present within the browser, and a Tclet interpreter can be installed (the Tcl plug–in 2.0). In contrast with a browser plug–in, applets are subject to a security model which provides clients with confidence that the applets are not performing malicious or destructive acts. This is good in that model authors may be willing to use applets, when they might be concerned with what an uncontrolled plug-in might do; however, the security model can add considerable difficulty to legitimate implementations. In the applet design configuration, Modeler is an applet. Translator can exist locally as an applet, or as a C++ executable or other program at the Warehouse. Engine and Scenario are applets, as is DOT. The key characteristic which distinguishes the Applet design configuration from the plug–in design configuration is that almost everything local is an applet, and it is relatively platform independent.

6. CONCLUSION

We defined digital objects as an extension of heterogeneous behavior multimodels. We defined the principal extension, which is the interface. We showed how to take a digital object to a multimodel by applying the DOT transform. We set forth the abstract base multimodel and specified four derived multimodel types. We described constituent partial order, specified multimodel hierarchy, and stated the two heterogeneity requirements, and discussed coupling and closure under coupling. We stated a digital object grammar. We showed how to transform a digital object a simulation program, honoring three requirements for behavior sequencing, and then showed that the transformation results in simulation programs which are correct with respect to execution sequence, and with an assumption regarding experimental frame, we showed closure under coupling of digital objects. We set forth an abstract base architecture for digital object flow and persistence, and showed four instantiated subtypes thereof.

This work provides a formal basis for multimodeling, and the architecture provides a uniform environment across quite different environments for digital object flow and persistence. The architecture provides the hooks necessary to integrate model repositories with the digital object development environment.

ACKNOWLEDGMENTS

We thank the following agencies that have contributed towards our study of modeling and simulation: (1) Jet Propulsion Laboratory under contract 961427 *An Assessment and Design Recommendation for Object-Oriented Physical System Modeling at JPL* (John Peterson and Bill McLaughlin); (2) Rome Laboratory, Griffiss Air Force Base under contract F30602-98-C-0269 *A Web-Based Model Repository for Reusing and Sharing Physical Object components* (Al Sisti and Steve Farr); and, (3) Department of the Interior under grant 14-45-0009-1544-154 *Modeling Approaches & Empirical Studies Supporting ATLSS for the Everglades* (Don DeAngelis and Ronnie Best). We are grateful for their continuing financial support.

REFERENCES

1. R. M. Cubert and P. A. Fishwick, "Modeling the simulation execution process with digital objects," in *Enabling Technology for Simulation Science III*, A. F. Sisti, ed., *Proc. SPIE* **3696**, pp. 2–22, 1999.
2. D. R. C. Hill, *Object-Oriented Analysis and Simulation*, Addison-Wesley, Reading, MA, 1992. page 104.
3. B. P. Zeigler, "Review of theory in model abstraction," in *Enabling Technology for Simulation Science II*, A. F. Sisti, ed., *Proc. SPIE* **3369**, pp. 2–13, 1998.
4. P. A. Fishwick and K. Lee, ""Two Methods for Exploiting Abstraction in Systems"," in *Proceedings of the Seventh Annual Conference on AI, Simulation and Planning in High Autonomy Systems: Distributed Interactive Simulation Environments*, vol. 7, pp. 257–264, Institute of Electrical and Electronic Engineers Computer Society (IEEECS), Los Alamitos, CA, 1996.
5. K. Lee and P. A. Fishwick, ""Semi-automated Method for Dynamic Model Abstraction"," in *SPIE AeroSense97 Conference Proceedings, volume 3083*, SPIE, Bellingham, WA, 1997.
6. P. A. Fishwick, *Simulation Model Design and Execution : Building Digital Worlds*, Prentice Hall, Englewood Cliffs, N. J., 1995. (For multimodel types, see chapters 4 and 5.).
7. B. P. Zeigler, *Theory of Modeling and Simulation*, John Wiley and Sons, New York, 1976.
8. B. P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, Harcourt Brace Jovanovich, Orlando FL, 1984.
9. R. M. Cubert and P. A. Fishwick, "Software architecture for distributed simulation multimodels," in *Enabling Technology for Simulation Science II*, A. F. Sisti, ed., *Proc. SPIE* **3369**, pp. 154–163, 1998.
10. R. M. Cubert and P. A. Fishwick, "OOPM: An Object-Oriented Multimodeling and Simulation Application Framework," *Simulation* **70**, pp. 379–395, June 1998.
11. R. M. Cubert, *Digital Objects in Object-Oriented Physical Multimodeling and Simulation.* PhD thesis, University of Florida, Gainesville, FL 32611-6120, August 1999.

OOPM/RT: A Multimodeling Methodology for Real-Time Simulation

Kangsun Lee
Software Lab., Corporate R & D Institute, SamSung Electronics, Seoul, Korea
and
Paul A. Fishwick
Dept. of Computer Information Science and Engineering, University of Florida, USA

When we build a model of real-time systems, we need ways of representing the knowledge about the system and also time requirements for simulating the model. Considering these different needs, our question is "How to determine the optimal model that simulates the system by a given deadline while still producing valid outputs at an acceptable level of detail?" We have designed OOPM/RT(Object-Oriented Physical Modeler for Real-Time Simulation) methodology. The OOPM/RT framework has three phases: 1) Generation of multimodels in OOPM using both structural and behavioral abstraction techniques, 2) Generation of AT (Abstraction Tree) which organizes the multimodels based on the abstraction relationship to facilitate the optimal model selection process, and 3) Selection of the optimal model which guarantees to deliver simulation results by the given amount of time. A more detailed model (low abstraction model) is selected when we have enough time to simulate, while a less detailed model (high abstraction model) is selected when the deadline is immediate. The basic idea of selection is to trade structural information for a faster runtime while minimizing the loss of behavioral information. We propose two possible approaches for the selection: an integer programming based-approach and a search-based approach. By systematically handling simulation deadlines while minimizing the modeler's interventions, OOPM/RT provides an efficient modeling environment for real-time systems.

Categories and Subject Descriptors: I.6.5 [**Simulation and Modeling**]: Model Development
General Terms: Modeling Methodology, Real-Time Simulation
Additional Key Words and Phrases: Model Abstraction, Model Selection, Real-Time Systems

1. INTRODUCTION

Real-time systems refer to systems that have hard real-time requirements for interacting with a human operator or other agents with similar time-scales. An efficient simulation of real-time systems requires a model that is accurate enough to accomplish the simulation objective and is computationally efficient [Garvey and Lesser 1993b; Garvey and Lesser 1993a; Lee and Fishwick 1998]. We define *model ac-*

Name: Kangsun Lee, ksl@swc.sec.samsung.co.kr
Name: Paul A. Fishwick, fishwick@cise.ufl.edu

This work is supported by (1) GRCI (Incorporated 1812-96-20 (Gregg Liming) and Rome Laboratory (Steve Farr, Al Sisti) for web-based simulation and multimodeling; (2) NASA/Jet Propulsion Laboratory 961427 (John Peterson and Bill McLaughlin) for web-based modeling of spacecraft and mission design, and (3) Department of the Interior under ATLSS Project contract 14-45-0009-1544-154 (Don DeAngelis, University of Miami) for techniques for both code and model integration for the across tropic level Everglades ecosystem.

curacy in terms of the ability of a model to capture the system at the right level of detail and to achieve the simulation objective within an allowable error bound. Computational efficiency involves the satisfaction of the real-time requirements to simulate the system, in addition to the efficiency of model computation. In existing applications, it is a user's responsibility to construct the model appropriate for the simulation task. This is a difficult, error-prone, and time-consuming activity requiring skilled and experienced engineers.

Most CASE tools [Digital 1989] try to help the modeling process by providing an extensive library of functions that allow a modeler to specify numerous aspects of an application's architecture. These tools deal with static models suitable for producing design documents, with limited facilities for simulating the models, analyzing the results of such simulations, running what-if questions, or translating the paper models to prototype code. However, these tools do not provide support for specifying the real-time constraints of an application's functions [Lark et al. 1990; Burns and Wellings 1994].

Our objective is to present a modeling methodology with which the real-time systems can be modeled efficiently to meet the given simulation objective and time requirements.

One of the contributions of our research is that, with the ability to select an optimal model for a given deadline, we provide a semi-automatic method to handle real-time constraints for a simulation. In particular, we handle a time constraint out of the modeling processes; therefore, modelers are relieved from considering constraints that are not supposed to be part of modeling. Another contribution is that by generating a set of multiple methods through abstraction techniques and selecting the optimal abstraction degree to compose a model for the real-time simulation, we meet not only the real-time constraints, but also the perspective which modelers see the system for a given time-constraint situation. We expect that the proposed method can provide better sources of multiple methods for real-time computing groups.

This paper is organized as follows : In Section 2, we discuss several related research activities. We propose the OOPM/RT modeling framework in Section 3. In Section 5, we present our abstraction methodology to generate a set of models for a system at different levels of detail. In Section 6, we show how to organize the models in a way that facilitates the selection process. Two optimal model selection algorithms are presented in Section 7. A complete process from model generation to the selection of the optimal abstraction level is illustrated in Section 8 through an example. We conclude in Section 9.

2. MODELING OF REAL-TIME SYSTEMS

Real-time systems differ from traditional data processing systems in that they are constrained by certain non-functional requirements (e.g., dependability and timing). Although real-time systems can be modeled using the standard structured design methods, these methods lack explicit support for expressing the real-time constraints [Kopetz et al. 1991; Lark et al. 1990; Burns and Wellings 1994]. Standard structured design methods incorporate a life cycle model in which the following activities are recognized:

(1) Requirements Definition - an authoritative specification of the system's required functional and non-functional behavior is produced.

(2) Architectural Design - a top-level description of the proposed system is developed.

(3) Detailed Design - the complete system design is specified.

(4) Coding - the system is implemented.

(5) Testing - the efficacy of the system is tested.

