Saeid Nourian
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Ph.D. (Computer Science)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

AVRA – An Architecture for VR-Based Applications
TITRE DE LA THÈSE / TITLE OF THESIS

Nicolas D. Georganas
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Ketan Mayer-Patel	Dorina Petriu
Abed El Saddik	Emil Petriu

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

AVRA
An Architecture for VR-Based Applications

By
Saeid Nourian

Submitted to the Faculty of Graduate and Postdoctoral Studies, University of Ottawa, in partial fulfillment of the requirements for the PhD degree in Computer Science

Ottawa-Carleton Institute for Computer Science

School of Information Technology and Engineering

University of Ottawa

Your file Votre référence
ISBN: 978-0-494-48410-4
Our file Notre référence
ISBN: 978-0-494-48410-4

ABSTRACT

Despite recent advances in the software and hardware aspects of Virtual Reality (VR), from a software *design* point of view VR technology remains primitive. In particular, most existing VR applications suffer from lack of extensibility, maintainability, reusability and interoperability. This thesis proposes a flexible and practical architecture for defining and constructing VR simulations that addresses the above issues. The proposed architecture employs lessons learned from the more architecturally advanced software fields such as those of web-based applications and database banks. The advancement of these fields is driven by the fact that the contents and views change frequently, hence the architecture must be (1) flexible with changes in content, and (2) decouple the content and the view (i.e. using MVC pattern [52]). These essential requirements gave birth to such technologies as the extensible markup language (XML), and the extensible style sheets (XSL).

Most VR applications are similar to the web-based applications in that they also deal with contents and views. The *content* (or *model*) describes the conceptual and the mathematical aspects of the elements that exist in the virtual environment. The *view* is the visual representation of the content often rendered in a 3D platform. As an application matures its content and view often change. The contents are the more reusable components whereas the views are more application specific. Take a heart surgery simulation for example. The content consists of some mathematical and conceptual models of the heart and the human body as a whole; and the view is the visualization of a human body (including the heart) and the operating room and surgical tools.

Although there is a clear separation between the content and the view models, the two are often tightly interconnected at run-time. For example, a change in the content should reflect the corresponding changes in the view, often through means of dynamic scene creations and animations. Changes can also be initiated at the view side through means of user interactions and specifications. These changes should be reflected back at the contents accordingly.

The above issue is addressed in the web-based applications through use of XML for representing the content (data) and XSL for specifying *how* the content should be viewed. Most VR applications are interactive simulations of the real world and these simulations are either mathematical in nature or based on a rich set of data models both of which can be easily represented by XML models. The view component, however, is much more complex in VR than it is in web-based applications. In such applications, the view target is simply a static hyper-text representation of the data whereas in VR applications the view consists of dynamic 3D geometries that interact with each other and whose shape and appearance may change on the fly.

This thesis presents AVRA, a novel architecture that dynamically generates VR simulations based on XML descriptions that are received as inputs. These XML descriptions are used to define and configure the various elements of an interactive VR application such as simulation models, 3D graphics, visualization behavior and the nature of user interactions. The proposed architecture uses two categories of XML models: those that describe the numerical model of a simulation and those that describe how the numerical output is to be visualized in a virtual environment.

Upon loading the XML descriptions, AVRA dynamically generates a VR application that corresponds to those specifications. The result consists of a 3D scene with configurable graphical elements that are animated based on the numerical outputs of the simulation models. The task of managing the communication between the model and view components as well as their construction and destruction is automatically handled by AVRA. In essence, this framework allows developers to quickly construct the simulation components of a VR application through XML descriptions and view plugins thereby allowing developers to focus their efforts on implementing the higher level functionalities of the application.

ACKNOWLEDGEMENT

This thesis is the result of several years of hard work whereby I have been accompanied and supported by family, friends and colleagues. It is a pleasure to thank the many people who made this thesis possible.

I wish to first express my gratitude to my Ph.D. supervisor, Dr. Nicolas D. Georganas for his continual invaluable guidance. His enthusiasm, inspiration, and encouragement made this thesis possible. I also like to thank Dr. Emil Petriu, Dr. Dorina Petriu and Dr. El-Saddik for their support during the development of this thesis.

Special thanks to my colleagues at the DISCOVER lab for providing an excellent and inspiring working atmosphere. Especially I like to thank Abdelwahab Hamam and Houman Rastgar for our insightful lunch breaks and Francois Malric for his help every time I busted a network card.

Finally, I would like to thank my family. A special thought is devoted to my sisters, Somayeh and Sara, for believing in me and for their never ending support.

I am forever indebted to my parents. Their encouragement and support is a powerful source of inspiration and energy. To them I dedicate this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF Acronyms

ASVC	Acquisition, Simulation, Visualization and Controller
AVRA	An Architecture for VR-Based Applications
CAVE	Cave Automatic Virtual Environment
CellML	Cellular Markup Language
DIVERSE	Device Independent Virtual Env. Reconfigurable, Scalable, Extensible
HAML	Haptic Application Meta Language
HMD	Head Mounted Display
IT	Information Technology
KRL	Knowledge Representation Layer
MathML	Mathematical Markup Language
MVC	Model-View-Controller
MVML	Model-View Markup Language
SCIVE	Simulation Core for Intelligent Virtual Environments
VR	Virtual Reality
VRML	Virtual Reality Modeling Language
XML	Extensible Markup Language
xPheve	Extensible Physics Engine For Virtual Environments
XSLT	Extensible Stylesheet Language Transformation

CHAPTER 1 - INTRODUCTION

The field of Virtual Reality (VR) has a promising future. VR is a technology that allows user interaction with computer generated environments that look and feel realistic. In this age of Information Technology, we are faced with seemingly never ending data streams in various forms and formats, often unfriendly to the human mind. VR provides a human-friendly way of visualizing and effectively understanding these otherwise meaningless data streams.

In the past decade, we witnessed the rebirth of VR as it finally overcame the hardware limitations. The VR technology is now widely used for medical, military, aero-space training and many other simulations and its popularity is steadily growing.

As more and more VR applications are developed, the software limitations are becoming more apparent. Since VR was facing hardware limitations for a long time, little effort was made toward addressing the software issues. For that reason, the basic software engineering principles such as extensibility and reusability are not addressed well in existing VR architectures. There are however valuable lessons learned in the web and IT sectors that can be applied to VR as well.

This thesis proposes an architecture that addresses the various software engineering issues that most VR applications are facing today by utilizing some of the lessons and technologies that proved effective in web-based applications.

This chapter will review the model-view architecture and then present the motivation for using it in VR by discussing some of the most significant shortcomings that exiting VR applications are facing today. Subsequently the objectives and contributions of this thesis will be outlined.

1.1 Model-View Architecture for Virtual Environments

The concept of model-view or data-view architecture has existed since the early database software applications [52]. It was not emphasized, however, since at that early age the capabilities of the computers were limited. Hence, the data structure and the view component often remained unchanged. The concept gained popularity and was extensively studied after the Internet revolution. Its necessity was felt due to the ever growing web-sites that relied heavily on dynamic databases and web-pages. XML technology which was introduced in 1998 provided a solution for this issue (see section 2.2.1 for more information about XML). XML and its suite of *style sheet markup languages* successfully addressed two major requirements of the modern web-based applications:

1) XML decoupled the data part of a web-page from its view elements. The data part is encoded in its own independent XML file and the view part is encoded in a style sheet file based on the knowledge of the general data structure.

2) XML offered much more flexible data structure capabilities than the traditional methods. Under XML, data structures are typically in the form of trees with nested tags, a form that is generally considered more flexible for representing databases than simple tables. In XML you can also have tags that reference other elements in a same or external XML file hence representing more complex graph-based data structures. In fact, XML is so flexible that its capabilities exceeds far beyond describing databases; it can be used to model any mathematical or graphical model.

Due to the above, XML technology was quickly adopted by not only web-based applications, but any application that requires storage and/or transmission of structured data and models in a standard and reliable manner. In addition to data, XML is used by many applications for the purpose of encoding configurations and parameter settings [16]. In practice, the extend of which the functionality of a software application can be defined and configured through use of XML files has often direct correlation with the extensibility, reusability and scalability of that software.

Virtual Reality (VR) applications can also benefit from the model-view architectures in a similar way as web applications do (see section 2.1 for information about VR). Just like their web-based counterparts, the primary task of most VR applications is to present a set of dynamic input data or models through human-friendly output channels. The difference is that the data models are often simulation based and the output is presented through 3D graphics, sounds, haptic data and other sensory channels.

A model-view architecture for virtual environments can be defined as an architecture which receives a set of [XML-based] models as its input from which it dynamically generates an interactive virtual environment. The input models can be simulation data, mathematical formulas, geometrical models, device configurations and more.

1.2 Motivation

For a long time, the VR technology was held back due to the limitations in hardware technology. VR applications are often heavily involved computationally and the fact that they must employ multiple output channels and stimulate the various senses of their human users adds considerably to their complexity. All these have resulted in more attention to the efficiency of the computations as opposed to effectiveness of the system design.

As hardware technology has finally caught up to a level suitable for the growth of VR technology, we are now faced with software limitations. Most VR applications developed to date were designed with specific functional requirements; requirements that were tightly hard coded within an architecture that does not welcome changes. Take a heart surgery simulation as an example. It involved a particular surgery procedure, some visual models of the heart and the body and a set of surgical tools. Consider a simple change to enhance the graphics by adding or upgrading the 3D models in the scene. It may seem feasible; after all it is simply the matter of loading different model files. Unfortunately, in most applications, this task involves more work that just loading the new models in the environment. The complexity arises when deciding how the new models are to interact with the rest of the application. For example, will the new 3D model of the heart still animate the heart-beat at the precise rate as calculated by the simulation module? Or is the developer forced to manually re-establish the correlation between the heart beat rate and the corresponding animation in the new 3D model? If the simple task of adding new

3D models that represent biological organs or surgical tools requires changes at the application level then the architecture of the application is not well designed.

Now consider changes at a wider scope such as changing the nature of the heart disease for which the simulation is designed. The extent of this change goes beyond the superficial changes in the appearance; it involves changes at the mathematical and physiological level. Consider a surgery simulation for *cardiovascular disease* [17]. It involves special procedures for reopening, repairing or replacing the damaged blood vessels. Now consider a surgery simulation for the *heart valve disease* [18] which involves removal of all infected tissue and repair or replacement of the affected valve. Although the surgical procedures involved in the treatment of the above diseases are very different in terms of tasks and objectives, they both involve interaction with the same organ (heart) and a same physiological system. A flexible architecture that allows reuse of the core physiology and the high-level simulation would be of significant value in such applications.

The extent of variation in the above simulation may further expand from the original heart surgery to that of kidneys, eyes, lungs and the brain. Naturally the physiology is different depending on the particular organ of interest. The basic nature of the simulation is nevertheless the same as before: performing surgical procedures. Hence, we have a potential source of reusability.

Within each of the above levels of changes, there is the issue of maintainability. Science changes on a daily basis. In the heart surgery example one must accommodate for the discoveries of the new physiological models that replace the older models to achieve a more precise simulation of the heart. Such models can change as frequently as the database model would in a typical web-based application for example. Unfortunately, most existing VR applications are not designed to accommodate for such frequent changes!

A flexible and extensible architecture is needed in order to overcome the same problems that web-based applications were facing not too long ago. This is an essential requirement that must be met before VR technology can enjoy the same rapid advancement that web technology has enjoyed a decade ago.

1.3 Objectives

The goal of this thesis is to introduce a new architecture that is more effective than existing architectures for constructing VR applications that are *extensible*, *maintainable* and *reusable*. The aforementioned factors are three basic software engineering principles that are currently absent in most VR applications due to the mixture of the underlying components.

An objective of this thesis is to adopt the model-view principles in order to decouple the mathematical and data models from their graphical representations. Such a design would allow users to change the mathematical models independently of the graphical models and vise versa. It would also give developers the ability to control the nature of the simulation being displayed in a VR application without changing the application itself.

The second objective of this thesis is to design and implement an architecture that automatically generates VR-based simulations that visualize target simulation models based on a reconfigurable XML file that describes the view. This architecture would enable users to specify exactly how the numerical results of a simulation model are to be visualized within an interactive and animated VR scene.

In addition to the above, this thesis involves researching systematic approaches for specifying the behavior of the VR application that built on top of the above architecture with parameterized configurations for input/output devices, user interactions, and physical attributes. Although these components are not directly relevant to the simulation being performed by the VR application, they do constitute a major portion of the desired functionalities of the application.

1.4 Contributions

In addition to achieving the aforementioned objective this thesis has resulted in several major contributions. The following constitute the novel contributions of this thesis:

- Design of an architecture that receives the core simulation models as XML files at run-time and dynamically generates an application whose behavior is dictated by the contents of the model description files.

- Design of a novel VR architecture that receives the view description of a simulation through XML files at run-time and dynamically generates a virtual environment with graphical models that are connected to corresponding simulation models in such a way that an update in the simulation time-step not only recalculates the mathematical models but also animates the graphical models.

- Design of a novel plugin architecture along with a unified interface scheme for defining and interfacing with compiled entities that are used for specific visualization tasks which are selectively added to simulation applications upon request. These view plugins are built on top a physics engine [2] hence include capabilities for physically realistic visualizations.

- Design of a dynamic system of model-view connections through entities called *connectors* in order to allow automated channeling of data from specified outputs of models to specified inputs of view plugins.

- Design of MVML, a novel XML-based markup language for defining the simulation view which consists of the visualization of numerical simulations in VR. Through MVML application developers can specify which view plugins to use, how to connect them to simulation models and which parameters to pass to the simulation components.

- Implementation of AVRA which consists of components and sub-components that address the various architectural elements as discussed above. The resulting implementation presents a fully functional framework that provides interfaces for receiving simulation models, view description and view plugins.

- Implementation of a Hodgkin-Huxley model of neuron action potential [32] as defined within a CellML model. The view is customized to use particle systems and various interpolators to present a realistic and numerically-accurate visualization of the axon of a neuron cell.

Although most examples in this thesis are based on CellML models and medical surgery simulations, the proposed architecture is not limited to medical simulations. The architecture is useful for most simulations in a variety of fields and disciplines.

1.5 Overview of Chapters

This thesis is organized as follows. Chapter 2 presents a brief summary of the related technology and a discussion of the existing VR frameworks along with some of their shortcomings. Chapter 3 presents the requirements that were considered in the design of AVRA along with an overview of its architecture. Chapter 4 presents the simulation models which constitute the basis of AVRA structure. The details of the view components span over chapter 5 through 7. The main components of view are discussed in chapter 5, followed by specification of view plugins in chapter 6 and finally model-view connectors in chapter 7. Chapter 8 uncovers the Model-View Markup Language along with simple examples that demonstrate how it can be generated. Chapter 9 presents the implementation of AVRA and a sample application along with a discussion on the results. Finally, a conclusion is presented in chapter 10.

1.6 Publications resulting from this thesis

- X. Shen, J. Zhou, A. Hamam, , S. Nourian, N. R. El-Far, F. Malric and N. D. Georganas, 'Haptic-enabled, Tele-mentoring Surgery Simulation: design, implementation and evaluation", IEEE Multimedia , Vol.15., No.1, Jan.-Mar. 2008, pp.64-76

- S. Nourian and N.D. Georganas, "AVRA: an Architecture for VR-based Applications" subm. to IEEE VECIMS2008, Istanbul, July 2008

- S. Nourian, X. Shen, N. D. Georganas, "A Model-View Driven Architecture for VR-Based Simulations" Proc.. HAVE 2007 - IEEE International Workshop on Haptic Audio Visual Environments and their Applications, Ottawa, Ontario, Canada, Oct.2007

- X. Shen, A. Hamam, F. Malric , S. Nourian, N. R. El-Far and N. D. Georganas, "Immersive Haptic Eye Tele-surgery Training Simulation", Proc.3DTV

Conference 2007 "The True Vision: Capture, Transmission And Display of 3D Video", Kos, Greece, May 2007

- Hamam, S. Nourian, N. El-Far, F. Malric, X. Shen and N.D. Georganas, "A Distributed, Collaborative, and Haptic-Enabled Eye Cataract Surgery Application with a User Interface on Desktop, Stereo Desktop, and Spatially Immersive Displays", Proc. IEEE HAVE 2006, Ottawa, Nov. 2006

- S. Nourian and N.D. Georganas, "Role of Extensible Physics Engine in Surgery Simulations", Proc. IEEE HAVE' 2005, Ottawa, Oct.2005

- N. R. El-Far, S. Nourian, J. Zhou, A. Hamam, X. Shen, and N. D. Georganas, "A Cataract Tele-Surgery Training Application in a Hapto-Visual Collaborative Environment Running over the CANARIE Photonic Network", Proc. IEEE HAVE' 2005, Ottawa, Oct.2005

CHAPTER 2 - BACKGROUND

2.1 Virtual Reality

The concept of Virtual Reality (VR) has been around since the 1950s, when Morton Heiling proposed and later built a film-viewer machine capable of employing multi-sensory (sight, sound, smell, and touch) technology in order to immerse the viewers in the films [1]. More interactive VR applications emerged in the 60s and 70s and the interactivity feature eventually became a necessary part of all modern VR application. In their book, G.C. Burdea and P. Coiffet define virtual reality as follows:

> *Virtual reality is a high-end user-computer interface that involves real-time simulation and interaction through multiple sensorial channels. Theses sensorial modalities are visual, auditory, tactile, smell and taste. [1]*

For each sensorial channel, there is a variety of hardware and software technologies available. Some of them, such as the auditory and visual hardware are in a matured state with widely accepted standards. The haptics technology (for tactile sensing) is a state-of-the-art technology but it is maturing quickly. Others such as smell and taste are still primitive.

2.1.1 Visual Technology

The head-mounted-display or HMD, first introduced in 1968 by Ivan Sutherland, is one of the most effective ways to feel immersive. HMDs usually cover the entire vision scope of human eyes and their visual contents can be updated as the head rotates about the neck or the user wonders around the room. Stereoscopic displays go one step further and generate an optical illusion that enables the viewers to perceive the depth of the 3 dimensional objects without the discomfort of wearing the head mounted display.