For hard real-time systems, this has the significant disadvantage that timing problems will be recognized only during testing, or worse after deployment [Burns and Wellings 1994]. Researchers have pointed out that the time requirements should be addressed in the design phase [Burns and Wellings 1994; Lark et al. 1990].

Two activities of the architectural design are defined [Burns and Wellings 1994]: 1) *the logical architecture design activity*, and 2) *the physical architecture design activity*. The logical architecture embodies commitments that can be made *independently* of the constraints imposed by the execution environment, and is primarily aimed at satisfying the *functional* requirements. The physical architecture takes these functional requirements and other constraints into account, and embraces the *non-functional* requirements. The physical architecture forms the basis for asserting that the application's non-functional requirements will be met once the detailed design and implementation have taken place. The physical architecture design activity addresses timing (e.g. responsiveness, orderliness, temporal predictability and temporal controllability) and dependability requirements (e.g., reliability, safety and security), and the necessary schedulability analysis that will ensure that the system once built will function correctly in both the value and time domains. Appropriate scheduling paradigms are often integrated to handle non-functional requirements [Burns and Wellings 1994].

The following issues arise :

—How to capture the logical aspects of the real-time systems?

—How to assess duration and quality associated with each model?

—How to resolve timing constraints?

—How to support both logical and physical activities under one modeling and simulation framework so that the resulting model is guaranteed to function correctly in both the value and time domains?

Several areas of research relate to these issues. Real-time scheduling focuses on how to deal with the physical requirements of the system. The main interest is to determine a schedule that defines *when to execute which task* to meet a deadline. Typical approaches to real-time scheduling assume that task priorities and resource needs are completely known in advance and are unrelated to those of other tasks, so that a control component can schedule tasks based on their individual characteristics. If more tasks exist than the system can process, the decision about which tasks to ignore is simple and local, usually based only on task priority [Ramamritham and Stankovic 1984; Stankovic et al. 1985]. The resulting schedule of tasks does not reflect the real objective of the simulation when the selection is made based solely on task priority.

Real-Time Artificial Intelligence studies problem-solving methods that "given a time bound, dynamically construct and execute a problem solving procedure which will (probably) produce a reasonable answer within (approximately) the time available." [D'Ambrosio 1989] Examples of this type are found in chess programs. Virtually all performance chess programs in existence today use full-width, fixed-depth, alpha-beta minimax search with node ordering, quiescence, and iterative-deepening for the real time problem solving. They make very high quality move decisions under real-time constraints by properly controlling the depth of search (or move) and having a good heuristic function that guides the search (or move). Research on the real-time problem solving through search methods can be found in Refs. [Korf 1990; Barr and Feigenbaum 1981].

The key to these approaches is to have a *single* problem solving method that achieves a better result as the method is given more time. One of the problems is that these approaches rely on the existence of iterative refinement algorithms that produce incrementally improving solutions as they are given increasing amounts of runtime. Clearly, such algorithms exist for some problem cases, but also there are problems that will be difficult to solve in an incremental fashion [Garvey and Lesser 1993b; Garvey and Lesser 1993a]. An alternative to this approach is to have *multiple* methods to model the system which make tradeoffs in cost versus quality, and may have different performance characteristics in different environment situations. Garvey and Lesser proposed the *Design-to-Time* [Garvey and Lesser 1993a; Garvey and Lesser 1995] method, which is related to our approach. *Design-to-time* assumes that one has multiple methods for the given tasks and tries to find a solution to a problem that uses all available resources to maximize solution quality within the available time. They present an algorithm for finding optimal solutions to a real-time problem under task tree graph and task relationships [Garvey and Lesser 1995]. The algorithm generates all sets of methods that can solve the problem and prunes those superseded by other sets of methods that generate greater or equal quality in equal or less time. *Design-to-time* has a single model type and multiple methods are generated through approximation techniques; therefore, it concerns only the behavioral aspects of the system. In order to model a complex system, it's better to have different model types to efficiently characterize the different aspects of the complex system.

3. OOPM/RT : A MODELING METHODOLOGY FOR REAL-TIME SIMULATION

We have built OOPM/RT(Object-Oriented Physical Modeler for Real-Time Simulation) for aiding the user to meet arbitrary time and quality constraints imposed upon the simulation. OOPM/RT adopts a philosophy of *rigorous engineering design*, an approach which requires the system model to guarantee the system's timeliness at *design* time [Lark et al. 1990]. OOPM/RT uses OOPM for the logical architecture design activity. OOPM [Fishwick 1997] is an approach to modeling and simulation which promises not only to tightly couple a model's human author into the evolving modeling and simulation process through an intuitive HCI (Human Computer Interface), but also to help a model author to perform any or all of the following objectives [Cubert and Fishwick 1998]:

—to think clearly about, to better understand, or to elucidate a model

—to participate in a collaborative modeling effort

—to repeatedly and painlessly refine a model with heterogeneous model types as required, in order to achieve adequate fidelity at minimal development cost

—to painlessly build large models out of existing working smaller models

—to start from a conceptual model which is intuitively clear to domain experts, and to unambiguously and automatically convert this to a simulation program

—to create or change a simulation program without being a programmer

—to perform simulation model execution and to present simulation results in a meaningful way so as to facilitate the prior objectives

By using OOPM for the sources of creating methods, we can model a system efficiently with different model types together under one structure. For time-critical systems, we may prefer models that produce less accurate results *within* an allowable time, over models that produce more accurate results *after* a given deadline. The key to our method is to use an abstraction technique as a way of handling real-time constraints given to the system. We generate a set of methods for the system at different levels of abstraction through a model abstraction methodology. When the constructed model cannot be executed for a different simulation condition, such as a tighter deadline, we change the abstraction degree of the model to deliver the simulation results by the given amount of time. A decision supporting tool is added to OOPM in order to take these constraints into account and determine a level of abstraction that satisfies both time and accuracy constraints. The decision process is placed out of the modeling process, therefore modelers are relieved from considering time constraints that are not supposed to be part of modeling.

OOPM/RT has three phases :

(1) Generating a set of models at different abstraction levels.

(2) Arranging a set of models under the abstraction relationship and assessing the quality/cost for each model

(3) Executing model selection algorithms to find the optimal model for a given deadline

In the first phase, a set of methods is generated at a different degree of detail using an abstraction methodology. The second phase is to assess the expected quality and runtime of each method and organizes a set of methods in a way to facilitate the selection process. When the constructed model cannot be simulated for a given amount of time, we select a level of abstraction that satisfies both time and accuracy constraints. In the third phase, we compose a model based on the determined level of abstraction. A more detailed model (low abstraction level) is selected when we have enough time, while a less detailed model (high abstraction level) is used when there is an imminent time constraint.

4. DISCUSSIONS

Several assumptions are made in OOPM/RT.

—Sacrificing solution quality can be tolerated: Systems can be modeled by multiple solution methods that produce tradeoffs in solution quality and execution time.

—Models are shared with different modelers: For a modeling task that involves intensive cooperation, keeping a model that is more complex than required is meaningful, even if the less complex model is valid. In the cooperative modeling environments, the concept of validity on the model varies among the cooperating developers; A less complex model that is valid to a modeler might be invalid to another modeler who needs more fidelity.

—Execution time of a method is fairly predictable: Many timing analysis techniques have been proposed, ranging from static, source-based methods to profilers and testing tools, through some combination thereof. These techniques find the worst-case paths of a program code and bound its execution time [Arnold et al. 1994; Lim et al. 1994; Marin et al. 1994; Wedde et al. 1994]. We do not propose a new timing analysis method in this paper. Instead, we assume that the execution time of a method is properly measured or predicted by these available techniques.

There are several research issues unresolved in OOPM/RT.

—Optimality: The optimality of the determined level of abstraction from OOPM/RT could only be guaranteed if we could also guarantee that we have completely decomposed our model specifications into the least complex nodes. Thus, the optimal level of abstraction is limited to how extensive a set of less complex models we have taken into account for the selection process. In this paper, we define an optimal level of abstraction as the level of abstraction that satisfies timing constraints with the minimum accuracy loss among the possible alternatives, and do not consider the optimality of model specifications.

—Validation: OOPM/RT methodology introduces a number of model validation challenges. In this paper, we assume that the errors induced and alleviated by model composition is up to a modeler and do not address the validation problems. For the studies on the validation and consistency problems, refer to Refs. [Davis and Hillestad 1993; Davis and Bigelow 1998; Kim 1998].

—Correlation: There are bound to be correlated effects between the various models defined in AT. Results from a model may decrease the quality of other models; Likewise, results from a model may increase the quality of other related models. Several studies have been done to estimate a model's quality under the existence of correlation effects [Rutledge 1995; Garvey and Lesser 1993b]. In this paper, we assess a model's quality in terms of 1) degree of abstraction, 2) degree of interest loss, and 3) degree of precision loss. Detailed descriptions on these are found in Section 6. However, we do not include correlation effects to assess model's quality. The correlation effects should be further investigated as a future work.

In the following Section 5 - 7, we explain each phase of OOPM/RT.

5. MODEL GENERATION

We generate a set of methods for the system by using model abstraction techniques. Model abstraction is a method (simplifying transformation) that derives a "simpler" model from a more complex model while maintaining the validity (consistency within some standard of comparison) of the simulation results with respect to the behaviors exhibited by the simpler model. The simpler model reduces the complexity as well as the quality of the model's behavior [Caughlin and Sisti 1997].

The proper use of abstraction provides computational savings as long as the validity of the simulation results is guaranteed. Our approach is to use the abstraction method when we need to reduce the simulation time to deliver the simulation results by a given deadline. A set of models are generated through abstraction techniques with different degrees of abstraction; each model simulates the system within a different amount of time, while describing the system with a different perspective. Therefore, a model that is selected for a given real-time simulation is useful not just because it meets a given deadline, but also because it suggests a perspective with which modelers view the system for a given time-constraint situation.

We have studied abstraction techniques available in many disciplines and created an unified taxonomy for model abstraction where the techniques are structured with the underlying characterization of a general approach [Lee and Fishwick 1996; Lee and Fishwick 1997b]. Our premise is that there are two different approaches to model abstraction: *structural* and *behavioral*. *Structural abstraction* is the process of abstracting a system in terms of the structure using refinement and homomorphism [Zeigler 1976; Zeigler 1990]. Structural abstraction provides a well-organized abstraction hierarchy on the system, while behavioral abstraction focuses only on behavioral aspects of the system without structural preservation. We organize the system hierarchically through the structural abstraction phase and construct an abstraction hierarchy with simple model types first, refining them with more complex model types later. Our structural abstraction provides a way of structuring different model types together under one framework so that each type performs its part, and the behavior is preserved as levels are mapped [Fishwick 1995]. The resulting structure becomes a *base model* which has the highest resolution to model the system. The problem is that selecting one system component from an abstraction level is dependent on the next lowest level due to the hierarchical structure. Each component cannot be *executed* at a random abstraction level, though it can be *viewed* independently. *Behavioral abstraction* is used when we want to simulate the base model at a random abstraction level. We isolate an abstraction level by approximating the lower levels behavior and replacing them with a behavioral abstraction method. The method discards structural information of the lower levels, but the behavioral information is preserved in some level of fidelity. Possible techniques for behavioral abstraction are system identification, neural networks, and wavelets [Masters 1995]. Since our method involves less computation time by discarding the structural information of lower levels, it will be used when there is too little time to simulate the system in detail. The abstraction mechanism is implemented in OOPM, and the models produced by OOPM become the sources of methods for performing the real-time simulation. More detailed discussions on our abstraction methodology are found in Refs.[Lee and Fishwick 1996; Lee and Fishwick 1997b; Lee and Fishwick 1997a].

6. CONSTRUCTION OF THE ABSTRACTION TREE

AT (Abstraction Tree) extends the tree structure to represent 1) all the methods that comprise the base model and 2) the refinement/homomorphism relationship among the methods. Every method that comprises the base model is represented as a **node**. Each node takes one of three types : M_i, A_i or I_i.

—M_i - High resolution method. It takes the form of dynamic or static methods of OOPM. We have FBM (Functional Block Model), FSM (Finite State Machine), SD (System Dynamics), EQM (EQuational Model), and RBM (Rule-Based Model) choices for the dynamic method, and the CODE method for the static method.

—A_i - Low resolution method. It takes the form of a neural network or a BJ (Box-Jenkins) ARMA (AutoRegressive Moving Average) model.

—I_i - Intermediate node to connect two different resolution methods, M_i and A_i. I_i appears where a method i has been applied to behavioral abstraction, and the corresponding behavioral abstraction method has been generated for a low resolution method to speedup the simulation.

The Refinement/Homomorphism relationship is represented as an **edge**. If a method M_i is refined into $N_1, N_2, N_3, ..., N_k$, an edge(M_i, N_j), for $j = 1, 2, ..., k$, is added to the AT. *AND/OR* information is added on the edge to represent how to execute M_i for a given submethod N_j, for $j = 1, 2, ..., k$.

—AND - M_i is executed only if N_j is executed, $\forall\ j$, $j = 1, 2, ..., k$

—OR - M_i is executed only if any N_j, $j = 1, 2, ..., k$, is executed

The decision of AND/OR is made based on 1) the node type of M_i and 2) the model type of M_i.

(1) Node type : An intermediate node I_i is executed either by M_i or A_i, where M_i is a high resolution method and A_i is the corresponding low resolution method. Therefore, I_i is connected with the *OR* relationship.

(2) Model type : If a method M_i takes the form of an FBM, and each block that comprises M_i is refined into $B_1, B_2, ..., B_k$, then the execution of M_i is completed when B_j, $\forall\ j$, $j = 1, 2, ..., k$, are executed. Therefore, an FBM method M_i is connected with the *AND* relationship. Other examples of the *AND* relationship are SD, EQM, and CODE method. However, other model types can take the *OR* relationship. If a method M_i takes the form of an FSM, and each state of the FSM is refined into $S_1, S_2, ..., S_k$, then the execution of M_i is completed when any S_j, $j = 1, 2, ..., k$, is executed. The decision of j is made according to the predicate that the FSM method, M_i, satisfies at time t. Therefore, FSM method M_i is executed with the *OR* relationship. RBM is another example of the OR relationship.

Let T represent M_i, A_i, and I_i. Each node, T, in AT has **duration** $D(T)$ and **quality** $Q(T)$. $Q(T)$ summarizes three properties of the quality associated with node T.

(1) QA(T) - Degree of abstraction. Degree of abstraction represents how much information would be lost if the execution occurs at node T. The base model will not be executed at the associated leaf nodes when the method T is selected for the behavioral abstraction. QA(T) is defined by how many methods are being discarded if behavioral abstraction occurs at node T, comparing to the case where no behavioral abstraction is applied to the base model.

(2) QI(T) - Degree of interest loss. A modeler may have certain nodes that he/she wants to see with a special interest; If there are two nodes, A_1 and A_2 in a given AT, and a modeler has a special interest in A_1, it is preferable to take A_2 for behavioral abstraction. QI(T) is defined by how many interesting nodes are being discarded if behavioral abstraction occurs at node T, comparing to the case where no behavioral abstraction is applied to node T.

(3) QP(T) - Degree of precision loss. Degree of precision loss represents how accurately the behavioral abstraction method approximates the high resolution method for node T. The precision can be assessed by testing the trained neural network or Box-Jenkin's ARMA model. Several techniques for estimating error rates have been developed in the fields of statistics and pattern recognition, which include *hold out, leave one out, cross validation,* and *bootstrapping* [Weiss and Kapouleas 1989]. We use *holdout* method which is a single train-and-test experiment where a data set is broken down into two disjoint subsets, one used for training and the other for testing. QP(T) is estimated through the testing phase of *holdout.*

Based on the three quality properties, Q(T) is defined by :

$$\begin{aligned} QA(T) &= \frac{N(T)}{N} \\ QI(T) &= \frac{N_i(T)}{N_i} \\ QP(T) &= E(T) \\ Q(T) &= \frac{QA(T) + QI(T) + QP(T)}{q} \end{aligned}$$

where N(T) for the number of nodes in a subtree that has T as a root node, N_i(T) for the number of interesting nodes in a subtree that has T as a root node, N_i for the total number of interesting nodes in a given AT, and N for the total number of node in a given AT. E(T) is the normalized error rate of behavioral abstraction method for node T. $N(T)$ of QA(T) is set to 1 for a leaf node. q represents the number of quality properties specified for a node, T ($1 \leq q \leq 3$). For an intermediate node, I_i

$$Q(I_i) = Q(M_i) - Q(A_i)$$

where M_i is the high resolution method for node I_i, while A_i is the low resolution method for node I_i.

$D(T)$ is defined based on 1) AND/OR relationship and 2) node types. For a node type, I_i, and the corresponding two different resolution methods, M_i and A_i,

$$D(I_i) = D(M_i) - D(A_i) + \theta$$

where θ is the *system factor*; θ considers if the target simulation platform is different from the environment in which the execution time has been measured. Therefore,

$D(I_i)$ represents the amount of speedup by replacing the high resolution model M_i with the behavioral abstraction model A_i, in any platform. For other node types, the duration of a node is defined based on its AND/OR relationship. For an AND related node,

$$D(T) = \sum_{j=1}^{k} N_j + \delta(T) + \theta$$

For an OR related node,

$$D(T) = Max\ (N_1, N_2, ..., N_k) + \delta(T) + \theta$$

where N_j, for $j = 1, ..., k$, is the method that T calls to complete its high resolution execution. k is the number of refined methods for T. $\delta(T)$ is the amount of time that method T takes for its own execution. For example, in the case of an FSM, checking the current state and determining the next state based on the predicates might take $\delta(T)$ time, while the execution of each state is assessed in the summation term. θ is a system factor as in the case of I_i node. D(T) of an OR node is set to the worst case execution time by taking the maximum duration of the possibilities. This worst case assignment securely guarantees the resulting model's timeliness. The quality and the duration function are constructed recursively, until individual methods at the leaf level are reached. We assume that the duration of each leaf node is measured by available worst-case timing analysis techniques. The program code in leaf nodes is simple in most cases, since modelers use dynamic methods (non-leaf nodes in AT) instead to represent complex aspects of the system. Using the available techniques is practical for simple program codes, such as the codes found in the leaf nodes of AT.

Figure 1 shows an example of an AT. M_{21} is an AND related method that calls M_{31} and then M_{32} followed by M_{33}. M_{22} is an AND related method that calls M_{34} and then M_{35}. A method M_{11} calls M_{21} and then M_{22}. We suppose that behavioral abstraction methods $\{A_{11}, A_{12}, ..., A_{35}\}$ have been generated for each of the corresponding high level method M_{ij}. Each A_{ij} may take different model type according to the behavioral abstraction technique. The intermediate nodes relate a high resolution method to a corresponding behavioral abstraction. These nodes are symbolized by $\{I_{11}, I_{12}, ..., I_{35}\}$. If all $\{I_{11}, I_{12}, ..., I_{35}\}$ are executed by high resolution methods only, the resulting structure is the base model which has been constructed through the structural abstraction process. The quality and duration of each node is determined by recursively applying the quality and duration equations. δ and θ are assumed to be 0 for each internal node.