In HMDs, stereo output is achieved by simply displaying slightly different images on the LCD screens to correspond to the left-eye and right-eye perspectives. Especial stereo glasses need to be used in order to perceive stereo effects when looking at non-HMD displays such as a monitor or projector. These stereo glasses are synchronized with the display system in such a way that they block the left eye when the display corresponds to the right eye and vise-versa.

Recently, special monitors are made available that contain one set of pixels visible to left eye and another set visible to right eye. At each frame, the images for both left and right eye perspectives are rendered and displayed on their corresponding pixels. As a result, stereo glasses are not required to perceive stereo images on these monitors.

The projector-based displays are used with the wide screens and are useful for those scenarios in which many participants and observers are present. The *CAVE* is an example of a projector-based display consisting of several projection screens that surround the observers from the sides and in some cases from above and below [2]. Stereo images are projected on these screens and the participants are typically required to wear stereo glasses.

2.1.2 Haptic Technology

In the early 70s, haptic technology emerged and refined the way we interact with 3D worlds. The first haptic device was introduced by Frederick Brooks and his colleagues at the University of North Carolina in 1971 [1]. Although still experimental, haptic technology has proved extremely valuable for boosting immersive-ness, as it not only allows 3D interaction but also generates force-feedback. The introduction of the force-feedback including the sense of touch, pulse, vibration, friction and viscosity opens a whole door to new possibilities for VR.

There are several companies actively involved in design and manufacture of haptic devices. The *PHANToM Arm* (see Figure 1*)* is one of the most popular devices available in the market [15]. It looks like a robotic arm and it can sense the movement and force exerted by the user [2]. It has 6 degrees of freedom for movement and 3 degrees of freedom for force-feedback. The *Haptic Master Arm* is a heavier haptic device with the capability of exerting more feedback force. It is therefore more adequate for applications that involve heavy machinery training but less attractive for sensitive applications such as medical surgery [1]. For those applications that require force feedback for the each and every figures of the human hand, CyberGrasp (see Figure 2) is more useful. It is a haptic device that senses the movements of each joint of each finger and exerts force feedback as required.

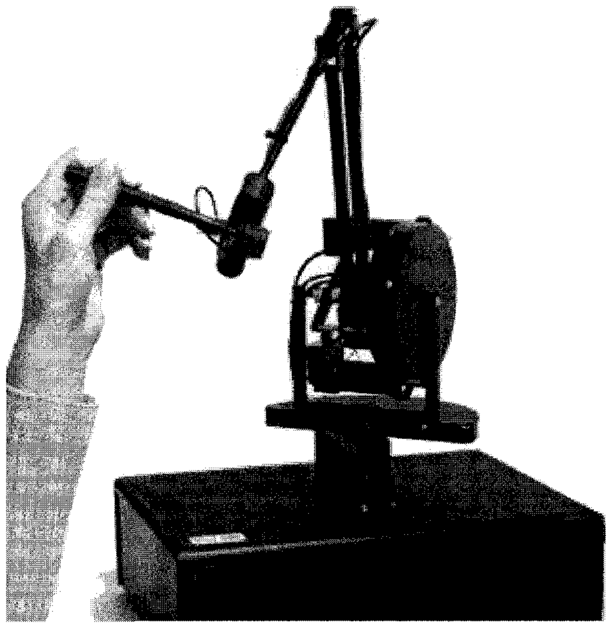


Figure 1 – PHANToM Arm

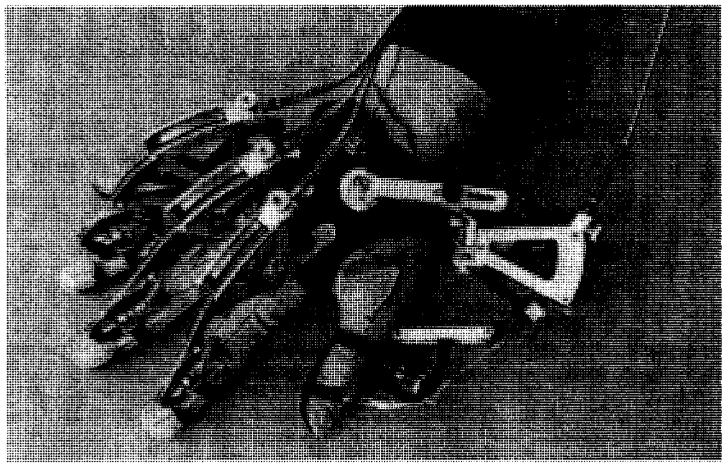


Figure 2 – Cyber Grasp

2.1.3 Physics Engine

Most VR applications require a realistic simulation of the basic laws of physics that govern our universe. For instance, all rigid objects in a VR environment must be affected by the pull of gravity. Also, unrealistic events such as a rigid object passing through a solid wall must be prevented. Since these requirements are common in most VR applications, it makes sense to implement them within reusable libraries in order to avoid *reinventing the wheel* every time a VR project is in need of such features. These reusable libraries of physics laws are called *physics engines*. Figure 3 demonstrates an example of how the objects of a VR application may be categorized according to their physical attributes.

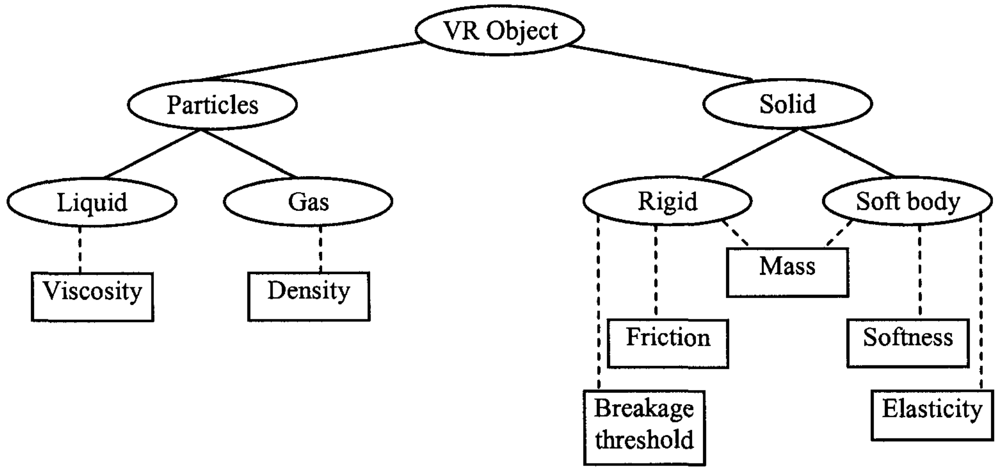


Figure 3 – Categories of physical objects and their attributes

There are generally two types of physics engines: *real-time* and *high-precision*. Most interactive VR applications require that the physics computations execute as efficiently as possible while the application is running; hence they use the real-time physics engines. High-precision physics engines are usually used in non-interactive application such as animations in which case the physics calculations are done offline during the rendering phase.

Real-time physics engines are widely used for game development where run-time efficiency takes precedence over precision. The most popular commercial physics engine is HAVOK [41]; it is used in over 150 game titles. Other physics engines include the Open Dynamics Engine (ODE), Newton Game Dynamics, Tokamak Physics, Bullet, OPAL and more. Most of these physics engines are optimized for games and as a result inadequate for precision applications such as surgery simulations or medical training.

Recently we witnessed the birth of Physics Processing Units (PPU) [42] that efficiently process many of the calculations involved in physics engines, hence taking some loads off the CPUs and improving the efficiency. The leading company in this field is Aegia; their PPU is called NovodeX which is used by their Megon physics engine in order to perform physics processing faster.

In a response to the introduction of PPUs, nVidia and ATI have designed and manufactured programmable graphics cards that add more capabilities to the existing parallel pipeline infrastructures in order to make them suitable even for non-graphics calculations (i.e. physics calculations) [43].

2.2 Modeling Languages

One of the motivations for design of AVRA is the vast availability of formal languages are used to represent the data or mathematical models of the simulations. This section introduces XML which is root of most modern modeling languages followed by some of its descendants that are used to represent mathematical formulations, biological phenomena and more.

2.2.1 XML Technology

The Extensible Markup Language (XML) is a general purpose markup language used in a wide variety of applications to represent structured data. XML is a well-formed markup language meaning that the beginning and the end of each clause is specified by corresponding tags. Hence XML data is readable not only for human observers but also for computer applications. There are many reasons that have led to introduction of XML however the objectives of interest when designing XML can be summarized in ten items. Table 1 outlines the designs goals of XML [21].

1	XML shall be straightforwardly usable over the Internet.
2	XML shall support a wide variety of applications.
3	XML shall be compatible with SGML.
4	It shall be easy to write programs which process XML documents.
5	The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6	XML documents should be human-legible and reasonably clear.
7	The XML design should be prepared quickly.
8	The design of XML shall be formal and concise.
9	XML documents shall be easy to create.
10	Terseness in XML markup is of minimal importance.

Table 1 – Ten design goals of XML

Figure 4 shows an example of a simple XML data file. XML data can be saved in disk (usually with an .xml extension) or they can be transferred to other applications through a network through the internet. As it is shown in the figure, the first line of an XML file is the XML *declaration* which denotes the XML version (usually 1.0) and other optional information such as the encoding and external dependencies.

```
<?xml version="1.0" encoding="UTF-8"?>
<recipe name="bread" prep_time="5 mins" cook_time="3 hours">
  <title>Basic bread</title>
  <ingredient amount="3" unit="cups">Flour</ingredient>
  <ingredient amount="0.25" unit="ounce">Yeast</ingredient>
  <ingredient amount="1.5" unit="cups" state="warm">Water</ingredient>
  <ingredient amount="1" unit="teaspoon">Salt</ingredient>
  <instructions>
    <step>Mix all ingredients together, and knead thoroughly.</step>
    <step>Cover with a cloth, and leave for 1 hour in warm room.</step>
    <step>Knead again, place in a tin, then bake in the oven.</step>
  </instructions>
</recipe>
```

Figure 4 – XML data that contains the recipe for making break. [22]

The application specific data start from the second line as shown in the figure. It consists of the recipe of an item (bread in this case). The recipe has 3 *attributes*: a name, a preparation time and a cooking time. It also contains 6 *elements*: a title, 4 ingredients and one set of instructions. The ingredients are Flour, Yeast, Water and Salt. Also each of these ingredients has two attributes: the *amount* and the *unit*. The set of instructions is a nested XML element, as itself consists of 3 inner elements (the steps).

2.2.2 MathML

When it comes to mathematical expressions, they can sever two distinct purposes: (1) to demonstrate [abstract] phenomena, and (2) to represent a calculation precisely. A teacher or project leader may use mathematical notations to visually demonstrate an idea or phenomena. In such situations, the details of the mathematical expression may be dropped in favor of presenting a more abstract human-friendly annotation. For this purpose, the mathematical expression is meant to be a visual representation. An example of this is the Leibniz notation for the chain rule when solving derivatives:

$$\frac{\partial f}{\partial x}\frac{\partial x}{\partial t} = \frac{\partial f}{\partial t}$$

The above serves as an excellent notation for demonstrating the subject mater even though it does not convey anything computationally meaningful. In the world of mathematics, any expression may serve either or both of the aforementioned purposes and MathML is a markup language that is designed to meet both [23].

The expression $(a + b)^2$ can be demonstrated in two different ways using MathML notation: *presentation* or *content*. From a *presentational* point of view (a+b) is a base and '2' is a script. In that case MathML notation for the expression would be as demonstrated in Figure 5.

Like any XML notation, MathML expressions consist of a hierarchy of elements, some of which recursively contain a bunch of inner elements. The leaf elements are those that do not contain any sub elements. Instead they may contain an identifier, a number or an operator represented by <mi>, <mn> and <mo> tags respectively.

```
<msup>
  <mfenced>
    <mrow>
      <mi>a</mi>
      <mo>+</mo>
      <mi>b</mi>
    </mrow>
  </mfenced>
  <mn>2</mn>
</msup>
```

Figure 5 – The MathML presentational notation for $(a + b)^2$

The <msup> and <mrow> tags are used for scripting and general layout respectively. In the above expression, <msup> denotes that its second child is visually superscripted with respect to the first child.

The above markup notation is great for visual rendering, but it does not effectively reflect the mathematically semantic meaning of the expression. In order to precisely encode a mathematical expression in MathML we must use the *content* markup instead of the presentation markup. The MathML code below demonstrates what the content markup for a same example looks like:

```
<apply>
  <power/>
  <apply>
    <plus/>
    <ci>a</ci>
    <ci>b</ci>
  </apply>
  <cn>2</cn>
</apply>
```

Figure 6 – The MathML content notation for $(a + b)^2$

The <apply> tag is perhaps the most important element in MathML content expressions. It denotes that the value of its children can be mathematically computed and returned. In the content markup we use <ci> to represent identified and <cn> to represent numbers. Unlike the presentation markup, the operators are denoted by empty elements in the content markup of MathML. For example <plus/> element denotes that the next two children are to be added together in current context. Similartly <power/> element denotes that the next element should be raised to the power of the one after. Currently MathML contents supports widely range of mathematical expressions in the following subject areas:

- Arithmetic, Algebra, Logic and Relations
- Calculus
- Set Theory
- Sequences and Series
- Trigonometry
- Statistics
- Linear Algebra

For the purpose of this thesis, we are more concerned with the *content* markup as opposed to the *presentation* markup as we would like to load, parse and compute the underlying mathematical expressions that constitute the high-level simulations.

2.2.3 CellML

CellML is an XML-based markup language designed to encode the models of biological phenomena [3][4]. Although it was originally designed to describe biological models at cellular level, its capabilities have grown beyond that. Indeed, CellML models can be used to describe the behavior of a biological organ or even an entire system of organs for that matter. There are also several models built with CellML that describe non-biological phenomena such as those of civil and mechanical engineering.

The idea behind CellML is as follows. When a model is published in a scientific journal, any reader who wishes to use or verify the model must implement his or her own software program to verify the behavior of a system under the new model(s). This is especially difficult for those with little or no programming experience. In addition, when a newer (more accurate) version of the model becomes available, the older simulation becomes worthless as the entire simulation needs to be rewritten. If the models are encoded in CellML however, the older simulation can be reused by simply replacing the older CellML with the new one.

Most models are defined by a set of mathematical equations that govern the interactions between two or more components. From a high-level point of view, CellML encodes a model by defining its components, the output and inputs of those components and finally mapping those outputs to their corresponding inputs (connecting the components together).

Consider a simple Membrane/Electrophysiology model as shown in Figure 7. This model consists of a one-compartment cell with a membrane that separates the intracellular from the intracellular subspace. There are two channels from which sodium and calcium ions flow back and forth between the two subspaces [19].

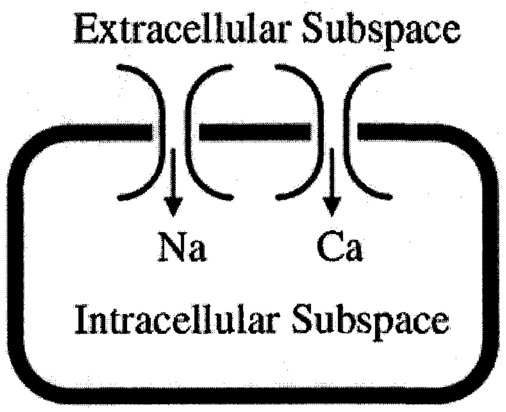


Figure 7 - The demonstration model consists of a one-compartment cell model, where the interior of the cell is separated from the extracellular space by a membrane [19].

Figure 8 shows the CellML structure for the above phenomenon. In CellML, the two subspaces and the membrane are represented by three components. The cell membrane component is connected to both Intra and Extra components. As this model is time dependant, the CellML model will also contain an environment component that passes on the value of the global time to whichever component that requires it for its internal calculations.

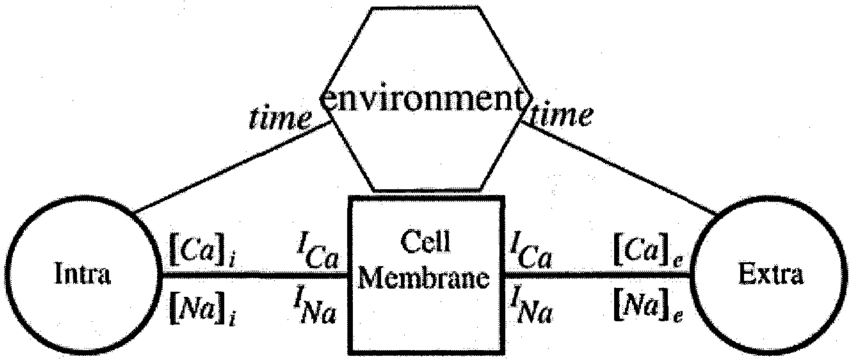


Figure 8 - The network defined in the CellML description of the simple electrophysiological model introduced in Figure 7. The model has four connections and four components, representing the intra- and extra-cellular compartments, the membrane, and an abstract container representing the environment. The variables in the model are shown next to the components in which their value may be modified and alongside the connections along which they are passed to other components (where their value may not be modified) [19].

The partial CellML code in Figure 9 demonstrates the skeleton of the above model. The *model* element is the root element of any CellML data. Every CellML model has a *name* attribute that uniquely identified the model.

Before we can define the components and their various interfaces, we need to define the *units* that are used by the models. The *concentration_units* in the above example is milli-mole per liter for instance. Figure 10 shows how the unit of mole with the prefix of milli is added to the unit of liter with the exponent of -1 in order to represent the desired target unit. The *flux_units* is the concentration per second; therefore it reuses the previously defined *concentration_units*.