7. SELECTION OF THE OPTIMAL ABSTRACTION MODEL

A *Task* is a granule of computation treated by the scheduler as a unit of work to be allocated processor time [Liu and Chingand 1991]. The scheduling problem is to select a task set each of which meets the given deadline. Partial orderings of the tasks should be met in the resulting schedule. Related research has shown the NP-completeness of this problem [Liu and Chingand 1991; Garey and Johnson 1979]. *Task* corresponds to *method* that comprises the base model, which has been

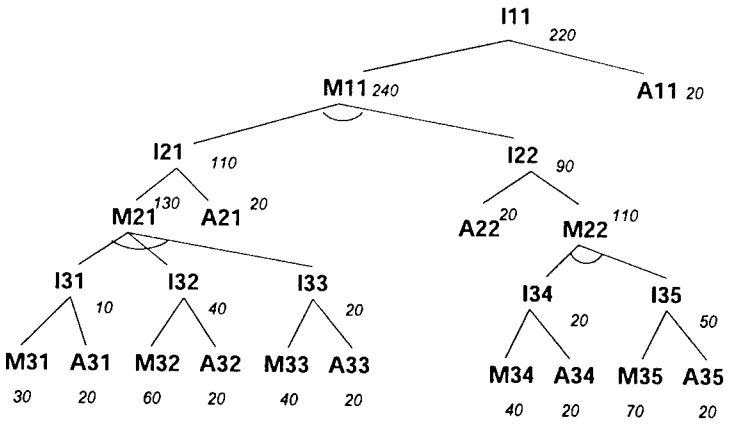


Fig. 1. Example Abstraction Tree

constructed from the model generation phase. The problem of finding the optimal abstraction level translates to the scheduling problem, which is to find a schedule for a set of methods that yields maximal quality for a given amount of time, while preserving the partial temporal orderings among the given methods. In the following sections, we define three approaches for optimal abstraction level selection. An optimal abstraction model is built based on the determined optimal abstraction degree.

7.1 IP (Integer Programming)- Based Selection

Operations research (OR) [Ravindran et al. 1987; Ragsdale 1998; R Fourer and Kernighan 1993] is a field that is concerned with deciding how to best design and operate systems under conditions requiring the allocation of scarce resources. Our optimal level selection problem falls under the OR umbrella, since the selection should be made for the best model that has an optimal abstraction level to simulate a given system under conditions requiring the allocation of scarce resources, such as time and accuracy. The essence of the OR activity lies in the construction and use of the mathematical models. The term *linear programming* defines a particular class of programming problems when the problem is defined by a linear function of the decision variables (referred to as the *objective function*), and the operating rules governing the process can be expressed as a set of linear equations or linear inequalities (referred to as the *constraint set*). IP refers to the class of linear programming problems wherein some or all of the decision variables are restricted to integers. *Pure integer programming* is a category where all the decision variables are restricted to be integer values. Our problem is a special case of integer programming, where the decision variables take *binary values* to indicate whether or not to select the examining node of a given AT for the behavioral abstraction. We formulate the problem as two IP models: *IP1* and *IP2*. A simple example is given to analyze the IP models.

7.1.1 *Formulation of IP-Based Selection.* Let a binary integer variable L_{ij} denote the decision to select or not to select the node I_{ij} for the behavioral abstraction.

$$L_{ij} = \begin{cases} 1 \text{ if behavioral abstraction occurs at the } I_{ij} \text{ node} \\ 0 \text{ otherwise} \end{cases}$$

Then, the objective function for *IP1* selection is defined in equation 1

$$Minimize \sum_{i=1}^{l} \sum_{j=1}^{n_i} L_{ij} \tag{1}$$

subject to

$$\sum_{i=1}^{l} \sum_{j=1}^{n_i} a_{ij} L_{ij} \leq a_c \tag{2}$$

$$\sum_{i=1}^{l} \sum_{j=1}^{n_i} t_{ij} L_{ij} \geq t_c \tag{3}$$

and, for each parent node L_{ik} of a given AT

$$\sum_{k=1}^{n_i} L_{i+1,k} \leq n_i(1 - L_{ik}), \text{ for each } i \tag{4}$$

where l is the maximum level of the AT and n_i is the number of nodes at level i. a_{ij} represents the accuracy loss and t_{ij} represents the expected duration. a_c defines the accuracy constraint given to the system, while t_c is the amount of desired speedup to meet a given deadline, D. Therefore, t_c = *execution time of the base model* - D.

The objective function of *IP1* reflects the fact that the smaller number of behavioral abstraction methods is desirable as long as the resulting model satisfies the given time deadline and the accuracy constraint.

The objective function for *IP2* selection is defined in equation 5

$$Minimize \sum_{i=1}^{l} \sum_{j=1}^{n_i} a_{ij} L_{ij} \tag{5}$$

subject to

$$\sum_{i=1}^{l} \sum_{j=1}^{n_i} t_{ij} L_{ij} \geq t_c \tag{6}$$

and, for each parent node L_{ik} of a given AT

$$\sum_{k=1}^{n_i} L_{i+1,k} \leq n_i(1 - L_{ik}), \text{ for each } i \tag{7}$$

The objective function of *IP2* minimizes the quality loss while satisfying the timing constraint defined in equation 6. *IP2* does not minimize the number of

behavioral abstraction methods as long as the resulting model minimizes the accuracy loss. For instance, if a model, $\mathcal{A}$, that cuts out three structural components, is expected to produce more accurate results than a model, $\mathcal{B}$, that cuts out only one structural component , *IP2* keeps $\mathcal{A}$ for a candidate of the optimal abstraction model.

7.1.2 *Analysis of IP-Based Selection.* Consider AT in Figure 1. We associate each I_{ij} node with a binary variable L_{ij} discussed in the previous section. Then the objective function of *IP1* for a given AT is defined as :

$$Minimize\ (L_{11} + L_{21} + L_{22} + L_{31} + L_{32} + L_{33} + L_{34} + L_{35}) \tag{8}$$

For simplicity, we assume that any behavioral abstraction method takes 20 units. θ is assumed to be 0, which means real-time simulation will be performed in the same platform with which the expected duration has been measured. Then, (t_{11}, t_{21}, t_{22}, t_{31}, t_{32}, t_{33}, t_{34}, t_{35}) is defined as (220, 110, 90, 10, 40, 20, 20, 50), respectively.

For the simplicity of illustration, we consider only QA(I_{ij}) to assess the quality loss, a_{ij}, when the corresponding node I_{ij} is selected for the behavioral abstraction. Then, a_{ij} is simplified as follows:

$$a_{ij} = \frac{C_{ij} + \sum_{k=1}^{ic} C_{i+1,k}}{N}$$

where C_{ij} is the number of children that L_{ij} has. Since the behavioral abstraction at this level discards all the structural information of the lower levels, we believe that the accuracy loss is proportional to the number of descendants that a node has. The right hand side of a_{ij} is to find out the number of descendants that node L_{ij} has. For a given AT in Figure 1, (a_{11}, a_{21}, a_{22}, a_{31}, a_{32}, a_{33}, a_{34}, a_{35}) is defined as (7/8, 3/8/, 2/8, 1/8, 1/8, 1/8, 1/8, 1/8), respectively.

For a given accuracy loss, a_c, the accuracy constraint of *IP1* is defined as :

$$(a_{11}L_{11} + a_{21}L_{21} + a_{22}L_{22} + a_{31}L_{31} + a_{32}L_{32} + a_{33}L_{33} + a_{34}L_{34} + a_{35}L_{35}) \leq a_c \tag{9}$$

Also, for a given deadline, t_c, the speedup constraint is defined as :

$$(t_{11}L_{11} + t_{21}L_{21} + t_{22}L_{22} + t_{31}L_{31} + t_{32}L_{32} + a_{33}L_{33} + a_{34}L_{34} + a_{35}L_{35}) \geq t_c \tag{10}$$

To define *parent and child* relationships in the given AT, we have a set of equations for all the child nodes of the given AT.

$$\begin{array}{ll} L_{31} + L_{32} + L_{33} & \geq 3(1 - L_{21}) \\ L_{34} + L_{35} & \geq 2(1 - L_{22}) \\ L_{21} + L_{22} & \geq 2(1 - L_{11}) \end{array} \tag{11}$$

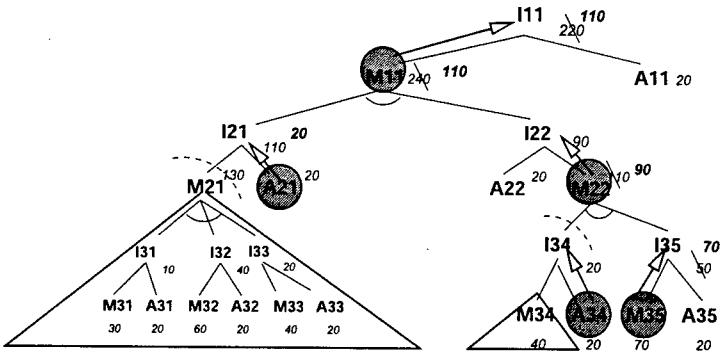


Fig. 2. Optimal abstraction tree decided from the integer programming approach

Then, the IP1 selection of the optimal abstraction level for a given AT is to solve the objective function defined in equation 8 subject to the constraints defined in equations 9- 11.