Figure 11 shows the implementation of the *intra_cellular_space* component. It consists of some variable definitions and a mathematical formula. The variables can have either "in" or "out" as their public_interface attribute. If a variable receives its value from an external source its public_interface should be set to "in". If on the other hand the value of a variable is to be calculated and fed into other components, it should be flagged as "out". Local variables should use the default value for this attribute ("none"). For the case of intra_cellular_space component, it receives I_Na , I_Ca and time as inputs and calculates Na and Ca as outputs.

```
<model name="basic_ep_model" >

  <units name="concentration_units">
    <!-- unit definition here -->
  </units>

  <component name="intra_cellular_space">
    <!-- component definition here -->
  </component>

  <component name="extra_cellular_space">
    <!-- component definition here -->
  </component>

  <component name="cell_membrane">
    <!-- component definition here -->
  </component>

  <connection>
    <!-- membrane-intra connection definition here -->
  </connection>

  <connection>
    <!-- membrane-extra connection definition here -->
  </connection>

</model>
```

Figure 9 – The skeleton of a CellML file that contains a model named basic_ep_model which is consisted of four components and a number of connections and units.

```
  <units name="concentration_units">
    <unit prefix="milli" units="mole" />
    <unit units="litre" exponent="-1" />
  </units>

  <units name="flux_units">
    <unit units="concentration_units" />
    <unit units="second" exponent="-1" />
  </units>

  <units name="rate_constant">
    <unit units="second" exponent="-1" />
  </units>
```

Figure 10 – Units definition in CellML

```
<component name="intra_cellular_space">
  <!-- the following variables are used in other components -->
  <variable name="Na" public_interface="out"
    units="concentration_units" />
  <variable name="Ca" public_interface="out"
    units="concentration_units" />

  <!--the following variables are imported from other components-->
  <variable name="time" public_interface="in" units="second" />
  <variable name="I_Na" public_interface="in" units="flux_units" />
  <variable name="I_Ca" public_interface="in" units="flux_units" />

  <math xmlns="http://www.w3.org/1998/Math/MathML">
    <apply><eq />
      <apply><diff />
        <bvar><ci> time </ci></bvar>
        <ci> Na </ci>
      </apply>
      <ci> I_Na </ci>
    </apply>

    <apply><eq />
      <apply><diff />
        <bvar><ci> time </ci></bvar>
        <ci> Ca </ci>
      </apply>
      <ci> I_Ca </ci>
    </apply>
  </math>
</component>
```

Figure 11 – The complete definition of intra_cellular_space component in CellML. It contains two output variables, three input variables and two math equations.

CellML uses MathML notation to represent mathematical formulas (see section 2.2.2). In Figure 11, the *math* element contains two distinct differential equations:

$$\frac{\mathrm{d}(Na)}{\mathrm{d}(time)} = I\_Na$$

$$\frac{\mathrm{d}(Ca)}{\mathrm{d}(time)} = I\_Ca$$

I_Na and I_Ca are the rate of change in the amount of Na and Ca respectively. Note that the I_Na and I_Ca are received as inputs from the *membrane* component. Given the rate of change and the Δtime, this component must calculate the new amount of the Na and Ca.

At the membrane side, the Na and Ca amounts are received as inputs from both intra and extra cellular spaces. Based on those amounts, the membrane calculates a new change formulated as follows:

$$I\_Na = v\_Na * (Na\_i - Na\_e)$$

$$I\_Ca = v\_Ca * (Ca\_i - Ca\_e)$$

Where v_Na and v_Ca are constants valued at `1.0e-8` and `1.5e-8` respectively. The final piece of the puzzle is to connect the inputs and outputs of the three components together. This is done in through *connection* elements as shown in Figure 12.

Each connection element contains a single *map_component* element and several *map_variables* elements. The component_1 and component_2 attributes in map_components element help determine which variables belong to which components. In the first connection element of the CellML code shown in Figure 12, all *variable_1* variables belong to *component_1* which happened to be intra_cellular_space. For example, variable Na of intra_cellular_space is connected to variable Na_i of cell_membrane component. If we refer back to the definition of the two components, we notice that Na is an output of intra component and Na_i is an input of the membrane component. Naturally you should not connect the output of one component to the output of another component, nor connect two inputs to each other. To see the complete CellML code of the basic_ep_model discussed here, refer to Appendix A.

```
<connection>
  <map_components component_1="intra_cellular_space"
    component_2="cell_membrane" />
  <map_variables variable_1="Na" variable_2="Na_i" />
  <map_variables variable_1="Ca" variable_2="Ca_i" />
  <map_variables variable_1="I_Na" variable_2="I_Na" />
  <map_variables variable_1="I_Ca" variable_2="I_Ca" />
</connection>

<connection>
  <map_components component_1="extra_cellular_space"
    component_2="cell_membrane" />
  <map_variables variable_1="Na" variable_2="Na_e" />
  <map_variables variable_1="Ca" variable_2="Ca_e" />
  <map_variables variable_1="I_Na" variable_2="I_Na" />
  <map_variables variable_1="I_Ca" variable_2="I_Ca" />
</connection>
```

Figure 12 – The specification of inter-component connections in CellML. The outputs of intra_cellular_space and extra_cellular_space components are connected to inputs of the cell_membrane component.

2.2.4 Physiome Project

The need of XML standards for specifying the underlying mathematics of physiological models has led to numerously research projects and tools collectively known as the Physiome project. The CellML modeling language described in the previous section was developed under the Physiome project. Other modeling languages such as AnatML and FieldML [36] are currently under development under this project as well.

The Physiome project is a worldwide effort of numerous universities and research institutes to provide common means of encoding and sharing databases of physiological models along with tools and technologies to aid researcher in constructing realistic computer simulations of physiological entities [3]. One such tool is JSim that receives CellML models as inputs and displays the numerical results of the simulation in the screen. There are however no tool or research project under the Physiome project which is focused on the VR-based simulations of the physiological entities.

2.3 Existing VR Frameworks

As the VR technology is gaining more popularity, there are more frameworks introduced by the researchers and the VR industry in order to maximize the reusability and minimize the time and cost of developing new VR applications. The challenges that VR frameworks are facing today include:

- Lack of standards, in both software and hardware industries;
- Infinite variety of VR applications;
- The vast number of available VR input/output devices;

The frameworks that are introduced in this section are some of the more evolved VR frameworks available to developers. Most of these frameworks address only a specific category of VR applications; therefore while they are of great assistance in building a certain category of applications, they are of little use for those projects that have a different set of requirements.

2.3.1 SCIVE Framework

There are numerous frameworks proposed by various research institutes to address the extensibility and reusability issues in VR applications. The most notable work in this field is SCIVE [5][6], a simulation framework proposed by Latoschik et al. that uses semantic reflection, a concept for modular design of intelligent applications.

The SCIVE framework uses what is known as a Knowledge Representation Layer (KRL) which contains all the data needed by the various modules of the VR application (see Figure 13). KRL allows sharing and updating of these data hence avoiding redundancy and duplications. As shown in Figure 13, KRL provides a common ground for all the data that is needed by the various interconnecting modules of a VR application such as user interaction, agent perception, animation and physics. Having all the involving data within a single platform has the advantage of avoid redundancy through sharing of hierarchical data nodes. For example it is common for the rendering module needs data structure for preserving the scene graph; the physics module also needs access to the scene graph however not in its entirely. It physics module does not require those scene graph attributes that define the color and appearance of the nodes. Also the physics module requires some additional information about the target graphical model such as mass, friction, elasticity and more. KRL allows all modules to share their data hence avoid redundancy while giving them the freedom to add additional data if necessary.

While SCIVE is an effective framework for its objectives, it puts great interconnectivity and dependencies on the individual modules. Furthermore, SCIVE does not address the specific needs of simulations that receive their behaviour and visual description dynamically at run-time. For such simulations, a clear separation between model and view behaviour is required. In addition, the focus of SCIVE is on data structures and it does not address alternative representations such as those of mathematical simulations.

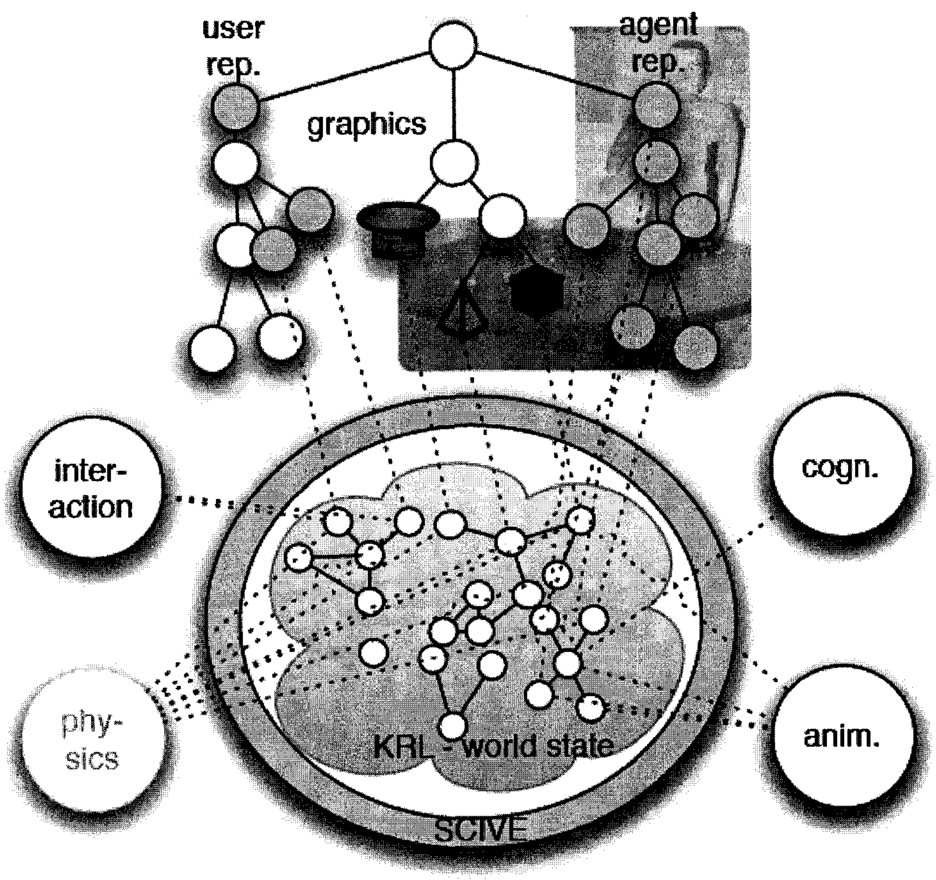


Figure 13 – Application Layout of SCIVE

2.3.2 VirtualExplorer: A Plugin-Based VR Framework

Although VirtualExplorer is designed for a limited number of tasks such as scientific data visualization and exploration, it has a flexible architecture thanks to its plugin-based design [9]. Four categories of components can be plugged into the system in order to configure and customize the VR applications that use this framework: System, Device, Utility and Application (see Figure 14). The first three are advanced plugin types and as such the users of this framework can simply pick and choose from a list of available components rather than defining them individually. The Application plugin is a user-defined component that dictates the behavior and features of the high-level application as specified by the application requirements. As such, it is the only plugin type that mandates custom definition from the users (application developers).

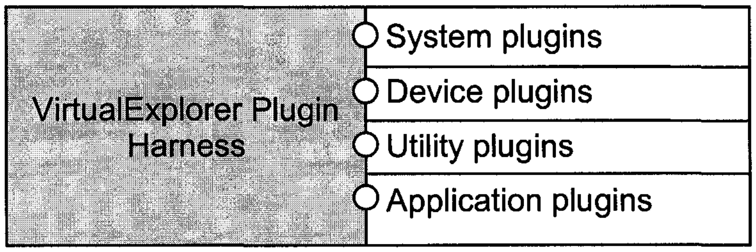


Figure 14 – VirtualExplorer plugin categories

The advanced plugins are used to specify or reconfigure those aspects of VR applications that are common and hence highly reusable. They include plugins for input devices, output devices, display devices, rendering platforms, network types, collision engines and more. VirtualExplorer has a rich library of such plugins that can be readily used by the applications that are built on top of this framework. For example, an application can use Keywork, PinchGlove or Fastrak as their input devices. Figure 15 shows the list of plugins that VirtualExplorer currently supports. Each plugin must be plugged into its own *plugin manager*. Plugin mangers are entities that provide control loop and are responsible for handling all operations associated with plugins.

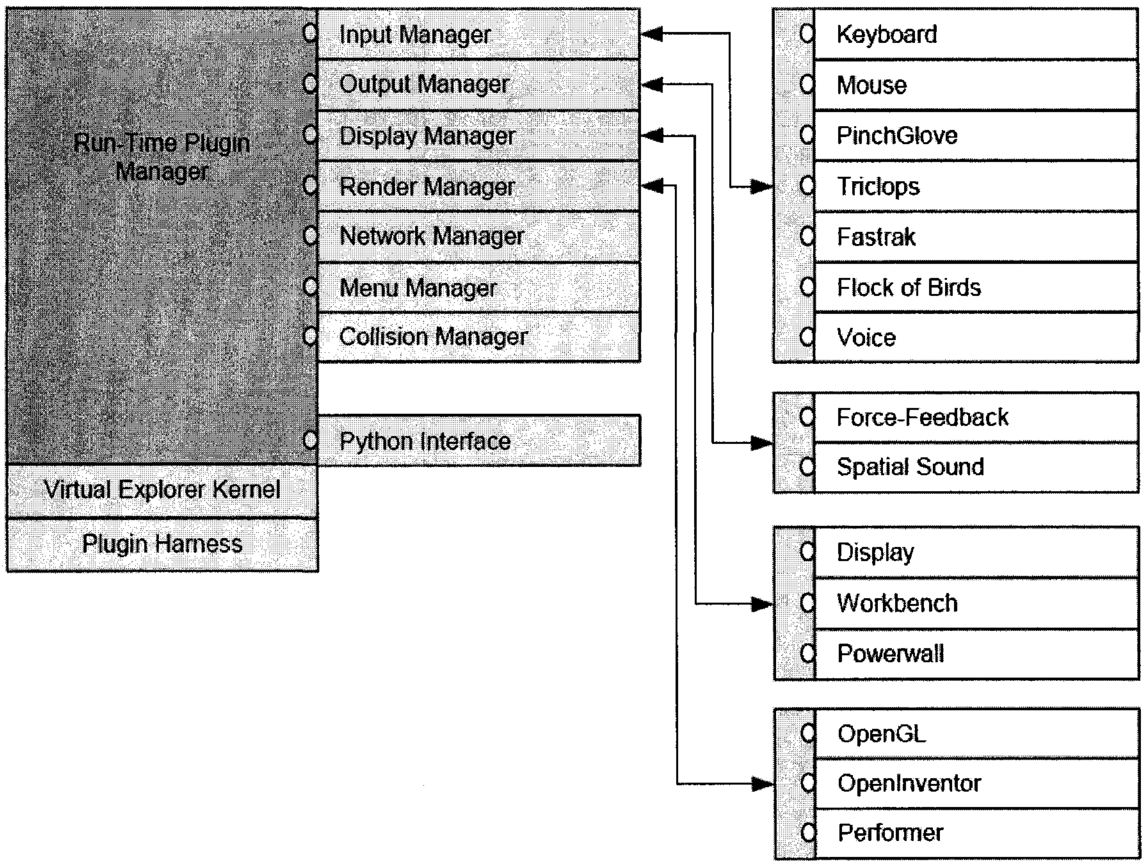


Figure 15 – VirtualExplorer plugin structure

2.3.3 ASVC Framework

There are very few frameworks that are designed specifically for surgery training applications. Researchers at the University of Akron have presented ASVC (Acquisition, Simulation, Visualization and Controller), a simple framework for integrating heterogeneous virtual surgery modules [10]. The ASVC system consists of three main modules: Data Acquisition, Simulation and Visualization. The Data Acquisition module is responsible for reading data from input devices periodically. These data typically represent the force and orientation of a surgical tool held by the user. The Simulation module continuously reads these data and generates simulation result such as a tissue deformation or the cutting of a flesh. Finally, the Visualization module uses the outputs generated by the Simulation module to render the scene. Figure 16 shows the arrangement of these modules within ASVC framework. The Controller module is in charge of the communications among the three main modules and also the communication with outside world.

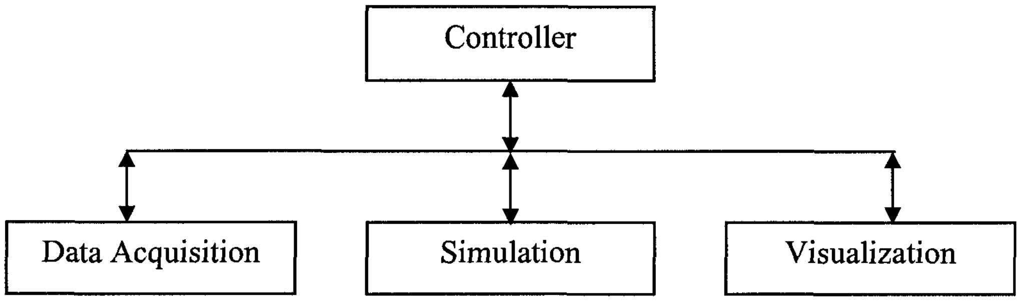


Figure 16 – The ASVC framework

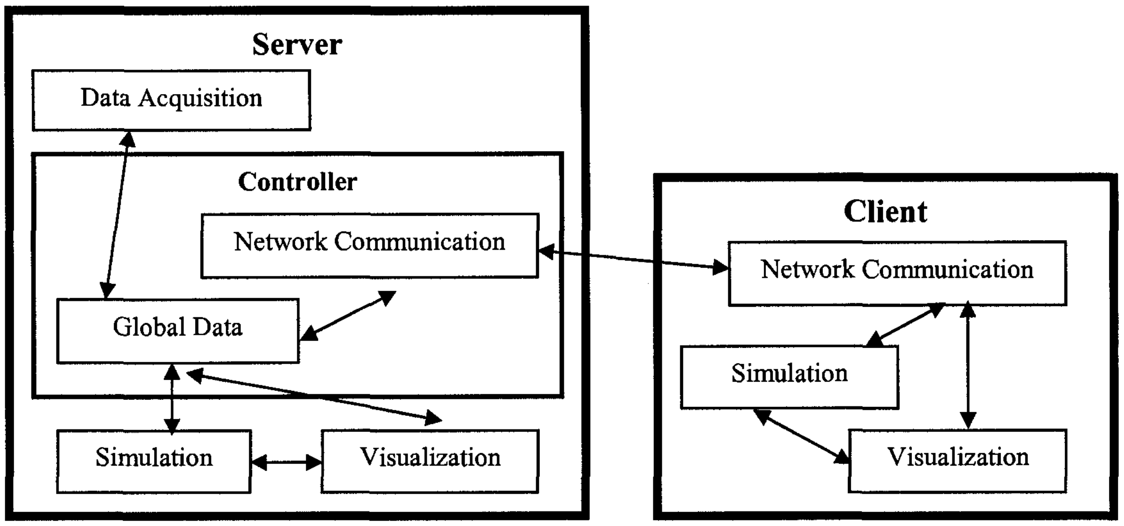


Figure 17 – The ASVC server-client architecture

The ASVC framework supports for Client/Server applications in which the client(s) observer the surgery procedure performed by the server. On the server side, the networking module continuously receives interaction data from Global Data module and sends them to clients (see Figure 17). On the client side, there is no Data Acquisition module; instead all input data are received by the networking module and forwarded to the Simulation module. Since the Simulation module is the same on both sides, the resulting visualization will be the same on both sides. Sending interaction data as suppose to the entire geometry data saves bandwidth resources and makes real-time performance possible.