Figure 2 shows the result when we have 120 units for a deadline ($t_c = 120$) and (50%) for an accuracy constraint ($a_c = 0.5$). The resulting AT concludes that applying behavioral abstraction at I_{21} and I_{34} gives the optimal abstraction level which simulates the system within a given deadline and the accuracy constraint with the minimum loss of the structural information. I_{21} is executed by its behavioral abstraction method, A_{21}. Also, I_{34} is executed by its behavioral abstraction method, A_{34}, instead of the high resolution method, M_{34}. Other intermediate nodes (I_{35}, I_{22}, I_{11}) are executed by the high resolution methods. The intermediate nodes, I_{31}, I_{32}, and I_{33} are not considered , since the behavioral abstraction occurs at the parent node, I_{21}, as defined in equation 11. The execution time of the AT is saved by the speedup amount that A_{21} and A_{34} yield. The execution of *IP2* is same to *IP1* with the objective function and the time constraint defined as :

$$Minimize(a_{11}L_{11} + a_{21}L_{21} + a_{22}L_{22} + a_{31}L_{31} + a_{32}L_{32} + a_{33}L_{33} + a_{34}L_{34} + a_{35}L_{35})$$

subject to

$$(t_{11}L_{11} + t_{21}L_{21} + t_{22}L_{22} + t_{31}L_{31} + t_{32}L_{32} + a_{33}L_{33} + a_{34}L_{34} + a_{35}L_{35}) \geq t_c$$

7.2 Search Based Selection

Our hypothesis is that *as we need tighter time constraints, we tend to employ more behavioral abstraction methods.* We increase the number of behavioral abstraction methods as we require more stringent deadlines for the heuristics. The selection algorithm starts from one behavioral abstraction. If this abstraction satisfies the time constraint, we stop and do not go further to examine other possibilities, with the hope that increasing the number of behavioral abstraction methods will result only in a less accurate model. If the time cannot be met by one behavioral abstraction method, we examine how many behavioral abstraction methods will be needed for a given time constraint. This is done by examining r fast behavioral abstraction methods. If combining r fast behavioral abstraction methods satisfies the time

Algorithm 1. Optimal Abstraction Level Selection

```
1:  nodes ⇐ at.ConstructAT(fid)
2:  baseCost ⇐ nodes[0].getCost()
3:  if baseCost ≤ deadline then
4:    return(0)
5:  endif
6:  at.CollectOrNodes(nodes,orNodes)
7:  size ⇐ at.SelectOneByDeadline(orNodes,deadline)
8:  if size > 0 then
9:    at.SelectByAccuracy(orNodes)
10: OptimalAbstraction ⇐ orNodes[0].getName()
11: else
12:   degree ⇐ OptimalAbsNumber(orNodes,deadline,baseCost)
13:   if degree == -1 then
14:     return(-1)
15:   else
16:     OptimalSet ⇐ OptimalCombination(ornodes,deadline,baseCost,degree)
17:     OptimalCost ⇐ OptimalSet.cost
18:     OptimalQualityLoss ⇐ OptimalSet.qualityloss
19:   end if
20: end if
21: return(0)
```

constraint, then the optimal abstraction level will be determined by r behavioral abstraction methods. At this point, we start to pick r behavioral abstraction functions until the most accurate combination is found while still satisfying the given time constraint. Algorithm 1 shows the overall method in detail.

This algorithm reads abstraction information about a given base model. The information contains methods, parent/child relationships between the nodes, and duration/quality information for each method. Based on the given information, the algorithm constructs an AT as in Line 1. The execution time of the base model that has no behavioral abstraction methods is calculated in Line 2. Lines 3 - 5 examine whether we need to employ behavioral abstraction methods to meet a given deadline. If the calculated duration of the base model is less than or equal to the given deadline, we don't have to employ behavioral abstraction methods; thus, the algorithm terminates. If the duration of the base model is greater than the given deadline, then it becomes necessary to use behavioral abstraction methods. Upon recognizing the necessity of behavioral abstraction methods, the algorithm collects *OR* nodes that contain the information about the behavioral abstraction methods and then starts to increase the number of behavioral abstraction methods. Line 8 examines whether one behavioral abstraction method will resolve the timeliness requirement. If the returned *size* is greater than 0, as in Line 9, we know that *one* behavioral abstraction is enough to meet a given deadline. Then, the algorithm looks up the most accurate method to ensure that the selected method will have the best quality while satisfying the given deadline. If one behavioral abstraction is not enough to achieve the given deadline ($size \leq 0$), the algorithm determines how many behavioral abstraction methods can achieve the given deadline. We know that behavioral abstraction methods will bring more savings to the execution time of the resulting model. Our objective is to minimize the use of behavioral abstraction

methods as long as the resulting model meets the given deadline. A simple method defining *degree* in line 12 is :

(1) $i = 2$

(2) select i behavioral methods that will bring the maximum time savings to the given base model;

(3) if the use of i behavioral methods cannot resolve the required speedup, increase i by 1 and go to step 2;

At this point, the algorithm knows how many behavioral abstraction methods will be needed for a given deadline. If the returned *degree* is -1, it means the given deadline cannot be met even if we use all available behavioral abstraction methods. Lines 16 - 18 look for the best combination that will lead to the most accurate model. The algorithm examines all nCr combinations, where n represents the number of behavioral abstractions available to the given base model and r is the calculated *degree* in Line 13.

7.3 Experiments

We implemented the proposed integer programming solutions with *solver* on Excel. For a small problem space as in the case of Figure 1, *solver* of Excel might be a good choice. However, if the problem size is large, we can use CPLEX [CPLEX 1995] which is an optimization callable library designed for large scale problems.

For the exact solution method, we use the *branch and bound method*, which is a classical approach to find exact integer solutions. The *branch and bound method* is an efficient enumeration procedure for examining all possible integer feasible solutions [Ravindran et al. 1987; CPLEX 1995]. Through the *branch and bound method*, the binary variable L_{ij} takes either 0 or 1.

Table 1 shows some results from the experiments of *IP1, IP2* and search-based approach. *IP1* and *Search* try to maintain the base model's structure (i.e. dynamic methods) as long as the given speedup amount is achieved. *IP2* does not try to maintain the base model's structure as long as the overall accuracy loss is minimized with the desired speedup amount. L_{ij} nodes indicate where the behavioral abstraction methods should be employed to meet both a time constraint, t_c, and an accuracy constraint, a_c. The actual accuracy loss and speedup amount from *IP1, IP2* and *Search* selection are listed in Table 1 along with the L_{ij} results.

When the base model meets a given deadline, as in case 1, no behavioral abstraction is suggested. Also, when a given deadline is immediate, the entire AT is behaviorally abstracted into one method as in case 12. L_{11} is selected as a place to apply the behavioral abstraction for a given deadline of 20. Other cases employ one or two behavioral abstraction methods to meet a given accuracy constraint while satisfying a time constraint. Note that *IP2* selects equal or more number of nodes comparing to *IP1*. Case 7 shows that the objective of *IP1* is to meet both accuracy and time constraint, while minimizing the loss of structural abstraction methods. Case 11 shows that the objective of *IP2* is to minimize the expected quality loss rather than to minimize the loss of structural abstraction methods.

Given an optimal abstraction level determined by the selection algorithms, the *Optimal Abstraction Model Composer* looks at the method names which comprise

Table 1. Experiment results from *IP1*, *IP2* and search-based approach: L_{ij} indicates where to employ behavioral abstraction method(s), A_{ij}, for a given simulation deadline and an accuracy constraint, a_c. The actual accuracy loss and speedup amount achieved from the selection are listed along with L_{ij} results

No	Deadline	a_c	t_c	*IP1*	*IP2*	Search
1	240	0.5(50%)	0	n/a	n/a	n/a
2	220	0.5(50%)	20	L_{34} a_c = 0.125(13%) deadline = 220	L_{34} a_c = 0.125(13%) deadline = 220	L_{34} a_c = 0.125(13%) deadline = 220
3	200	0.5(50%)	40	L_{35} a_c = 0.125(13%) deadline = 190	L_{35} a_c =0.125(13%) deadline = 190	L_{35} a_c =0.125(13%) deadline = 190
4	180	0.5(50%)	60	L_{22} a_c = 0.25(25%) deadline = 150	L_{32}, L_{34} a_c =0.25(25%) deadline = 180	L_{22} a_c = 0.25(25%) deadline = 150
5	160	0.5(50%)	80	L_{22} a_c = 0.25(25%) deadline = 150	L_{22} a_c = 0.25(25%) deadline = 150	L_{22} a_c = 0.25(25%) deadline = 150
6	140	0.5(50%)	100	L_{21} a_c = 0.375(38%) deadline = 130	L_{22}, L_{33}, L_{35} a_c = 0.375(38%) deadline = 80	L_{21} a_c = 0.375(38%) deadline = 130
7	120	0.5(50%)	120	L_{21}, L_{34} a_c = 0.5(50%) deadline = 110	L_{22}, L_{32} a_c = 0.375(38%) deadline = 110	L_{22}, L_{32} a_c = 0.375(38%) deadline = 110
8	100	0.5(50%)	140	L_{21}, L_{35} a_c = 0.5(50%) deadline = 80	L_{22}, L_{32}, L_{33} a_c = 0.5(50%) deadline = 90	L_{21}, L_{35} a_c = 0.5(50%) deadline = 80
9	80	0.5(50%)	160	L_{21}, L_{35} a_c = 0.5(50%) deadline = 80	L_{21}, L_{35} a_c = 0.5(50%) deadline = 80	L_{21}, L_{35} a_c = 0.5(50%) deadline = 80
10	60	0.7(70%)	180	L_{21}, L_{22} a_c = 0.625(63%) deadline = 40	L_{21}, L_{22} a_c = 0.625(63%) deadline = 40	L_{21}, L_{22} a_c = 0.625(63%) deadline = 40
11	40	0.9(90%)	200	L_{11} a_c = 0.875(88%) deadline = 20	L_{21}, L_{22} a_c = 0.625(63%) deadline = 40	L_{11} a_c = 0.875(88%) deadline = 20
12	20	0.9(90%)	220	L_{11} a_c = 0.875(88%) deadline = 20	L_{11} a_c = 0.875(88%) deadline = 20	L_{11} a_c = 0.875(88%) deadline = 20

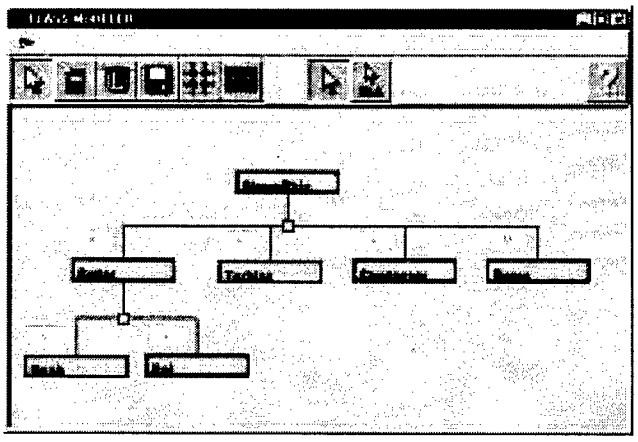


Fig. 3. Conceptual model of FULTON: FULTON is modeled within OOPM Conceptual model is designed in terms of classes, attributes, methods (dynamic method and static method) and relationships of classes (inheritance and composition)

the optimal abstraction level for a given AT. The optimal abstraction model is composed by observing the partial temporal orderings of the selected methods.