2.3.4 DIVERSE Framework

DIVERSE (Device Independent Virtual Environments – Reconfigurable, Scalable, Extensible) is an open source framework that was introduced to specifically address the problem of *device dependency [21]*. DIVERSE introduces a modular framework allows VR applications to run independent of the underlying platform and connecting devices. It uses reconfigurable modules to conveniently allow users to select different input/output sources without having an impact in the application implementation. For example, the output device can be a typical non-immersive desktop monitor or an immersive CAVE or HMD system. Similarly, the inputs can come from keyboard, mouse, 3D sensors, or haptic devices.

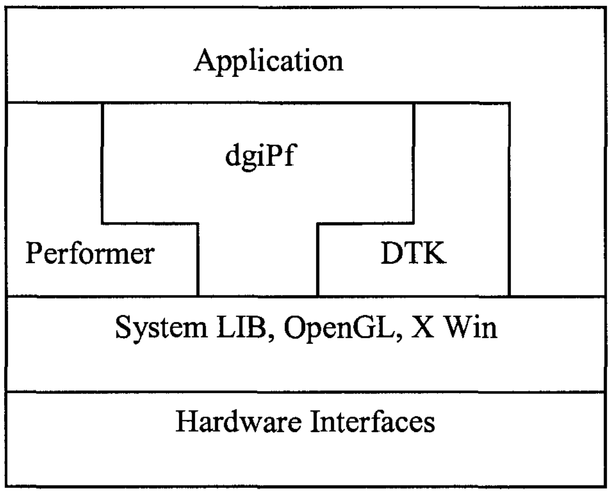


Figure 18 – DIVERSE Layered Architecture

Figure 18 shows the layered structure of the DIVERSE system. The objective of this system is to separate the Application layer from the Hardware Interface in order to allow hardware reconfiguration without affecting the application itself. The basic idea is that application does not care where the input comes from and where the output goes to; it simply allows DIVERSE to take care of those issues.

As it is shown in Figure 18, DIVERSE minimized the device-dependency through dgiPf and DTK modules. DgiPf is the graphical module that is built on top of OpenGL performer. This library contains a Display module that can be reconfigured to channel the application output to either non-immersive desktop or immersive CAVE or HMD systems. DTK is in charge of non-graphical tasks such as peripheral hardware services and networking. It contains an Input module that can be configured to use different input sources. The input source can be a hardware device or a software program that simulates a hardware device (useful for testing).

CHAPTER 3 - GENERAL ARCHITECTURE

This chapter will introduce a novel architecture that uses XML technology to provide a flexible mean of defining, constructing and customizing interactive simulations in VR. First we will discuss some of the fundamental requirements of the interactive VR-based simulations. We will then introduce AVRA, a dynamic architecture that addresses these requirements by incorporating a variety of dedicated and interconnected infrastructures.

3.1 Overview of the Requirements

Computer simulations often involve a variety of hardware and software technologies, some of which are more common than others. However at the most abstract level, almost all simulations are consisted of two categories of specifications: the *model* and the *view*. The model defines the numerical aspects of the simulation and the view defines the visual aspects of the model. Before one can develop any given simulation, the nature of the model and view must be determined.

The model may be a collection of data that is generated through scientific experiments or is collected from some sensors for example. Alternatively, the model can be a set of mathematical formulas originated from the laboratory observations or some theoretical predictions proposed by the scientific communities.

The view of a simulation specifies the visualization of the simulation results from the time it starts until the time it ends. It can be anything from a simple 2D chart to a sophisticated 3D scene graph in a virtual environment. More often than not, several different views are employed to display the various aspects of a single simulation model. Consider the three different views of a heart model as shown in Figure 19. Although all three views are used to visualize a same model (the heart), their focus is on different aspects of the model. The image on the top is an electrocardiogram graph generated in real-time by *Body Simulation*, a software application developed by RTI International for the purpose of training first-aid workers [26]. The middle image is an application developed by the University of Ulm that utilizes 3D anatomy and animations for the purpose of visualizing the heart [24]. The bottom image belongs to a haptic-enabled interactive surgery simulation that trains new surgeons through means of deformable tissues and laws of physics [25]. What these three applications have in common is that all of them implement the mathematical model of a heart. Their only difference is that they channel the numerical outputs of the heart model toward different views to meet the particular visual needs of their applications. Our proposed architecture addresses this redundancy problem by making a clear separation between the views and the models, thus making them reusable.

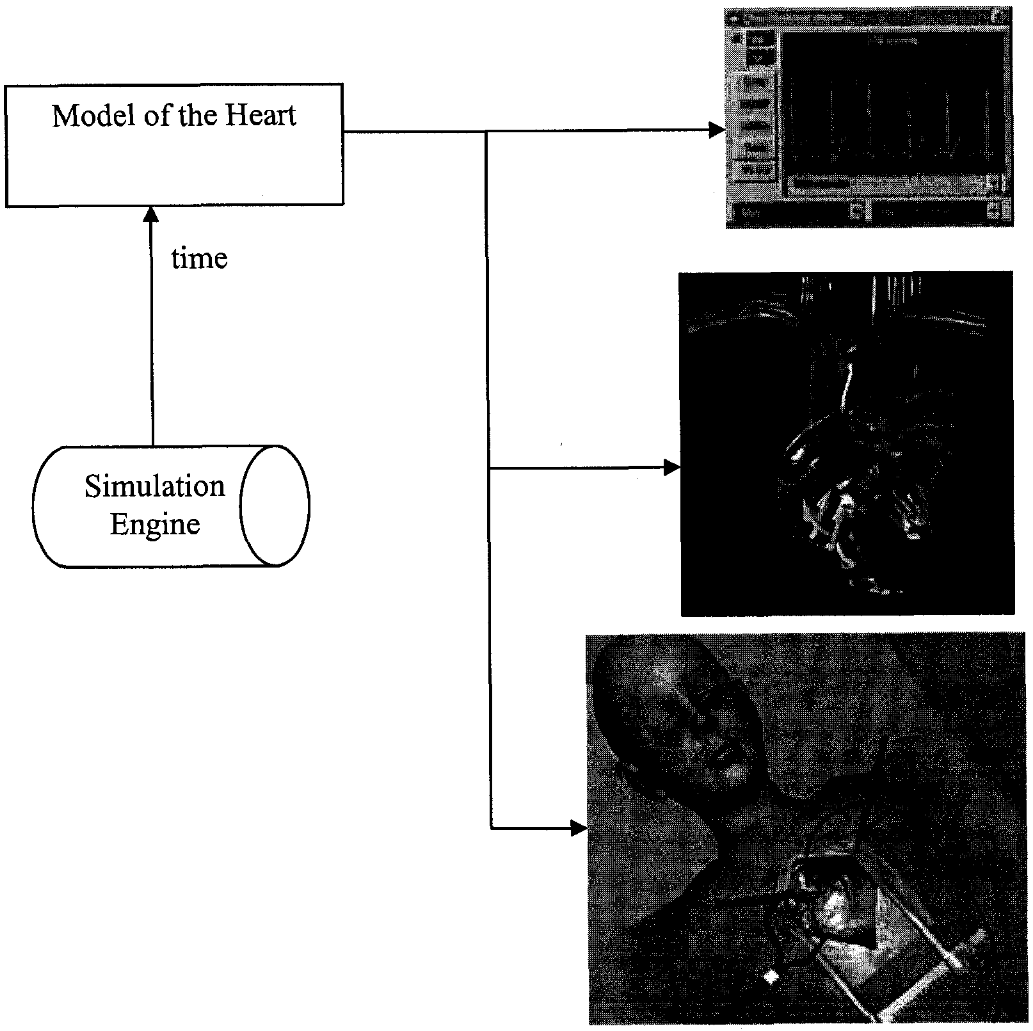


Figure 19 – Different views of a same model (the heart): The electrocardiogram graph, the 3D animation of heart beats, and a complex interactive surgery simulation [24].

In addition to model and view, simulations may have interaction elements. The *interaction* element of a given simulation determines (1) what kinds of interactions are allowed, and (2) how the system should respond to those interactions. A simulation may allow *haptic* interaction with the view contents for example (i.e. touching the 3D model of the heart). As a response, the heart may experience an electrical shock or it may naturally deform. Unlike models and views the interaction elements are not pre-deterministic and are often implemented in the higher level applications.

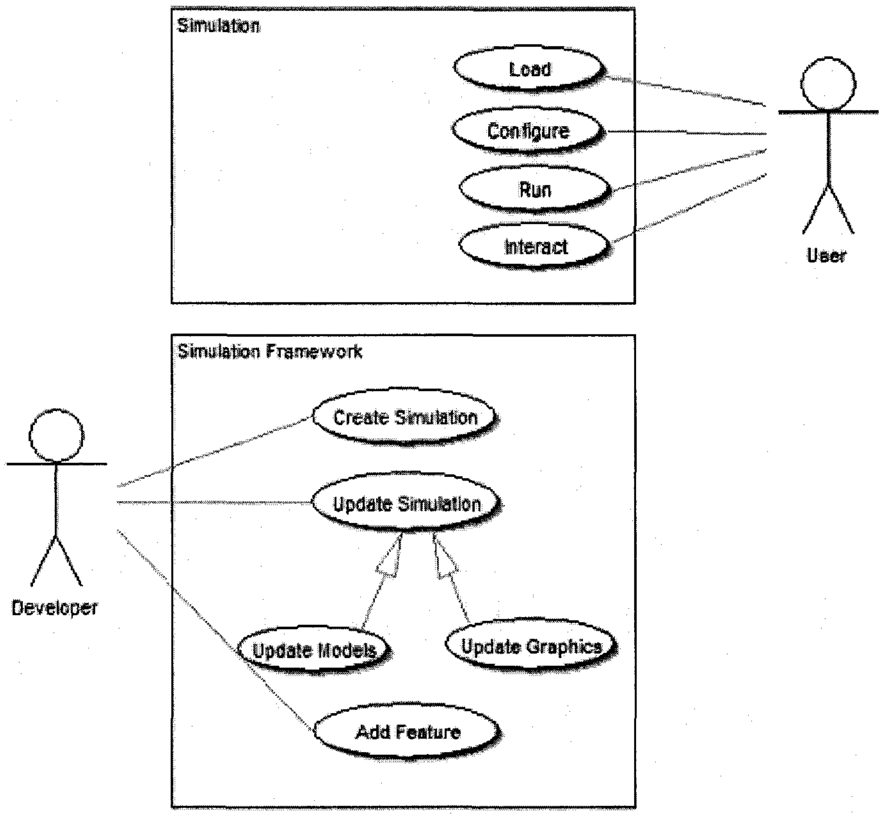


Figure 20 – Use-case diagram of simulations built based on existing frameworks

Figure 20 shows the use-case diagram of a typical simulation system under a traditional framework. As it is apparent from the figure, the end user is capable of applying a limited set of configurations through those features that are implemented by the developer(s) of the system upon request. These configurations however are typically parametric (i.e. specifying the initial Na and Ca quantities in a neuron cell model under simulation) hence not capable of changing the underlying mathematical and visual behaviors. In order words, updating the models and graphics of the simulation falls under the responsibilities of the core simulation framework which requires considerable changes and the expertise of the original developer(s) of the system. Considering how frequently the mathematical and visual models of VR applications change during the lifetime of the application, the need for a highly configurable simulation framework is apparent.

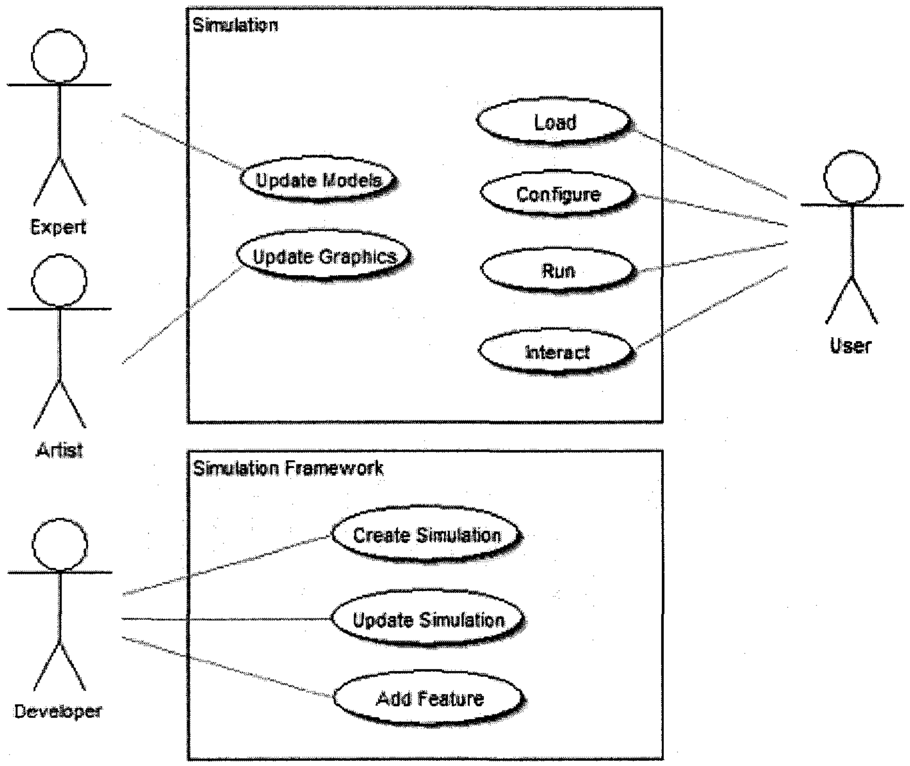


Figure 21- Use-case diagram of simulations built based on the proposed framework

As shown in Figure 21, under the proposed framework (AVRA), the mathematical and visual configuration tasks are highly configurable through XML files that can be conveniently fed into the system by anyone, taking the burden off the developer's shoulder. Take the simulation of an open heart surgery for example. The end user of such system may be students that use the simulation for training purposes. In that case, the expert is a heart surgeon or a heart researcher that updates the mathematical model of the heart upon new discoveries in the field. The artist is a graphics designer that constantly develops new anatomical models that are visually more appealing and/or more accurate.

3.2 Layered Architecture

AVRA is a framework that generates VR-based simulations for predefined models. More specifically given a simulation model and a desired view description, AVRA can generate a virtual scene in which visual objects change and interact according to the numerical outputs of the model. Applications that are built on top of AVRA can simply feed the framework with a set of model and view descriptions in XML and then start the simulation. Alternatively applications can control the simulation or add new functionalities by extending the existing capabilities of AVRA.

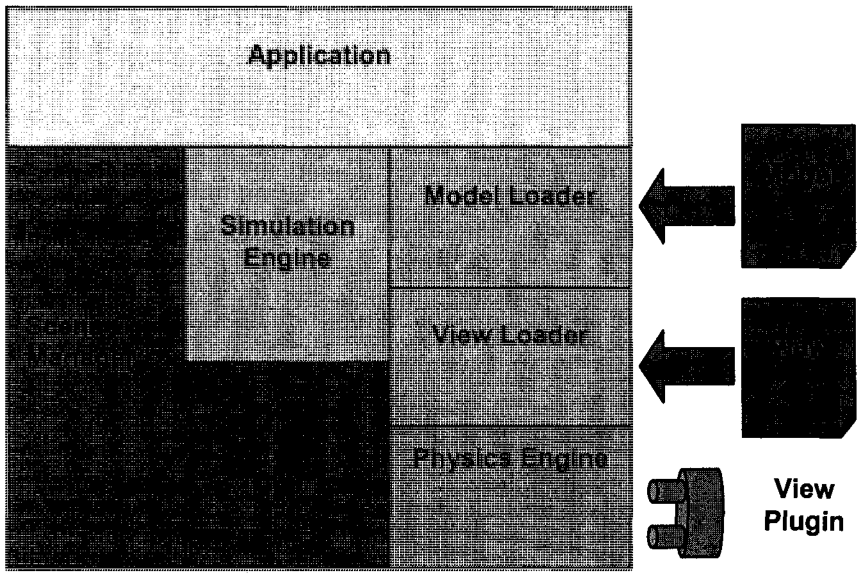


Figure 22 – The layered architecture of AVRA.

As shown in Figure 22 the architecture of AVRA consists of several main components with which the higher level application interacts. Each of these main components consists of a set of static or dynamic sub-components that will be discussed in depth in upcoming chapters. The model loader receives XML description of the model and generates a data structure to represent data models or a system of interconnected sub-components to represent mathematical models. When a model description is loaded and its corresponding model entity constructed it is sent to the simulation engine which maintains a list of loaded simulations, model entities and view components.

The view loader receives the description of how the previously loaded model is to be visualized through MVML, an XML format that we designed for this purpose. A MVML file contains descriptions that specify which graphical models to load in the virtual environment and how to modify the graphical models based on the outputs received from the simulation models. In addition to constructing view entities, the view loader generates connector components and view plugins with dependencies on the physics engine (details discussed in section 5.1). The physics engine has a dual role in AVRA: it generates physically realistic animations and interactions, and it plays a central role in the development of view plugins which are critical for transforming numerical outputs of models into visual effects in virtual environments.

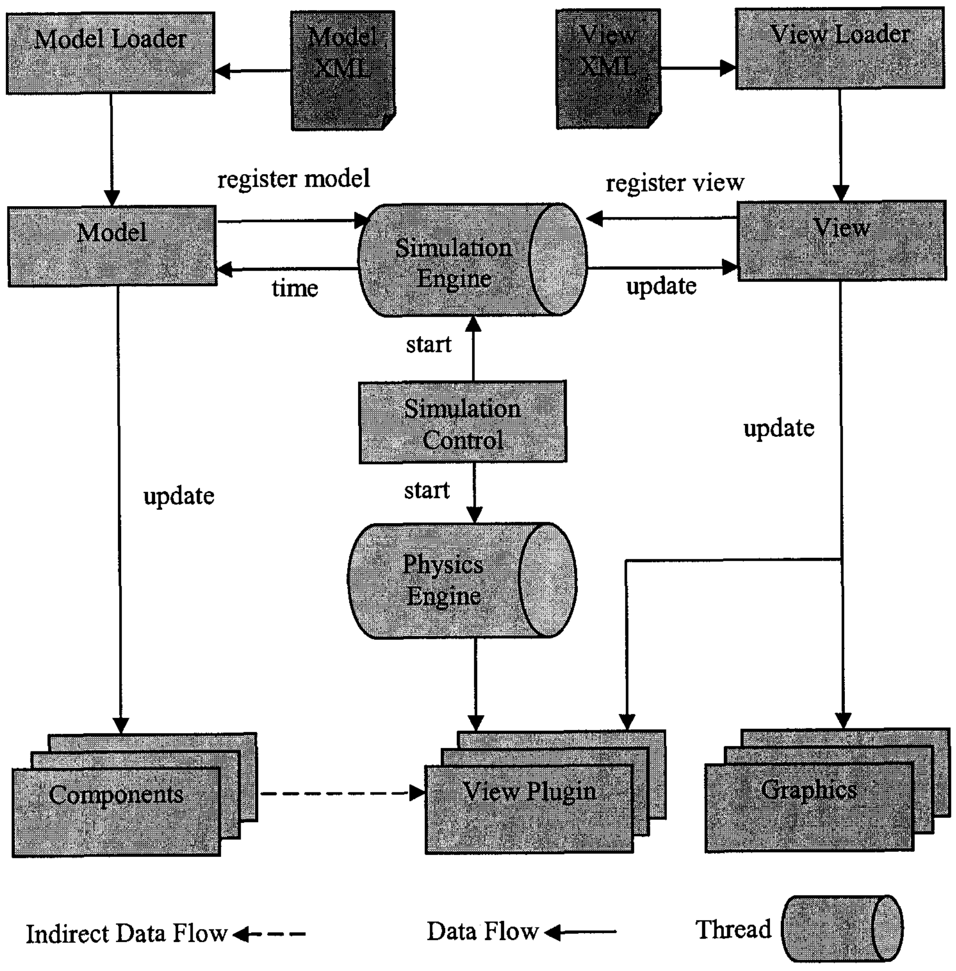


Figure 23 – Control flow within the AVRA framework

The scene manager is responsible for maintaining the graphical models as they are added or removed from the virtual environment. The scene manager is the main communication medium between AVRA and the higher level application. It allows application to introduce their own graphical models to coexist with those that are automatically generated by AVRA. Scene manager also allows the applications to retrieve the AVRA-generated entities for additional processing if necessary.

The control flow in AVRA is summarized in Figure 23. Upon loading a model a single model entity which consists of data or mathematical sub-components is created. Loading a view typically results in a view entity that has one or more graphical elements and several view plugins. The simulation engine continuously updates the models and channels their outputs toward view and view plugins. The physics engine continuously updates the view plugins which ultimately results in changes in graphical models and the virtual environment.

3.3 Model Components

The most fundamental element of the proposed framework is the simulation model that contains the data and mathematical formulas that constitute the behavior of the system. As it was discussed in section 2.2.3 CellML is currently the most popular XML format for defining the mathematical behavior of a system especially those of biological and physiological nature. Regardless of whether or not CellML modeling is used, the model components of the proposed architecture are designed to represent a series of inter-connected simulation components that compute their outputs based on given inputs and internal mathematics as demonstrated in Figure 24.

As it is shown in the diagram, the outputs of the components are directly fed into the inputs of their neighboring components. For example the outputs o1.1 and o1.2 of component 1 is linked to the inputs i2.1 of component 2 and i3.3 of component 3. This schema does not set a one-to-one limitation on the input-output linking. For example, the output o2.1 of component 2 is linked to the inputs of both component 3 and component 1.

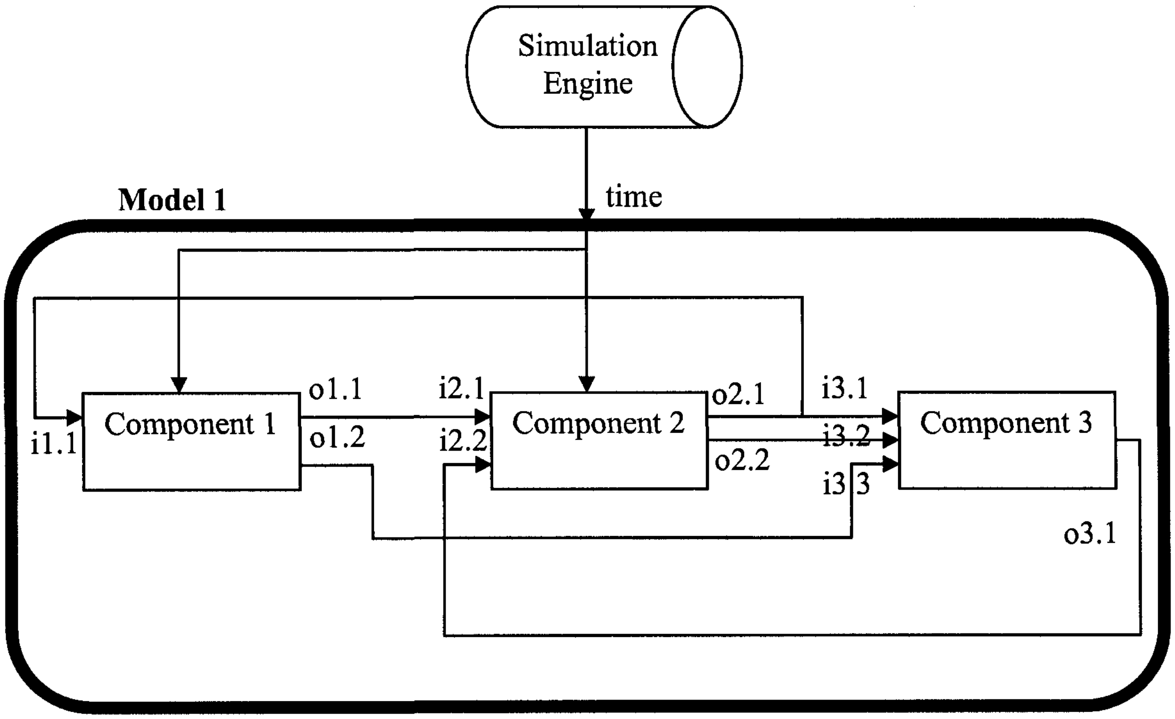


Figure 24 – The inter-connectivity of model components. The outgoing variables are denoted by a preceding 'o' and the incoming variable are denoted by a preceding 'i'.

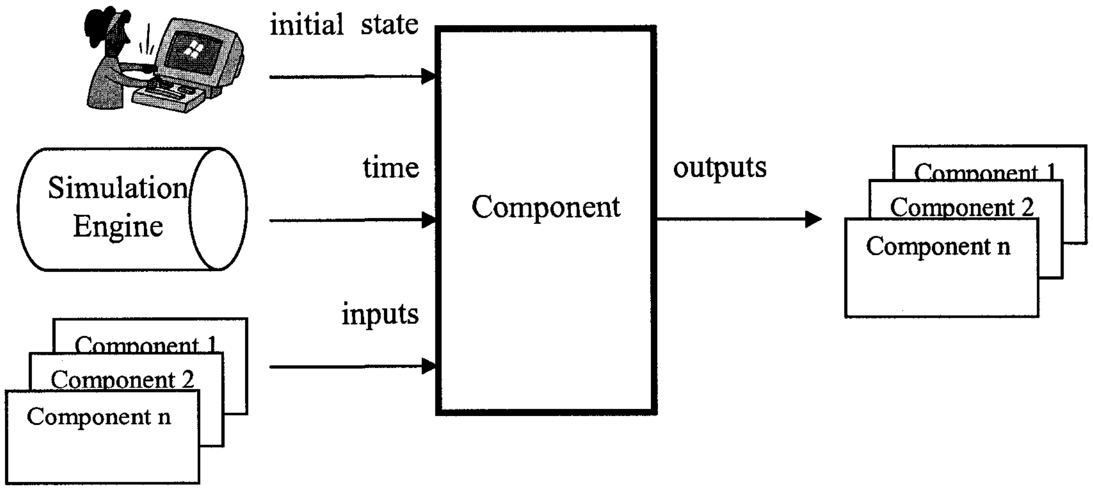


Figure 25 – The three types of inputs that components may receive.

Many of the inter-connected components in a given model are time-dependent, that is they require access to the current simulation time in order to perform their internal computations. This simulation time is generated by the Simulation Engine and made available to the simulation components at each simulation step cycle.

Figure 25 summarizes the three types of inputs that a component can receive. As it is apparent from the diagram, in addition to the time and input values, some components require initial values. This typically applies to those components that contain differential equations as part of their internal mathematics. The use of differential equations is rather common in CellML format and is often used to express the rate of change. After computing the rate of change, one needs to know the initial amount before the current amount can be calculated. Take the CellML model demonstrated in section 2.2.3 as example. It contains a component named *intra_cellular_space* that has the following formulas encoded within it:

$$\frac{\mathrm{d}(Na)}{\mathrm{d}(time)} = I\_Na$$

$$\frac{\mathrm{d}(Ca)}{\mathrm{d}(time)} = I\_Ca$$

This component receives I_Na and I_Ca as its inputs and must calculate and supply the corresponding outputs for Na and Ca. As this is a differential equation, and the time is supplied by the SimulationEngine, the only missing elements are the initial amounts for Na and Ca:

Na = init_Na + I_Na * time

Ca = init_Ca + I_Ca * time

These initial amounts are typically set by the user indirectly and through the higher level applications.

Figure 26 shows the class diagram of a system that addresses the above scenarios. The ModelLoader class is in charge of loading CellML models and generating the corresponding instance(s) of the Model class. The Model class represents a model entity that is consisted of a set of interconnected components. Each of these components is composed of a single Math object and four different sets of variables, namely inputVariables, outputVariables, initVariables and localVariables.

The local variables are internal variables that are often declared as constant (i.e. a constant variable for PI); they are merely used for the internal calculations of the given component and as such their values are not accessible through the public interface of the component. All other variables are accessible to public.

The VariableSet has a hash table data structure that quickly and efficiently finds and returns the Variable object associated with a given variable name. The Value objects are used to contain the current values of the variables. The Math object needs access to initial, local and input variables and their corresponding values before it can compute and update the values of the output variables.

Note in the above class diagram that a given variable does not "contain" a value object but is rather associated with a value object in a *shared* manner. The reason for this is to allow different variables to share a same value object. This mechanism effectively allows the outputs of one component to be linked to the inputs of another component by defining two different variables that share a same value space. Figure 38 demonstrates this configuration in our simple CellML example.

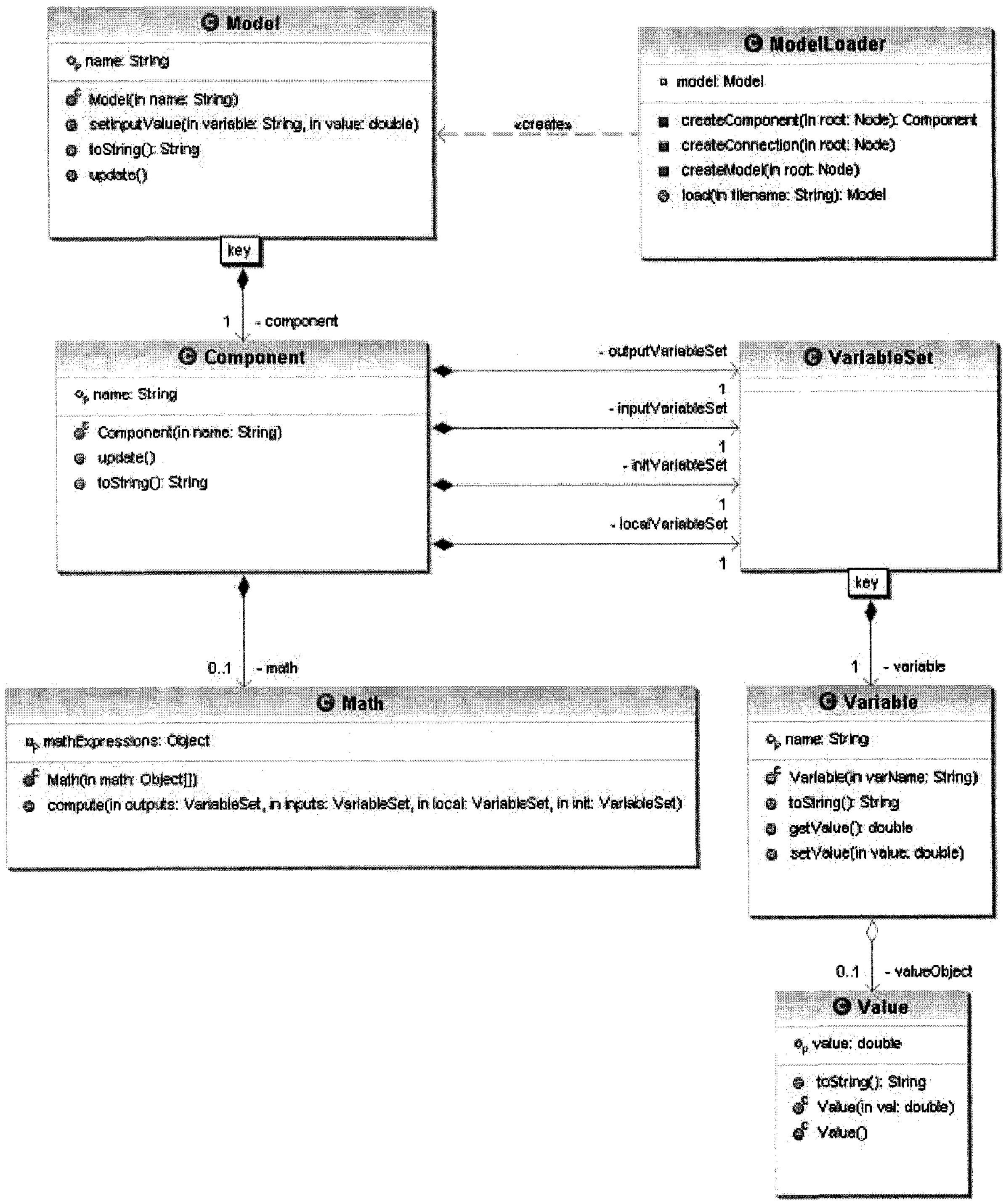


Figure 26 – The UML class diagram of the model infrastructure

3.4 View Components & Model-View Mapping

The view aspect of a simulation is more than just the graphics and visual elements. It may include everything that defines the core elements of a simulation including the graphical elements, the virtual scene, the animations and the interactions. This is why the view component along with its flexible architecture and XML-based schema is the most challenging part of AVRA.

As it was presented in the previous section, the Model component is the numerical representation of the simulation model that computes the current state of the model based on current time and possibly other incoming data. From the most abstract point of view, the view component is responsible for visualizing the current state of the model. From an implementation point of view, this translates to receiving the numerical outputs of the model and mapping them to the various attributes of the visual objects, dynamic animations or physical interactions. For example, given that the numerical output of a model is 0.5, the simulation engine may interpret it as:

- The current position of a corresponding 3D object is to be incremented by 0.5 units in the X direction;
- The scale of a corresponding 3D object is to be set to 0.5;
- The speed of a heart-beat animation must be reduced by 50% times;
- The universal gravity is to be incremented by 0.5;
- The velocity of a Newtonian object is to be incremented by 0.5;
- The density of 3D particles within a given environment is to be 0.5 particles per squared inch;
- etc.

Indeed any of the above scenarios may be the desired behavior of the intended simulation. Considering how different these behaviors are in term of implementations, such drastic changes to the view components often require re-implementation of the entire application. It is therefore evident how valuable it would be to have a generic framework that permits customization of the view components to such extend.