8. FULTON EXAMPLE

Consider a steam-powered propulsion ship model, named FULTON. In FULTON, the furnace heats water in a boiler: when the fuel valve is OPEN, fuel flows and the furnace heats; when the fuel valve is CLOSED, no fuel flows and the furnace is at ambient temperature. Heat from the furnace is added to the water to form high-pressure steam. The high-pressure steam enters the turbine and performs work by expanding against the turbine blades. After the high-pressure steam is exhausted in the turbine, it enters the condenser and is condensed again into liquid by circulating sea water [Gettys and Keller 1989]. At that point, the water can be pumped back to the boiler.

8.1 Model Generation

A conceptual model is constructed on OOPM. It is designed in terms of classes, attributes, methods, and relationships between classes (inheritance and composition). Figure 3 shows the class hierarchy of FULTON, which follows the physical composition of a steamship. Classes are connected by a *composition* relationship as denoted by the rectangular boxes in Figure 3. *V* denoted in the white box specifies 1 for the cardinality of the associated class. In Figure 3, *Ship* **has** *Boiler*, *Turbine*, *Condenser*, and *Pump*. Class *Boiler* **has** *Pot* and *Knob*. Each class has attributes and methods to specify its dynamic behaviors.

8.1.1 *Structural Abstraction of FULTON.* Figures 4, 5, and 6 show structural abstractions of FULTON. Since FULTON can be configured with 4 distinct physical components and a functional directionality, we start with FBM. The FBM is located in class *Ship*, as shown in Figure 4. Figure 4 has 4 blocks: L_1 for the function of

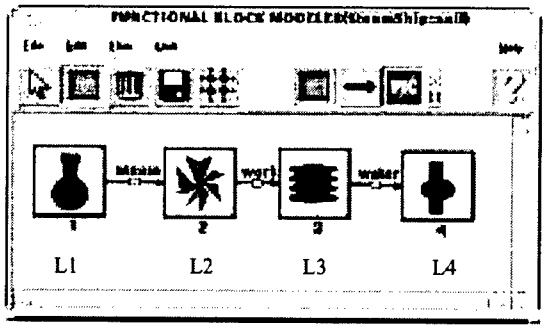


Fig. 4. Top level : structural abstraction for FULTON

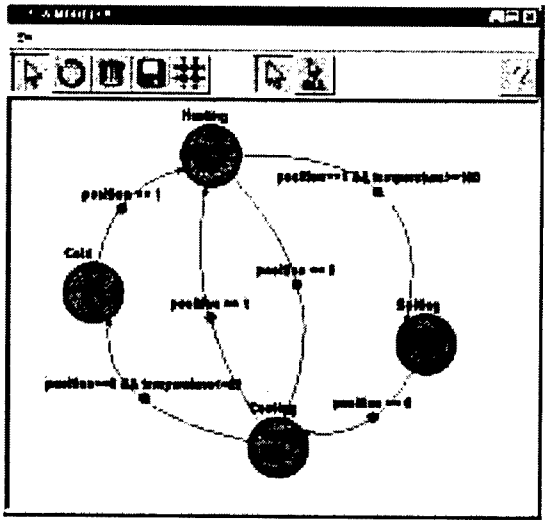


Fig. 5. Structural abstraction of M_7: FSM has 4 states (Cold ($M13$), Cooling ($M14$), Heating (M15) and Boiling (M16))

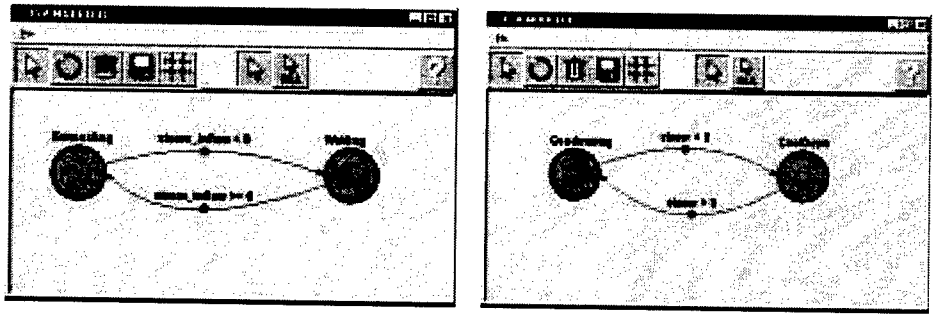
Boiler, L_2 for *Turbine*, L_3 for *Condenser*, and L_4 for *Pump*. *Boiler* assembly (L_1) has distinct states according to the temperature of the water. L_1 is refined into :

(1) B_1 : method of class *Knob*, $M6$, which provides fuel to the boiler

(2) B_2 : FSM, M_7, in Figure 5, which determines the temperature of the boiler and makes state transitions according to the temperature

(3) B_3 : provides high-pressure steam, defined in a CODE method, $M8$

Each state of Figure 5 (Cold (M13), Heating (M14), Boiling (M15), and Cooling (M16)) is refined into an algebraic equation, which calculates the temperature based on the position of the knob (Open, Closed). Each state of B_2 is refined into a *CODE* method that defines the temperature equations with C++ syntax.



(a) Structural abstraction of M_{10} (b) Structural abstraction of M_{12}

Fig. 6. Structural abstraction of M_{10} and M_{12}

L_2 is refined into two functional blocks: $M9$ and $M10$. $M9$ gets the high pressure steam from the boiler. $M10$ is decomposed into two temporal phases: *Exhausting* ($M17$) and *Waiting* ($M18$). If there is no steam to exhaust, $M10$ resides in the waiting state. Otherwise, $M10$ exhausts steam to generate the work of the steamship. Figure 6 shows the FSM of the turbine. L_3 is also refined into two functional blocks: $M11$ and $M12$. $M11$ gets the exhausted steam from the turbine. $M12$ has two distinct temporal phases: *Condensing* ($M19$) and *Cooldown* ($M20$), in Figure 6. *Condenser* decreases the temperature in *Cooldown* state, waiting for the turbine to send more steam. Otherwise, $M12$ resides in *Condensing* state where the steam from the turbine turns into liquid again.

8.1.2 *Behavioral Abstraction of FULTON.* We start with the observed data set of (input, output) from the simulation of the base model. With this *prior* knowledge, the method of behavioral abstraction is to generate a C++ procedure which encodes the input/output functional relationship using a neural network model (MADALINE, Backpropagation) or a Box-Jenkins model.

We abstract the multimodel method of $M7$, $M10$ and $M12$ with Box-Jenkins models. Given three behavioral abstraction methods for $M7, M10$ and $M12$, 8 (2^3) new models can be generated with different degrees of abstraction. Table. 2 shows the possible combinations of the behavioral abstraction methods. For example, $C6$ uses two behavioral abstraction methods for $M7$ and $M12$. Therefore, the structural information associated with $M7$ and $M12$ methods, which are both FSMs, are abstracted.

When modelers create a behavioral abstraction method, they pick the dynamic function to abstract. Figure 7 shows the Box-Jenkins abstraction process for $M7$. Based on the learning parameters of the Box-Jenkins, we learn the input/output functional relationship of M7. Once the performance of the Box-Jenkins model is accurate enough, we generate a behavioral abstraction method based on the resulting weight and offset vector.

Table 2. New multimodels of FULTON with behaviorally abstracted component(s): A capital letter represents a full-resolution model, while a small letter represents a low-resolution model. The low resolution model is generated through behavioral abstraction

no	Model	Abstracted Method	Abstracted Component
C_1	BTCP	N/A	N/A
C_2	BTcP	$M12$	Condenser
C_3	BtCP	$M10$	Turbine
C_4	BtcP	$M10$, $M12$	Turbine, Condenser
C_5	bTCP	$M7$	Boiler
C_6	bTcP	$M7$, $M12$	Boiler, Condenser
C_7	btCP	$M7$, $M10$,	Boiler, Turbine
C_8	btcP	$M7$, $M10$, $M12$	Boiler, Turbine, Condenser