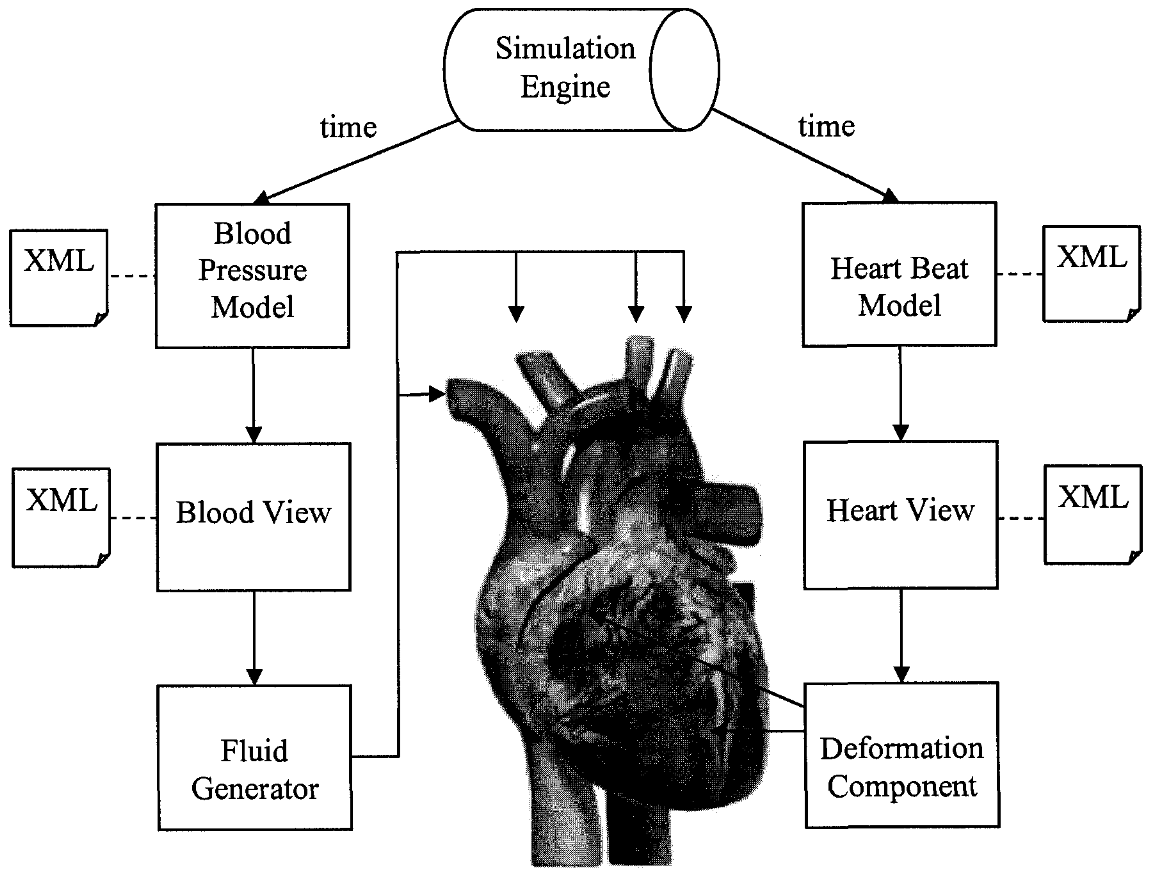


Figure 27 – Model-View configuration in a sample heart simulation that is generated base on two models XML files and two views XML files.

Figure 27 shows the configuration of a sample application that utilizes both model and view XML files. The heart beat and blood pressure models are constructed based on the contents of their corresponding model files. The heart and blood views are similarly constructed based on the attached view files. The heart view is connected to a deformation component during the initialization phase. This connection is something that is specified and configured in the view XML (MVML) and can be customized as such. Indeed as it is shown in the diagram, the blood view has a different connection configuration (it is connected to a Fluid Generator).

Once the simulation is started, the current time is continuously distributed amongst models, as a result of which the models generate new outputs. These outputs are received by the view models and translated into parameters that are understandable by the graphical modules such as the deformation and the fluid generator components. Subsequently those components will modify the visual model of the heart to simulate the fluctuations in the mathematical models. For example, the deformation component would continuously deform the polygon mesh of the lower and higher segments of the heart to animate a heart beat. The fluid generator on the other hand is in charge of regulating the float of the blood particles through the veins. It does that by generating or removing the blood particles and changing their speed or color.

3.5 Physically Realistic Views

When it comes to visualizing the outputs of the simulation models, one can simply trigger a pre-designed animation. Alternatively physics engines can be used for dynamic and physically realistic effects. Physics engines are libraries that enforce laws of physics in a scene by continuously updating the position, orientation and shape of the visual models based on their physical attributes such as mass, velocity, force, friction, aerodynamics and more.

There are many physics engine libraries available today some in the form of commercial products and some are freely available as open source projects. AVRA uses xPheve (Extensible Physics Engine for Virtual Environments) which is particularly beneficial due to its extensible architecture and the use of pluggable physics laws [2][35]. In AVRA physics laws form the basis of *view plugins* that are in essence those components that receive parameters from the mathematical models and subsequently translate them into visual animations.

Figure 28 shows the general architecture of xPheve. The diagram shows a multi-threaded implementation of xPheve that consists of the application thread, the physics engine thread and the physical law thread. The application thread initializes the physics engine and may occasionally modify the physical attributes of objects. The physics engine thread continuously updates the graphical attributes of physical objects such as the position, orientation and shape (i.e. for deformation) of the physical objects. The physical law thread continuously updates the physical attributes such as force, velocity, momentum, etc.

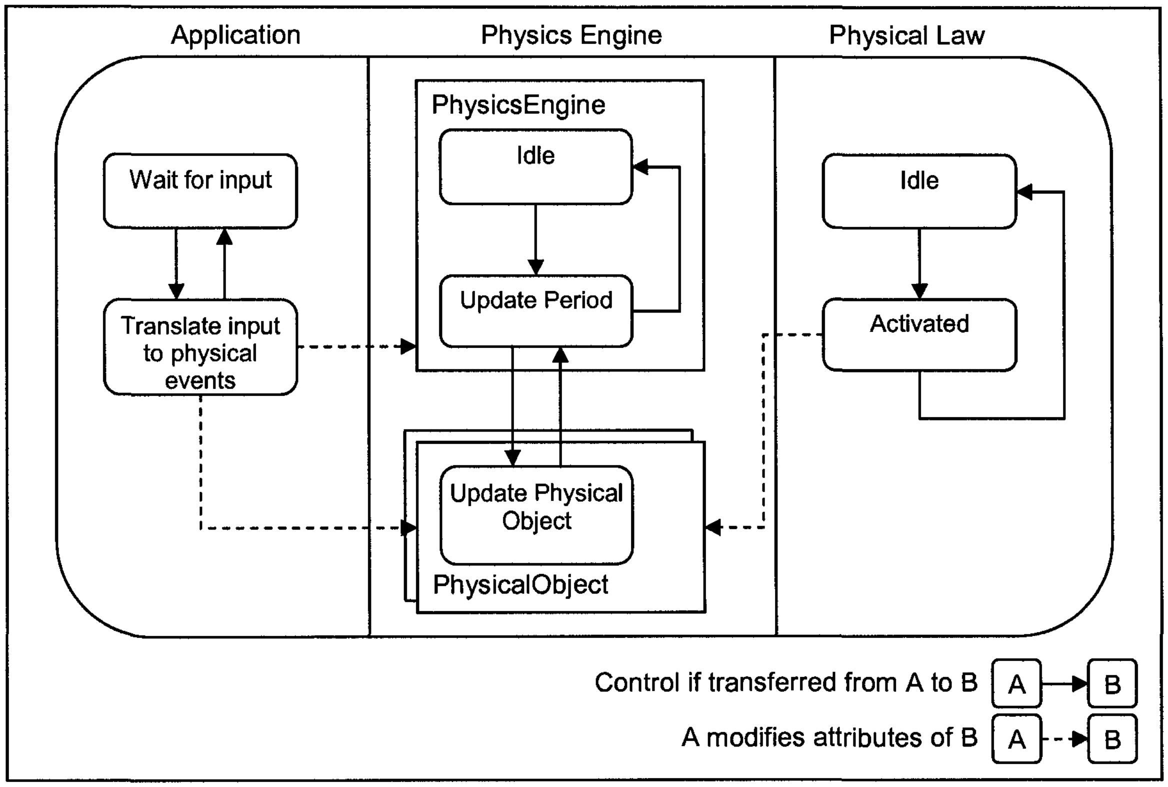


Figure 28 – The xPheve architecture [2]

3.6 Interactive Views

Today VR applications employ a wide variety of technologies and devices in order to stimulate the various senses of human body including the sense of touch. The haptic technology as described in section 2.1.2 utilizes force feedback arms and sensor gloves in order to present a realistic 3D interaction experience. Not all potential users have access to such interaction devices however. In addition, different applications utilize these input devices for different purposes. For example, in our simple cellular model, haptic device may be used to alter the concentration of Na and Ca inside and outside of the cell's subspace. In a heart surgery simulation, the haptic device may control a virtual knife that can cut through veins and flesh, indirectly causing disturbances in the parameters of the blood pressure and heart beat models as shown in Figure 29. All these scenarios encourage a flexible architecture through which the haptic (along with other input/output devices) can be configured with ease.

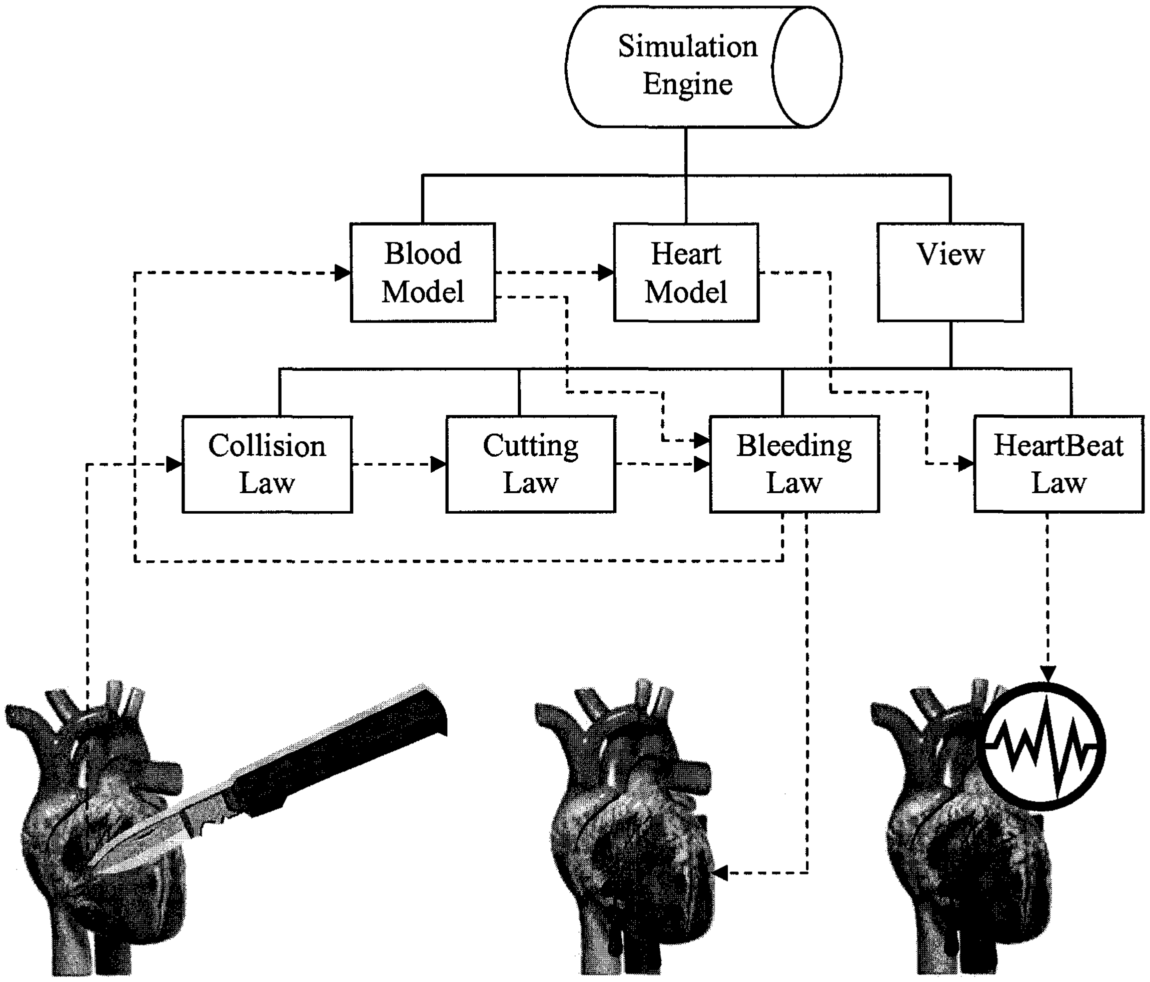


Figure 29 – A sample heart surgery simulation configured with model and view XML. The model and view components affect each other in a cascading manner.

There are two models loaded in the simulation shown in Figure 29: the heart model and the blood model. The view instance contains four elements for performing collision detection, cutting, bleeding and animating heart beats. Up until this point there is nothing distinctive about this diagram. Note however that the outputs of some of the view elements are fed back into the model elements giving the view element the capability to alter the outcome of the mathematical models and hence the entire simulation. In particular the user action of forcing a knife on the surface of the virtual heart causes a collision event that triggers a visual cutting of the mesh and a bleeding effect through means of fluid particles. The rate of this bleeding however depends on the output of the blood model. Once the bleeding is started, this fluctuation in blood supply is reported back to the blood model which recalculates its outputs, as a result of which both bleeding view and heart beat model are affected. Therefore the effect of the cutting is ultimately felt as visual fluctuations in the heart beat animation.

3.7 Summary

This chapter presented the general architecture of the proposed AVRA framework which uses XML technology for describing the model and view elements of VR-based simulations. Those systems that are built based on this proposed framework are highly customizable through their XML configuration files. In particular, the details of the model, graphics and animation can all be specified through these XML files. The generic simulators that use the proposed architecture will read these XML descriptions during their initialization and dynamically generate a functional and highly customizable simulation.

CHAPTER 4 - SIMULATION MODELS

There are two common ways for modeling simulations: (1) through experimental data or (2) through mathematical formulas. Both options require storage of the model description in some standardized format and the ability to reliably share them so that a given simulation model generates a same numerical [and visual] output regardless of which simulation tool or platform is used.

AVRA is designed to handle both data and mathematical models. Naturally, a different kind of model loaders and model objects are needed in order to successfully integrate different formats of simulations models. AVRA allows easy integration of custom designed model objects as long as they abide by the following rules:

- A Model object should allow values for its input parameters to be set anytime during the simulation.
- Given the current time and input parameters and the identity of an output parameter, the model should be able to compute and return the value of the output parameter anytime during the simulation.

Most simulation models considered when designing AVRA are time-dependant; that is, their internal state and external outputs depend on the time as well as other possible inputs. However those models that do not depend on time may also use AVRA by simply ignoring the time parameter.

As shown in Figure 30, in AVRA every simulation contains one or more Model objects. Each model has a VariableSet that contains a list of variables that represent the input and output parameters of that model. The Model class is an abstract class that needs to be implemented. AVRA includes two implementations of the Model: DataModel and MathModel. A DataModel has a DataTable which contains the simulation data sorted by time in an ascending order. The MathModel often has several Math Components, each with a variable set that consists of a subset of the complete list of input/output variables.

Due to the interface provided by the abstract class Model a wide range of simulation models can be represented in AVRA. If the structure provided by DataModel or MathModel is not sufficient for a particular simulation modeling format then new custom subclasses of the Model can address the additional requirements. For example if neural-network [53] simulation models are to be loaded in AVRA then a new representative subclass of Model can be constructed to address the specific structure of neural-network models.

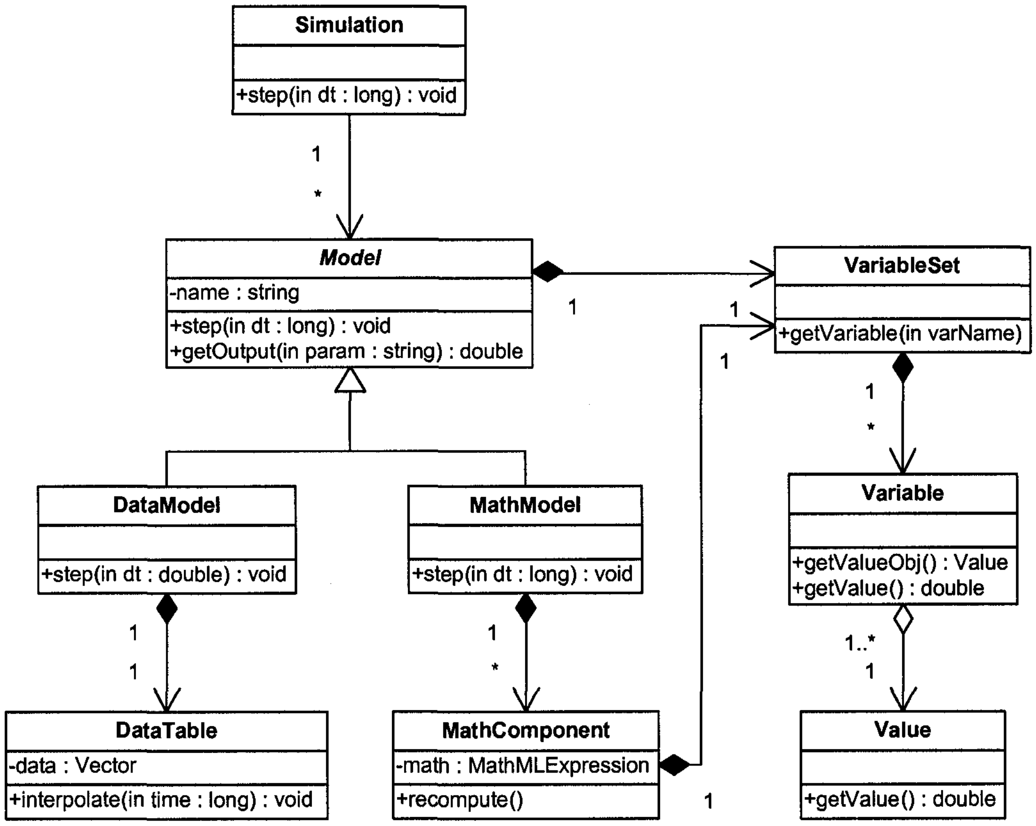


Figure 30 – The UML Diagram of Model hierarchy

The task of loading model files and constructing simulation models is a responsibility of the model loaders. As shown in Figure 31 all model loaders inherit from ModelLoader which is an abstract class with an abstract *load* method that returns a new instance of Model upon successful loading. AVRA does not put any restrictions on how a model loader is implemented as long as it inherits from the ModelLoader. For convenience, there is an XMLModelLoader available that includes the implementations for loading any XML file and generates a DOM tree that is passed to the abstract method *processXML*.