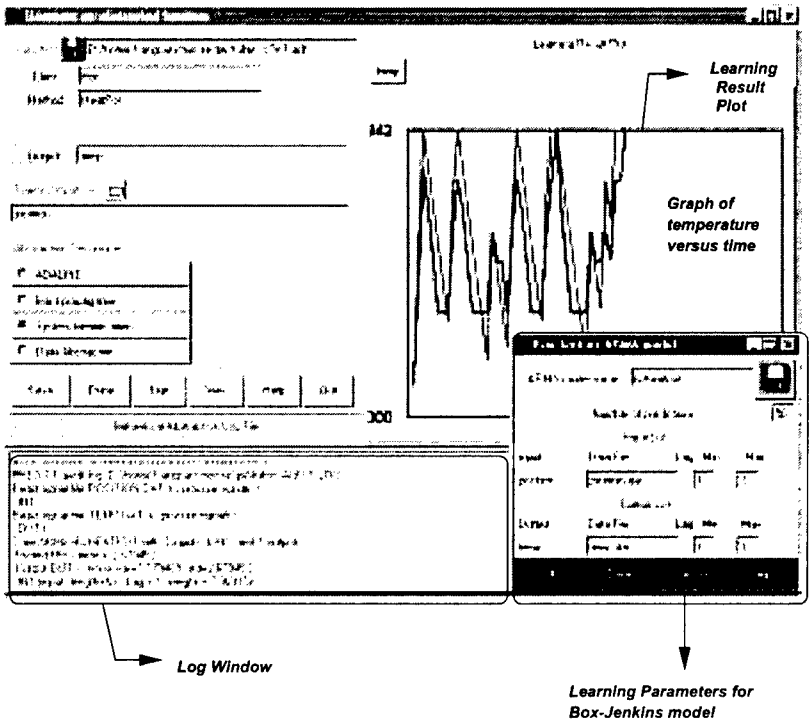
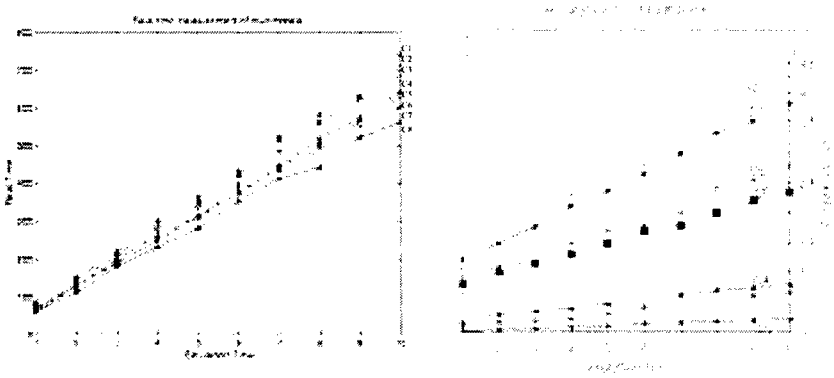


Fig. 7. Behavioral abstraction : The user selects a dynamic function to abstract, Pot::HeatPot, which is an FSM. States and their transitions will be lost, but behavioral information will be preserved to some level of fidelity. In the learning process, the user gives several parameters, for example, *lag* for input and output variables



(a) Execution time (msec($\frac{1}{1000}$)) of multimodels. (measured on Windows NT, x86, Intel 133 MHZ)

(b) Accuracy loss of multimodels: relative accuracy loss to the most detailed model, C_1. (measured by the sum of the squared error)

Fig. 8. Execution time/ accuracy loss of models: Behavioral abstraction yields shorter elapsed times. As we increase the level of abstraction by using more behavioral abstraction methods, the model execution time decreases. Accuracy is lost as we increase the number of behavioral abstraction methods in the base model

8.2 Construction of the Abstraction Tree

The model sets in Table 2 provide alternatives that vary in the expected accuracy and in the computation time; it thus allows us to investigate the effects of model structure on model accuracy and on computation time. We measured the execution time of each model by varying the simulation logical clock from 50 to 500 using a time step of 50. As shown in Figure 8, the most detailed model, C_1, takes the longest time, and the least detailed model, C_8, runs faster than the other models.

The cumulative accuracy loss of each model is also measured in Figure 8. The accuracy loss of the Box-Jenkins model is measured by examining the sum of the squared error through the Box-Jenkins testing process. As simulation proceeds, the cumulative accuracy loss increases for each model. The least detailed model, C_8, has the maximum accuracy loss, while C_2 shows the minimum accuracy loss over time.

Figure 9 shows the AT of FULTON. We applied the Box-Jenkins behavioral abstraction technique to three methods, ($M7$, $M10$, $M12$), and produced ($A7$, $A10$, $A12$), respectively. I_i node is positioned where the two associated different resolution methods reside. Intermediate nodes I_i are connected to the children by the *OR* relationship.

The execution time of leaf methods can be measured by extensive experimentation [Lark et al. 1990], or assessed by available worst-case timing analysis techniques [Arnold et al. 1994; Lim et al. 1994; Marin et al. 1994; Wedde et al. 1994]. To simplify the illustration, we assume that the execution time of each leaf method ($M6$, $M13$, $M14$, $M15$, $M16$, $M8$, $M9$, $M17$, $M18$, $M11$, $M19$, $M20$, $M5$) in

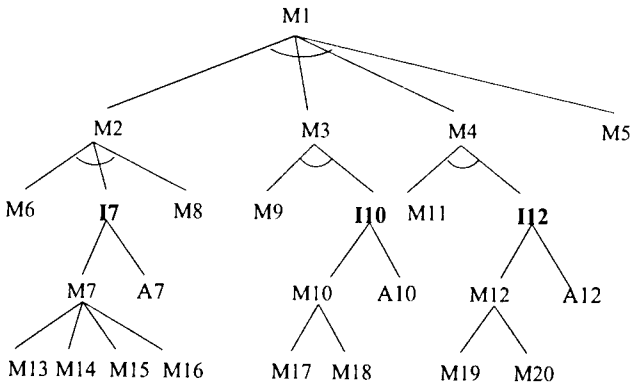


Fig. 9. Abstraction Tree of the FULTON example: Intermediate node I_i is introduced to connect the high resolution method M_i and low resolution method (behavioral abstraction method), A_i

the AT is properly assessed to (2,2,8,4,6,2,2,4,3,2,2,4,2), respectively by using the available analytic tools. Also, we assume that (A_7, A_{10}, A_{12}) takes (4,2,2), respectively. Non-leaf nodes repeatedly look for child's execution time and calculate its own execution time by applying the execution time assessment equations discussed in Section 6. System factor, θ, is assumed to be 0; therefore, the target platform in which the model will be executed is the same one in which the model execution time is measured. We assume that there is no special interesting class for the simulation. The precision loss of A_7 is assumed to be 0.4, while (A_{10}, A_{12}) is assumed to be (0.1, 0.2), respectively. Then, the quality loss of each method, a_7, a_{10}, and a_{12} is defined by 0.325, 0.125, and 0.175, respectively.

8.3 Selection of the Optimal Abstraction Model

The base model of FULTON takes 26 units to complete the simulation. Suppose we have 20 units for a deadline. Upon receiving the time constraint, we immediately know that the behavioral abstraction is needed to make the simulation faster. The optimal abstraction level is determined by *IP1*, *IP2* and the search-based algorithm discussed in Section 7.2.

For a given AT in Figure 9, the objective function of the *IP1* is defined as:

$$Minimize\ (I_7 + I_{10} + I_{12}) \tag{12}$$

subject to

$$\begin{aligned} & a_7 I_7 + a_{10} I_{10} + a_{12} I_{12} \leq a_c \\ & t_7 I_7 + t_{10} I_{10} + t_{12} I_{12} \geq t_c \\ & a_7 = 0.375,\ a_{10} = 0.125,\ a_{12} = 0.175 \\ & t_7 = 4,\ t_{10} = 2,\ t_{12} = 2 \end{aligned} \tag{13}$$

Then, the *IP1* selection of the optimal abstraction level is to solve the objective function defined in equation 12 with the constraints defined in equation 13. Since the desired speedup to be achieved for a given deadline is $26 - 20 = 6$, we assign 6 to t_c. To find out the most accurate combination, we assign 1.0 to a_c. Therefore, the accuracy is not constrained to a certain bound.

The objective function of the *IP2* approach is defined as :

$$Minimize\ (I_7 * a_7 + I_{10} * a_{10} + I_{12} * a_{12}) \tag{14}$$

subject to

$$\begin{aligned} & t_7 I_7 + t_{10} I_{10} + t_{12} I_{12} \geq t_c \\ & a_7 = 0.375, a_{10} = 0.125, a_{12} = 0.175 \\ & t_7 = 4, t_{10} = 2, t_{12} = 2 \end{aligned} \tag{15}$$

Then, the *IP2* selection of the optimal abstraction level is to solve the objective function defined in Equation 14 with the constraints defined in Equation 15.

The search-based algorithm increases the number of behavioral abstraction methods to be used for the deadline. The algorithm examines whether one behavioral abstraction method will resolve the time constraint. Neither of the candidates meets the deadline. Therefore, the algorithm increases the number of behavioral abstraction methods to use for the simulation. The fastest behavioral abstraction A_7 achieves the deadline if either of A_{10} or A_{12} is combined with A_7. Therefore, the algorithm concludes that using 2 behavioral abstraction methods will resolve the timeliness requirement. At this point, the algorithm starts to find the most accurate combination. (A_7, A_{10}) meets the deadline with the maximum accuracy. Therefore, the algorithm declares (A_7, A_{10}) as the optimal abstraction degree for a given AT and a deadline of 20. Figure 10 shows the optimal abstraction level of the given AT. The execution of (I_7, I_{10}, I_{12}) is made by (A_7, A_{10}, M_{12}), respectively. Then, the optimal abstraction model is composed of the sequence $(M_6, A_7, M_8, M_9, A_{10}, M_{11}, M_{12}, M_5)$. Note that the FSM models of *Boiler* and *Turbine* are cut off to save simulation time. The corresponding scheduling diagram is shown in Figure 11.

Table 3 shows other selection examples. *IP2* produces a different answer for a deadline of 22. *IP1* and the search-based methods minimize the number of behavioral abstraction methods in order to minimize the loss of structural information. When modelers want to minimize the loss of structural information (to preserve the base model structure as much as possible), behavioral abstraction occurs at I_7. However, if the simulation objective is to minimize the expected quality loss, we apply behavioral abstraction at I_{10} and I_{12}, as suggested from *IP2*.

9. CONCLUSIONS

We demonstrated a semi-automated methodology to build a model that is right for the simulation objective and real-time constraints. The key to our method is to use the model abstraction technique to generate multiple methods of the system

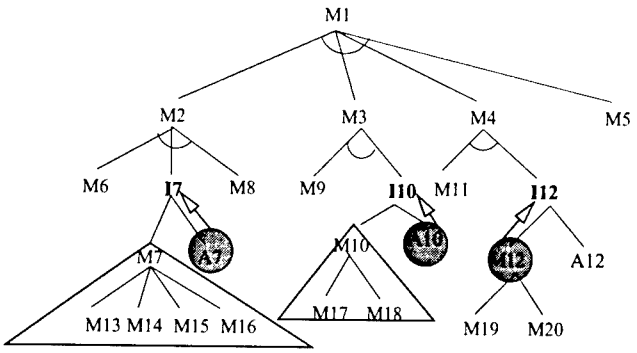


Fig. 10. Optimal abstraction level for a deadline of 20

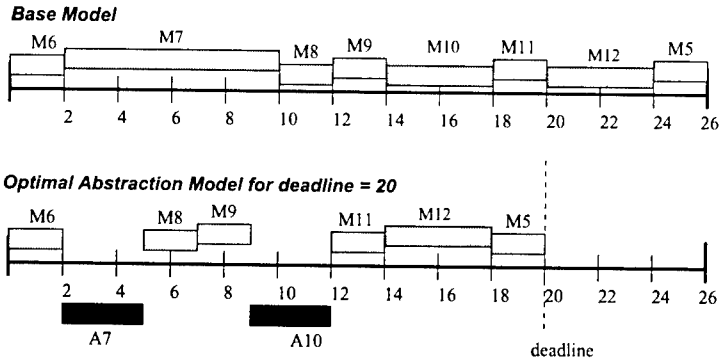


Fig. 11. Scheduling diagram for a deadline of 20

Table 3. Selection examples of three algorithms for FULTON

deadline	20	22	24
IP1	I_7, I_{10}	I_7	I_{10}
	$a_c = 0.5$	$a_c = 0.375$	$a_c = 0.175$
IP2	I_7, I_{10}	I_{10}, I_{12}	I_{10}
	$a_c = 0.5$	$a_c = 0.3$	$a_c = 0.175$
Search	I_7, I_{10}	I_7	I_{10}
	$a_c = 0.5$	$a_c = 0.375$	$a_c = 0.175$

which involve tradeoffs in runtime versus accuracy. Modelers construct the abstraction hierarchy through a structural abstraction phase, and we use the abstraction hierarchy for the source of information where the optimal abstraction degree is determined. By applying the proposed algorithms that determine the optimal abstraction level to simulate the system for a given deadline, we find position(s) where the behavioral abstraction technique is applied. Behavioral abstraction yields time savings of the simulation by discarding detailed structural information, though accuracy is sacrificed. The resulting model simulates the system at an optimal level of abstraction to satisfy the simulation objective for a given deadline so as to maximize the tradeoff of model execution time for accuracy. One of our assumptions was that quality and execution time of the abstraction methods are fairly predictable. Predicting the execution time is possible, in general, by using the available research on runtime estimation techniques; however, assessing the method's quality is difficult. Especially, when the result from a method decreases the quality of other methods, the estimation becomes more complicated. One of the possible solutions is to monitor the selected model's execution under the real-time simulation [Garvey and Lesser 1993b]. When the selected model takes longer time than expected, or the solution quality is lower than expected during the real-time simulation, it is reported to the monitor to take an action. Then, the appropriate actions can be taken to adjust the problems that have been caused by under-estimated/over-estimated duration or quality.

ACKNOWLEDGMENTS

We would like to thank the following funding sources that have contributed towards our study of modeling and implementation of the OOPM multimodeling simulation environment: (1) GRCI Incorporated 1812-96-20 (Gregg Liming) and Rome Laboratory (Steve Farr, Al Sisti) under contract F30602-98-C-0269 a web-based model repository for reusing and sharing physical object components; (2) NASA/Jet Propulsion Laboratory 961427 (John Peterson and Bill McLaughlin) for web-based modeling of spacecraft and mission design, and (3) Department of the Interior under ATLSS Project contract 14-45-0009-1544-154 (Don DeAngelis, University of Miami) for techniques for both code and model integration for the across-tropic-level Everglades ecosystem. Without their help and encouragement, our research would not be possible.

REFERENCES

ARNOLD, R., MUELLER, F., AND WHALLEY, D. 1994. Bounding worst-case instruction cache performance. In *The IEEE Real-Time Systems Symposium* (1994), pp. 172–181.

BARR, A. AND FEIGENBAUM, E. A. 1981. *The Handbook of Artificial Intelligence.* William Kaufmann.

BURNS, A. AND WELLINGS, A. 1994. Hrt-hood: A structured design method for hard real-time systems. *Real-Time Systems 6*, 73–114.

CAUGHLIN, D. AND SISTI, A. 1997. A summary of model abstraction techniques. In *Proceedings of SPIE97* (1997), pp. 2–13.

CPLEX. 1995. *Using the CPLEX Callable Library.* CPLEX Optimization, Inc.

CUBERT, R. M. AND FISHWICK, P. A. 1998. Oopm: An object-oriented multimodeling and simulation application framework. *Simulation 70*, 6, 379–395.

D'AMBROSIO, B. 1989. Resource bounded-agents in an uncertain world. In *International Joint Conference on Artificial Intelligence* (1989).

DAVIS, P. K. AND BIGELOW, J. 1998. Introduction to multiresolution modeling (mrm) with an example involving precision fires. In *SPIE: Enabling Technology for Simulation Science II* (1998), pp. 14–27.

DAVIS, P. K. AND HILLESTAD, R. 1993. Aggregation, disaggregation, and the challenge of crossing levels of resolution when designing and connecting models. In *Proceedings of AI, Simulation and Planning in High Autonomous Systems* (1993), pp. 180 188.

DIGITAL. 1989. *CASE for Real-Time Systems Symposium.* Digital Consulting, Andover, MA.

FISHWICK, P. A. 1995. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall.

FISHWICK, P. A. 1997. A visual object-oriented multimodeling design approach for physical modeling. *University of Florida Technical Report 9.*

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and intractability.* W.H. Freeman and company.

GARVEY, A. J. AND LESSER, V. R. 1993a. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics 23*, 6, 1491 1502.

GARVEY, A. J. AND LESSER, V. R. 1993b. A survey of research in deliberative real-time artificial intelligence. *UMass Computer Science Technical Report 93-84.*

GARVEY, A. J. AND LESSER, V. R. 1995. Design-to-time scheduling with uncertainty. *UMass Computer Science Technical Report 95-03.*

GETTYS, E. AND KELLER, F. 1989. *Physics.* McGraw-Hill.

KIM, G. 1998. A model validation methodology for isolating inconsistent knowledge between fuzzy rule-based and quantitative models using fuzzy simulation. *Ph.D Dissertation, Department of Computer Information Science and Engineering.*

KOPETZ, H., ZAINLINGER, R., FOHLER, G., KANTZ, H., PUSCHNER, P., AND SCHUTZ, W. 1991. The design of real-time systems: From specification to implementation and verification. *Software Engineering 6*, 72–82.

KORF, R. E. 1990. Depth-limited search for real-time problem solving. *Real-Time Systems 2*, 7-24.

LARK, J. S., ERMAN, L. D., FORREST, S., AND GOSTELOW, K. P. 1990. Concepts, methods, and languages for building timely intelligent systems. *Real-Time Systems 2*, 127–148.

LEE, K. AND FISHWICK, P. A. 1996. Dynamic model abstraction. In *Proceedings of Winter Simulation Conference* (1996), pp. 764 771.

LEE, K. AND FISHWICK, P. A. 1997a. A methodology for dynamic model abstraction. *Transactions of the society for computer simulation international 13*, 4, 217–229.

LEE, K. AND FISHWICK, P. A. 1997b. A semi-automated method for dynamic model abstraction. In *Proceedings of AeroSense 97* (1997), pp. 31 41.

LEE, K. AND FISHWICK, P. A. 1998. Generation of multimodels and selection of the optimal abstraction level for real-time simulation. In *AeroSense 1998* (1998), pp. 164–175.

LIM, S. S., BAE, Y. H., JANG, G. T., AND AL., E. 1994. An accurate worst case timing analysis for risc processors. In *the IEEE Real-Time Systems Symposium* (1994), pp. 97-108.

LIU, S. AND CHINGAND, Z. 1991. Algorithms for scheduling imprecise computations. *IEEE Computer 24*, 5, 58–68.

MARIN, G., BAKER, H. T., AND WALLEY, D. B. 1994. A retargetable technique for predicting execution time of code segments. *Real-Time Systems 7*, 2, 130–159.

MASTERS, T. 1995. *Neural, Novel and Hybrid Algorithms for Time Series Prediction.* John Wiley and Sons, Inc.

R FOURER, D. G. AND KERNIGHAN, B. 1993. *AMPL: A Modeling Language for Mathematical Programming.* The Scientific Press.

RAGSDALE, C. T. 1998. *Spreadsheet Modeling and Decision Analysis: A Practical Introduction to Management Science.* South-Western College Publishing.

RAMAMRITHAM, K. AND STANKOVIC, J. A. 1984. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software 1*, 3, 65–75.

RAVINDRAN, PHILLIPS, D., AND SOLBERG, J. J. 1987. *Operations Research.* John Wiley and Sons.

RUTLEDGE, G. W. 1995. Dynamic selection of models. *Ph.D Dissertation, Department of Medical Information Sciences, Stanford University.*

STANKOVIC, J. A., RAMAMRITHAM, K., AND CHENG, S. 1985. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers 34*, 12, 1130–1143.

WEDDE, H. F., KOREL, B., AND HUIZINGA, D. M. 1994. Formal timing analysis for distributed real-time programs. *Real-Time Systems 7*, 1, 58–90.

WEISS, S. M. AND KAPOULEAS, I. 1989. An experimental comparison of pattern recognition, neural nets, and machine learning classification methods. In *Proceedings of IJCAI-89* (1989), pp. 781–787.

ZEIGLER, B. P. 1976. *Theory of Modelling and Simulation.* John Wiley and Sons.

ZEIGLER, B. P. 1990. *Object Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* Academic Press.

*U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10047

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.