This class provides much of the functionalities needed for loading XML-based models such as CellML; however using this class for loading XML models is optional and developers may directly inherit from the ModelLoader class when developing custom loaders for XML models.

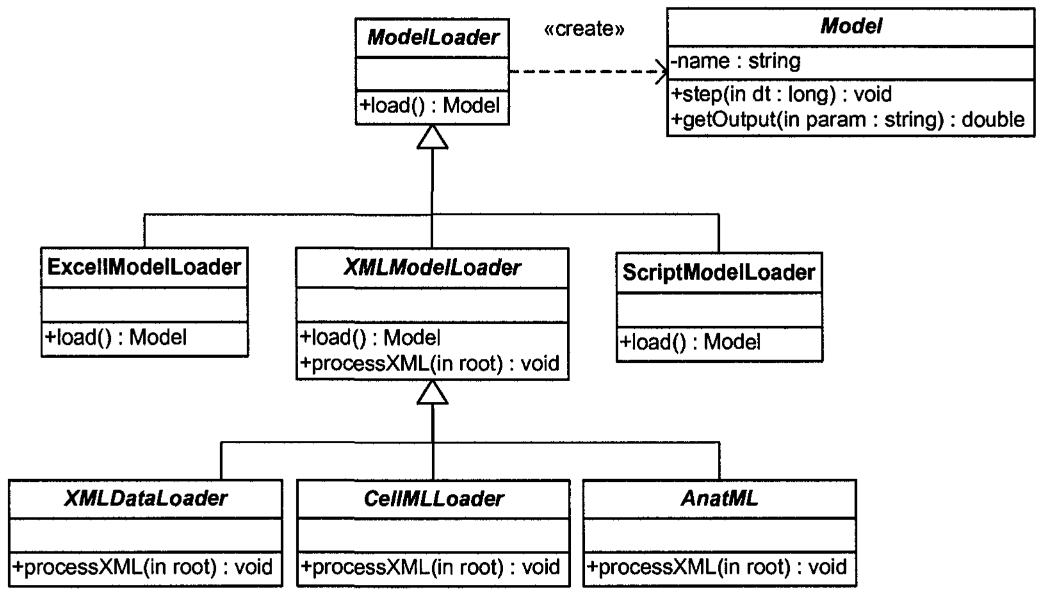


Figure 31 – The UML Diagram of ModelLoader hierarchy

4.1 Data Models

Data models are passive simulation models with pre-generated simulation data. There are many applications to data models; for example, a data model can be used to store the results of an experiment, or it can be generated manually to describe the theoretical or expected results of an experiment that has not taken place yet. Regardless of the purpose, a data model typically consists of a simple or hierarchical data table with fields that represent the input and output values of the simulation at each instance of time.

Table 2 is a sample data model that represents the simulation data of a neuron cell during action potential. The data starts from time zero and ends 10ms later. At time zero, the cell is at its rest with rest voltage of -75. Action potential is invoked by a stimulus at time 1ms resulting in an increase in the voltage with its peak at time 4ms after which the cell slowly returns to its resting state.

Time (ms)	m_gate	i_Na (μA)	V (mV)
0	0.05	-1.035	-75
1	0.051225	-1.1167	-75.3269
2	0.129692	-15.6724	-65.2911
3	0.547273	-388.366	-6.65066
4	0.996577	-463.506	15.01582
5	0.96885	-388.151	-19.2617
6	0.719583	-159.714	-51.8486
7	0.043962	-0.11764	-83.6873
8	0.015446	-0.01043	-84.949
9	0.01647	-0.01812	-84.3782
10	0.017984	-0.02951	-83.6696

Table 2 – A sample data model that constains the measurements of the gate opening of Sodium channel (m_gate), the current of Sodium channel (I_Na) and the Voltage (V) of a cell during action potential.

Figure 32 shows the graphs that are extracted from a data table similar than that of Table 2 but with a higher data resolution with data fields for every 0.01ms instead of every 1ms as is the case with of Table 2. Regardless of how high or low the resolution of a data model is, the model must be ready to supply output values even for those time values that do not have an exact match in the data table (i.e. smaller than the step size). For example, in order to generate a virtual reality application that visualizes a simulation based on the data of Table 2 over a period of 10 seconds we need a simulation step size of 0.02ms as calculated below:

Actual Time of Simulation (ATS) = 10ms

Visual Time of Simulation (VTS) = 20s

Frame per Second (FPS) = 25 f/s

Total Frames (TF) = VTS * FPS = 20s * 25 f/s = 500f

Simulation Step Size per Frame = ATS / TF = 10ms / 500f = 0.02ms/f

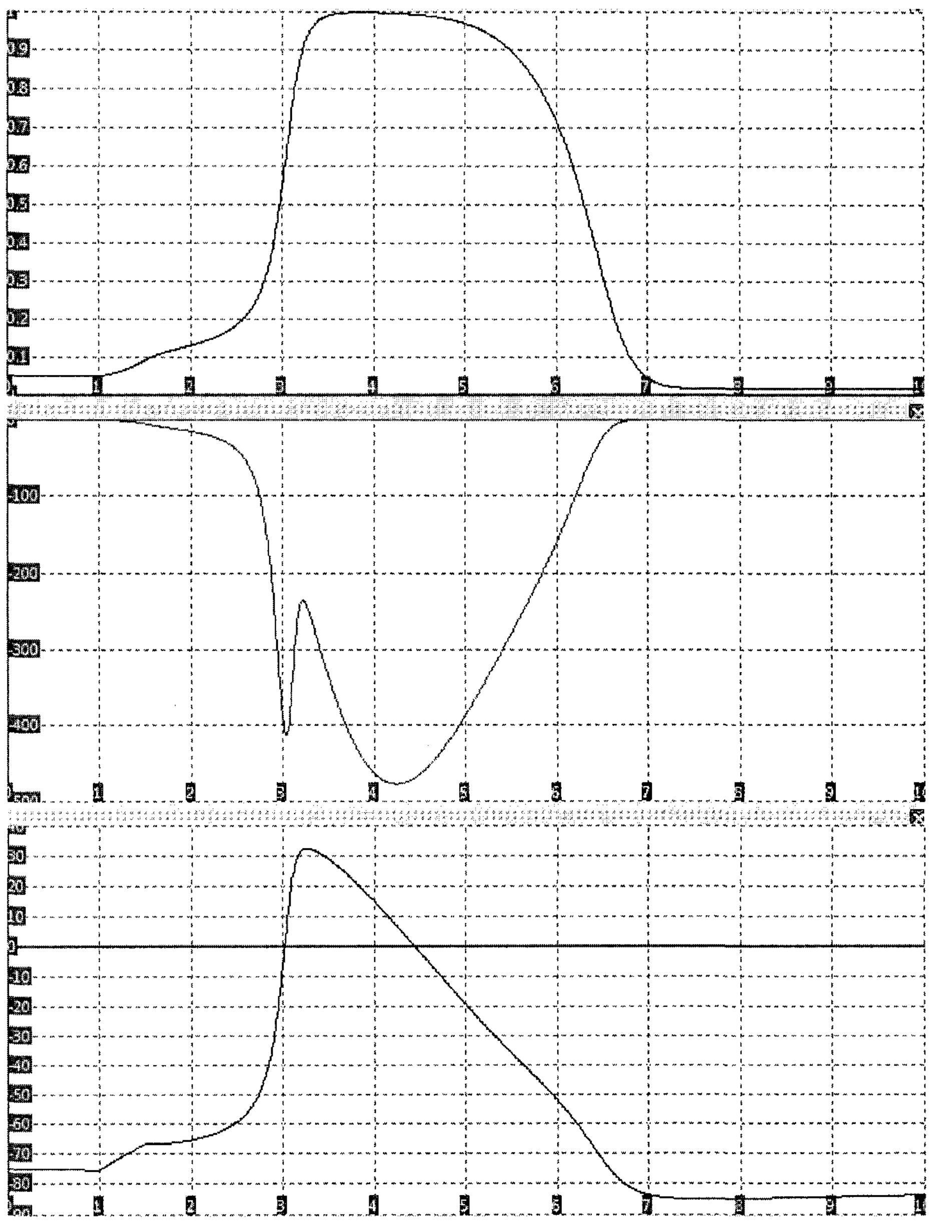


Figure 32 – The graph of m_gate (top), i_Na (middle) and V (bottom) of the data in Table 2

Since the data in Table 2 has a larger step size, the data model needs to interpolate the data in order to extract additional rows from the existing data whenever needed. Figure 33 demonstrates the effects of some of the popular interpolation algorithms available. Naturally those interpolation algorithms that generate more accurate results are computationally more expensive.

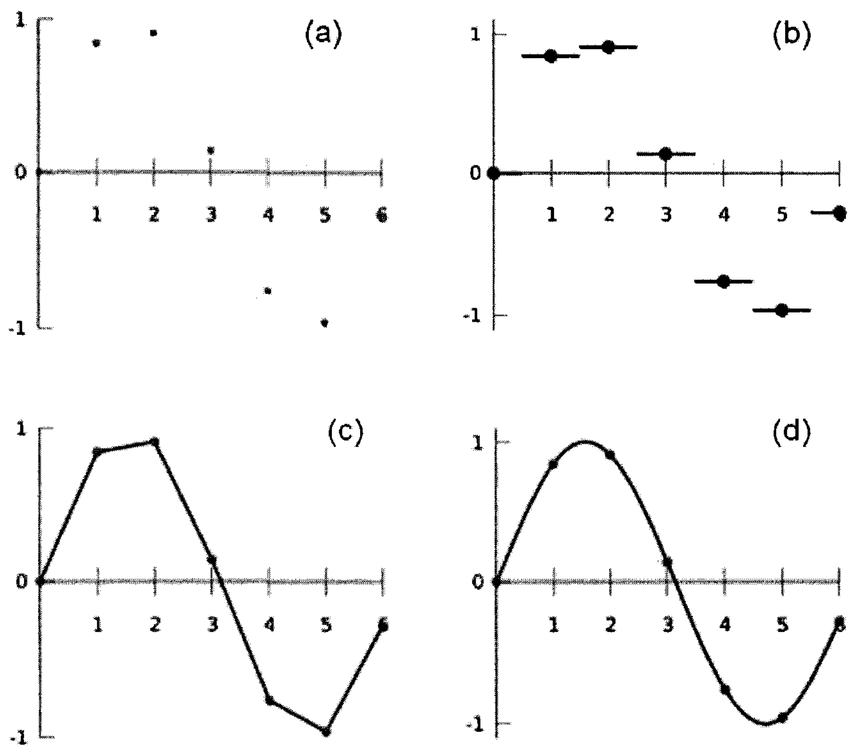


Figure 33 – Given data points in (a) new points can be extracted using (b) Piece-wise interpolation (c) linear interpolation, and (d) Spline interpolation algorithms.

In AVRA this issue is addressed in DataTable which is primarily responsible for storing the data but also have the responsibility of interpolating the data as needed. By default, the DataTable interpolates values linearly in order to balance between the quality of the extracted data and the computation cost. In addition, the DataTable is easily extendable and in effect implements the Bridge pattern [28] hence allowing custom interpolation algorithms to be implemented by the developers if the default interpolator is not sufficient. This task involves extending the DataTable class and overriding its interpolate method.

4.2 Mathematical Models

The mathematical aspects of the simulations are more difficult to be modeled as generic components since their internal structures are more complicated than the simple scheme of the tables as seen in the previous section. XML is the popular choice for encoding the mathematical behavior of simulations. There are many XML-based markup languages that are developed for the purpose of encoding the mathematical nature of different scientific phenomena in various fields such as biology, chemistry, mechanics and electronics. AVRA takes advantage of the existing popularity and presents a flexible framework that can be extended to include support for any of the existing formats.

At its simplest form, a mathematical model receives the simulation *time* as input, performs some mathematical operations and generates some outputs as shown in Figure 34. However most scientific models are more complicated than that and need to break down a simulation into smaller sub-components. Each sub-component typically receives inputs from other sub-components or from the interface of the model, applies some mathematical function to those inputs, and sends the resulting output to the other sub-components or makes them available to the outside world through the interface of the model.

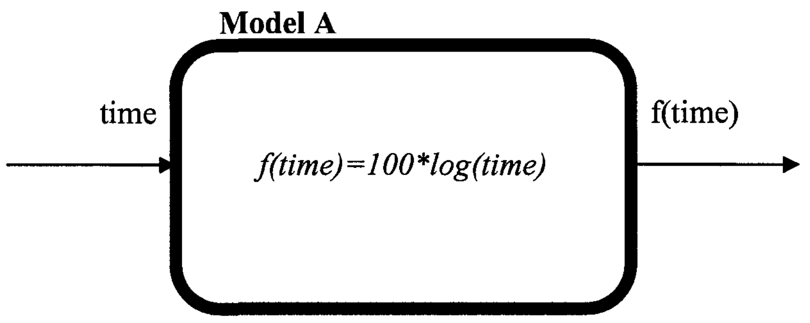


Figure 34 – A simple mathematical model that simulates a phenomenon as formulated by function f

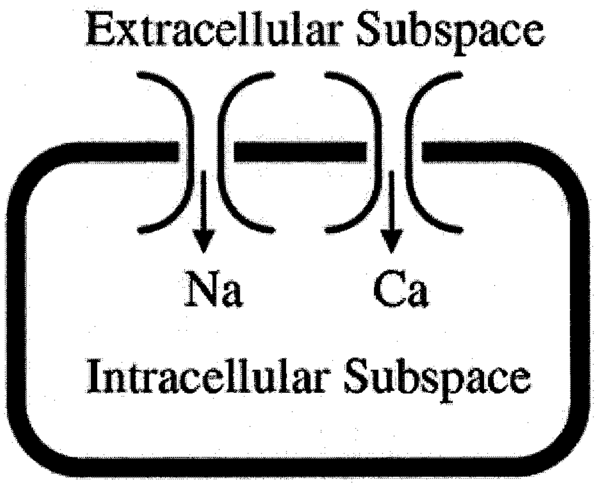
There are no unified standards that determine exactly how these sub-components must interact with each other; different modeling formats such as CellML and AnatML have different ways of representing the behavior of their models. The MathModel class in AVRA includes a default set of functionalities that presents a generic way of representing the mathematical models. When constructed by the model loaders, a MathModel object would typically contain several MathComponents each responsible for computing a smaller set of mathematical expressions and channeling the results to those components that need them. A sample mathematical model will be explored in the next section with discussions of how it can be represented in AVRA. If the structure of the MathModel is not sufficient for representing a particular model format such as AnatML or FieldML then developers may extended it to address any extra structure that may be required.

4.2.1 A Sample Mathematical Model

To demonstrate how AVRA represents mathematical models we take the simple model of a neuron cell which was demonstrated in section 2.2.3 as example. This model consists of a simple cell that contains some pre-defined quantities of Na and Ca that flow in and out of the cell through the cell membrane. The model which is encoded in CellML format contains mathematical formulas that govern the rate at which Na and Ca ions move across the membrane. As it was discussed in section 2.2.3 CellML models typically consist of several *components*. These components translate to MathComponents in AVRA. In essence, upon loading the sample CellML model, AVRA generates an instance of Math Model (named Simple Cell Model in this case) as shown Figure 35. It contains three math components each of which consists of two mathematical formulas. Each of the three math components receives parameters as inputs from the other math components and, upon computing the results, sends some outputs to other math components.

This complex system of interconnected components and differential equations is common among scientists especially those of biological fields; hence AVRA is designed to handle them as such. Figure 36 shows how instances of MathModel and MathComponent can be constructed in order to effectively represent the simple cell model. As it was shown in Figure 30 each instance of MathComponent is associated with a VariableSet which ultimately contains all the variables needed including the input and output parameters. For example, Extra Space which is a MathComponent has 4 variables, two of which are inputs and two of which are outputs. A same instance of VariableSet is used because unlike CellML, AVRA does not need to differentiate between inputs and outputs. As far as AVRA is concerned the task of forwarding outputs to inputs is automated once the components are properly connected together. The process of connecting the math components together is discussed in the next section.

(a)



(b)

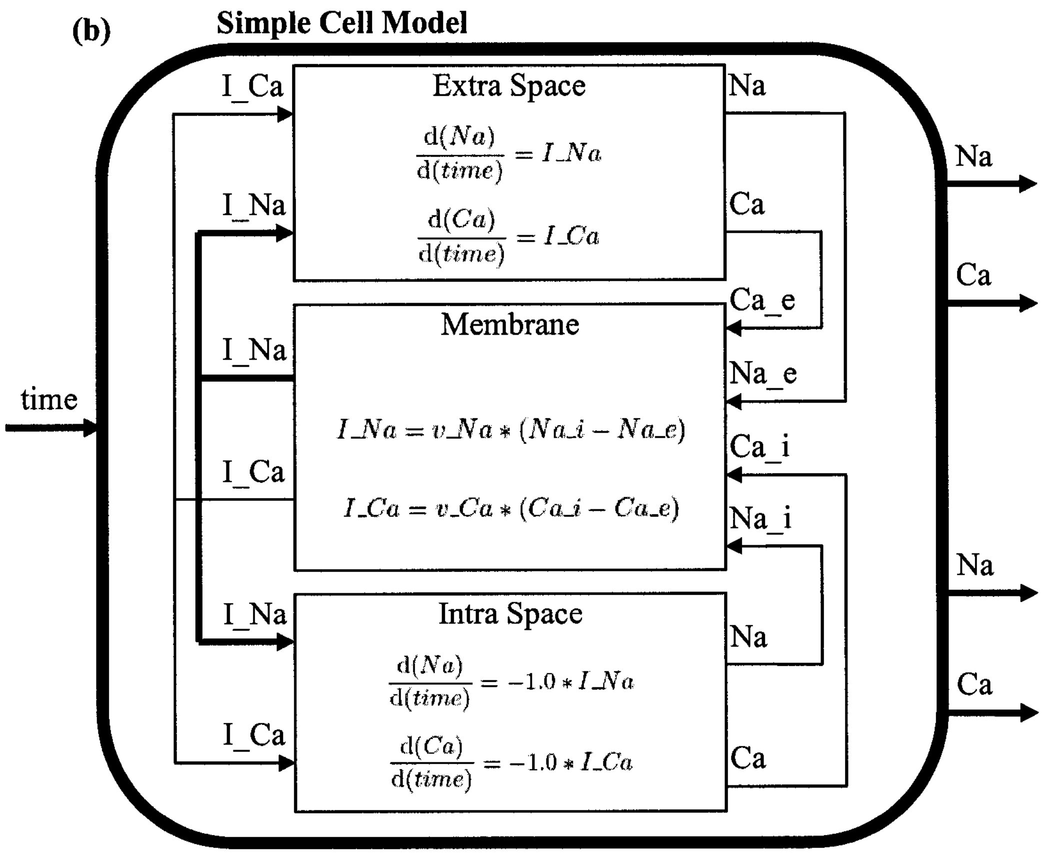

Simple Cell Model
time
I_Ca
I_Na
I_Na
I_Ca
I_Na
I_Ca
Extra Space
$\frac{d(Na)}{d(time)} = I\_Na$
$\frac{d(Ca)}{d(time)} = I\_Ca$
Membrane
$I\_Na = v\_Na * (Na\_i - Na\_e)$
$I\_Ca = v\_Ca * (Ca\_i - Ca\_e)$
Intra Space
$\frac{d(Na)}{d(time)} = -1.0 * I\_Na$
$\frac{d(Ca)}{d(time)} = -1.0 * I\_Ca$
Na
Ca
Ca_e
Na_e
Ca_i
Na_i
Na
Ca
Na
Ca
Na
Ca

Figure 35 – The Math Components of a cell model (b) that represents flow of Na and Ca particles in and out of a neuron cell during action potential (a)

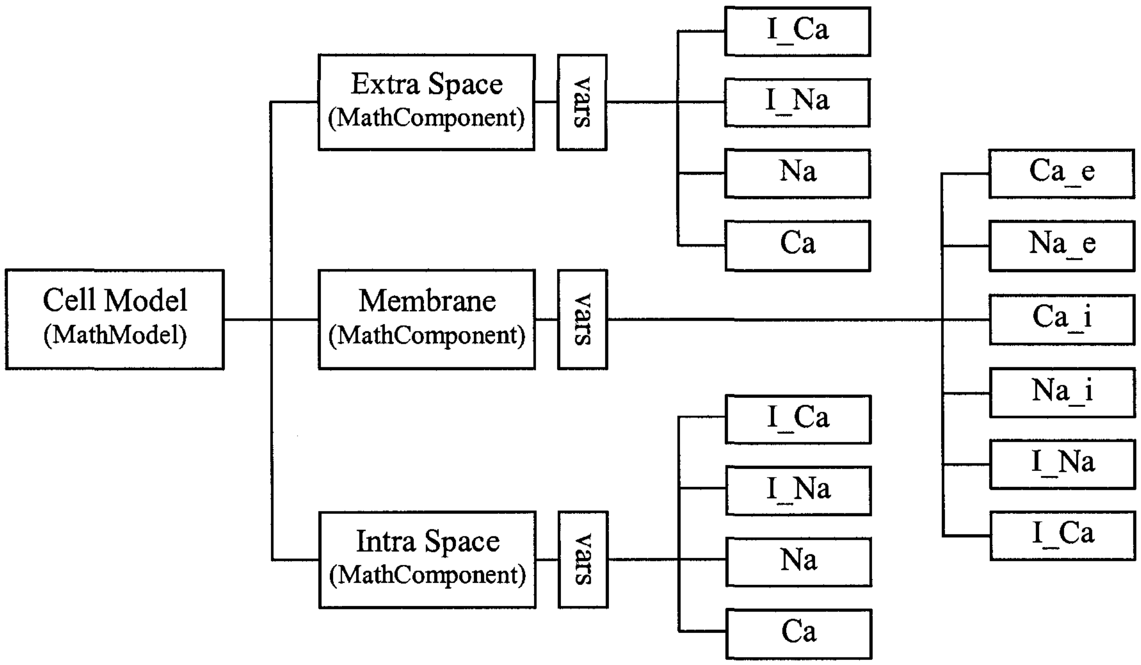


Figure 36 - The object diagram of a MathModel that is generated in AVRA in order to represent the Simple Cell Model of Figure 35

In addition to the local parameters of the math components, the math model which acts as a container also has its own set of input/output parameters. A special input parameter is *time* which is distributed among all math components. Those which do not need the time parameter may simply ignore it. In this example, both extra space and intra space math components need the time whereas the membrane does not. For other parameters, they are typically directed to one or more math components selectively and the outputs are typically received from the outputs of the interior math components as well. As it is shown in Figure 35 our model has outputs of conflicting names. For instance, there are two Na outputs, one coming from the extra space and the other coming from the intra space. To address this, in AVRA the name of the interior component must be expressed as well as the name of the output variable when referring a particular input/output parameter.

4.2.2 Connecting Math Components

Recall from the beginning of this chapter that each Model entity is associated with a variable set which consists of a list of unique Variable objects. Each variable references a Value object that may be shared with other variables; this is denoted by a white diamond (shared aggregation) in Figure 37. In AVRA this sharing of value objects allows the output and input variable to be conveniently and efficiently connected together.

As it is shown in Figure 38, there is a clear separation between the variables and their corresponding values. For example, the *intra* component has an output variable named "Na" which according to the CellML *connection* configuration (see Figure 35) is to be linked to the input variable of *membrane* component named "Na_i". Even though the two variables are different in that they belong to different components and have different names, they are both linked to a same *Value* object as shown in Figure 38.

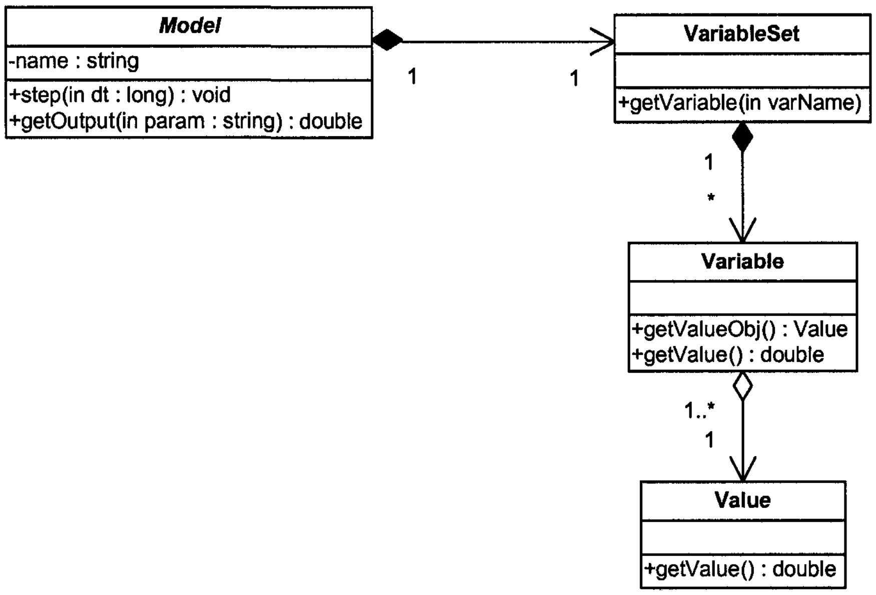


Figure 37 – Class diagram of VeriableSet

One might be concerned that such arrangement might be problematic when two or more components try to modify the content of a shared value object simultaneously. This however would never occur in a valid XML-base model description such as CellML because from the set of all variables that are pointing to a given Value object only one is an output variable (this is a rule enforced by CellML specification). Therefore only one variable is capable of modifying the Value object; others are merely observers of its numerical content. The task of ensuring that this configuration holds true for all components of the model is a responsibility of the XML-based markup language specification.

Figure 39 shows examples of legal and illegal connection configurations as per CellML specification. Note that a single output variable can be linked to as many inputs as you like, however two or more output variables cannot be linked to a single input channel as they would otherwise override each other and result in an incorrect behavior.

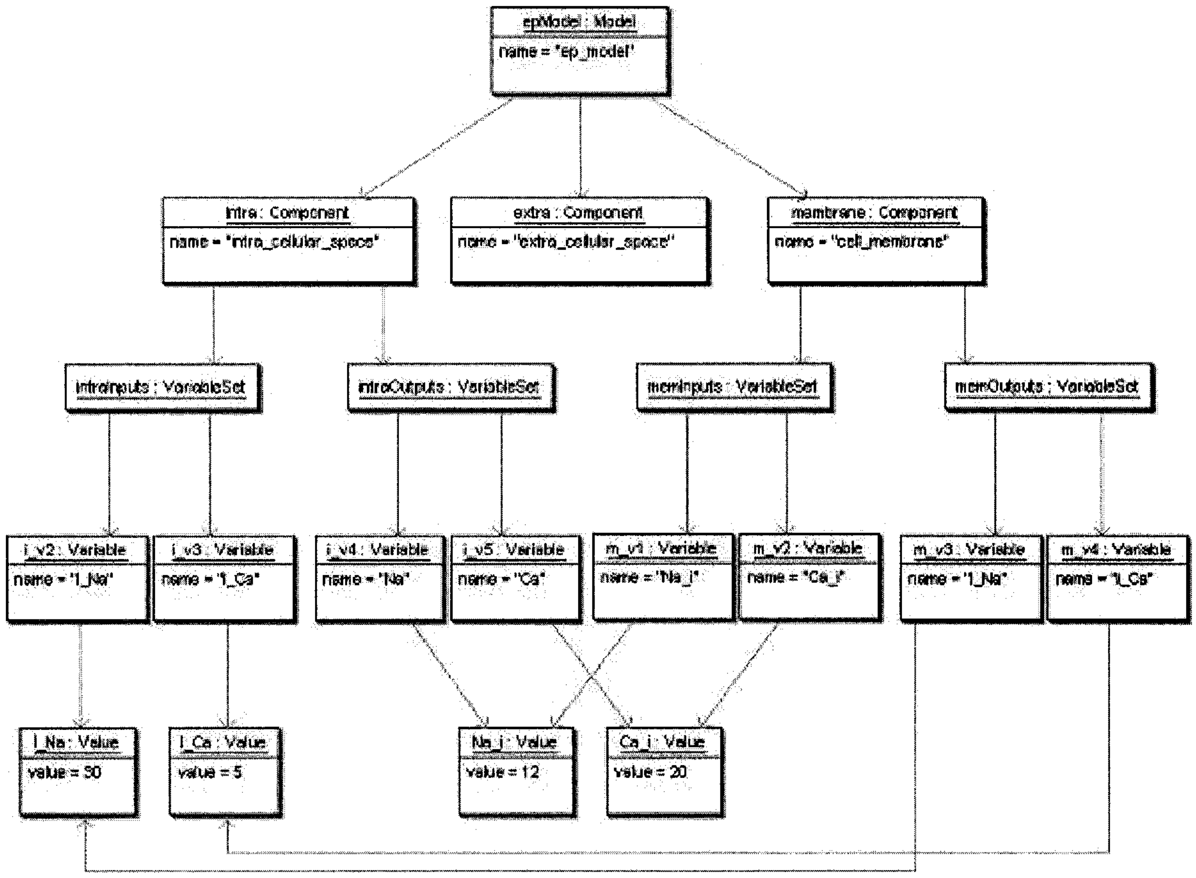


Figure 38 – The UML object diagram of instances that are automatically constructed when the basic_ep_model CellML model is parsed (note: some instances are omitted due to lack of space).

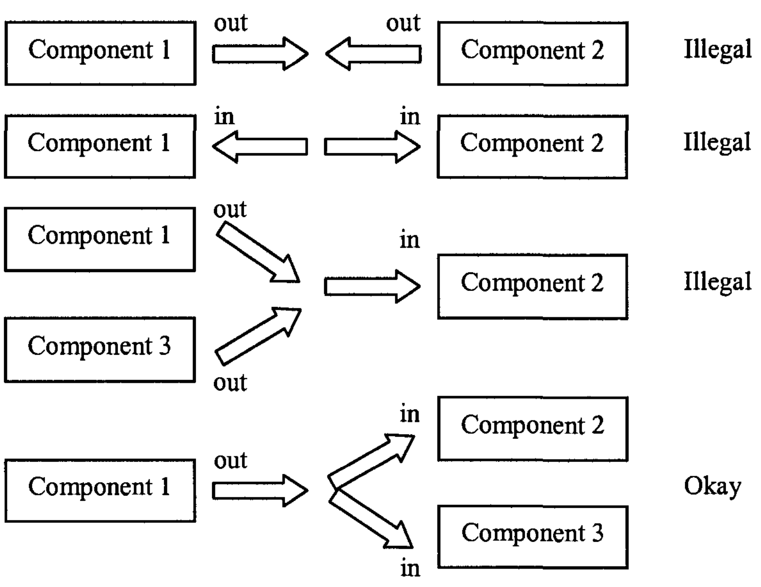


Figure 39 – Examples of legal and illegal connectivity configurations among input-output variables.

4.2.3 Alternative Connection Linking

There are two alternatives to the above design. Using the observer-observable design pattern [29], one can implement all of the components as observable entities that fire events when the values of one or more output variables are changed due to new incoming inputs or elapse of time. Those components that are interested in the outputs of a given observable component can register themselves in the observers list of that component and begin receiving update events. When such events are received by the listening component, it often uses the new input values to recalculate and update its own outputs for which it is obligated to fire update events if there is a sensible numerical change in the output amount.

Figure 40 demonstrates the observer-observable design for our CellML model. The cascaded events shooting starts with the simulation engine as it notifies the Intra component (intra_cellular_space) of change in time. The Intra component would then recalculate Na_i based on the current time and fire an event to notify the membrane component of this. Membrane would then use the new Na_i along with the possibly outdated Na_e value to recalculate I_Na. Update events will then be fired and subsequently received by Intra and Extra components.

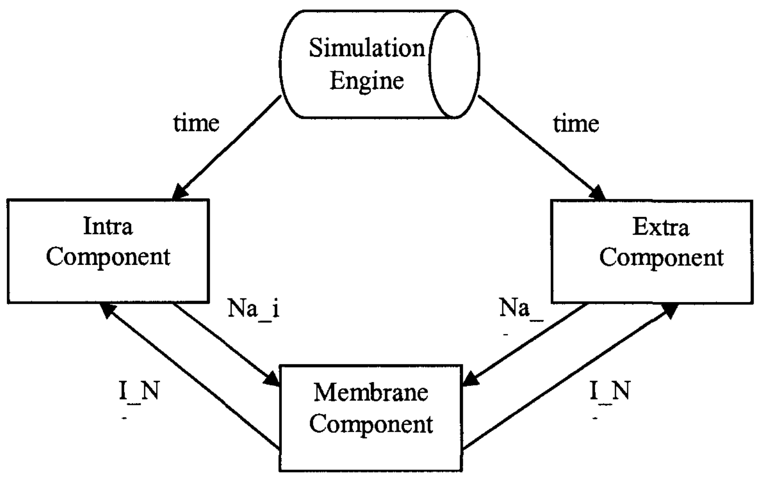


Figure 40 – Event driven simulation system using observer-observable design pattern.

An obvious flaw with such event driven system is that the observer-observable pattern uses synchronized event notification. That is, once a component fires an event it will not get a hold of the CPU until all observers of that event process it. Since firing an event typically has a cascading effect there is a good chance that the control will never return and the program gets stuck in a loop until it runs out of memory. Therefore, if this design is to be implemented, a special mechanism must be used to avoid the infinite cascading problem.

Another alternative is to employ even-driven design within a multi-threaded system [31]. In this case, each component is implemented as a thread that *waits* until it is *notified* that one of its input variables has changed. Upon this notification, the thread becomes activated and the values of the output variables are recomputed. The component would then issue a notification to point out this change of its output, after which those thread waiting for such notification wake up and proceed with their own computations. This alternative is especially attractive since today's multi-core processors can take full advantage of such multi-threaded systems for maximizing the performance. AVRA can be implemented to wrap each simulation model in a thread or create a thread for every component of every simulation model. Since there are over-heads associated with threads the developers must be careful not to overwhelm the system with too many simultaneous running threads.

Table 4 compares the pros and cons of the three alternatives.

	Shared Value Object	Observer-Observable	Multi-Threaded
Pros	-Fast -Avoids unnecessary computations: The output of all components is calculated only once per simulation cycle.	-Avoids sharing of value objects, allowing each component to maintain a local copy of their input/output values. -In certain models, avoid unnecessary computations: The output of any component is only calculated when one of its input values has changed.	-Avoids sharing of value objects, allowing each component to maintain a local copy of their input/output values. -In certain models, avoid unnecessary computations: The output of any component is only calculated when one of its input values has changed. -Easily avoids cascading problem with Observer-Observable alternative as firing event occur non-synchronously. -Added efficiency when running in a multi-core or multi-CPU systems.
Cons	-Extra care must be given to ensure that only output variable modify the content of value objects.	-Cascading effect often results in infinite loops. Special mechanism must be implemented to avoid it. -Unnecessary computation may result in slower simulation overall as some components may recalculate their outputs several times per simulation cycle (each time one of inputs are updated.)	-In a simulation with hundreds of components, the overhead of switching between threads may be too much, resulting in a slower simulation overall.

Table 3 – The pros and cons of three approaches toward connecting math components

4.3 Control Flow in Simulation Models

In AVRA the simulation models are created by the model loaders which themselves are created by the simulation controller whenever the user loads a new model file into the system. This interaction is demonstrated in the sequence diagram of Figure 41. When the simulation controller receives a new model file it checks to see if a valid loader is available for that file. In the sequence diagram, the file being loaded is an Excel sheet which contains a table that represents the simulation data. If an Excel loader is available, the simulation controller will create an instance of it, otherwise an exception is thrown.

A model is created by simply passing the model file to the *load* method of the corresponding model loader. In this case, our excelModelLoader instance creates instances of DataModel and TableModel and subsequently parses the content of the file extracting the relevant data and passing them to the dataTable. Once the model object is ready, it is returned back to the simulation controller at which point it will be assigned to a simulation instance. After the simulation is started, it periodically calls the *step* method of its models include our dataModel. As it was mentioned in section 4.1 the time received by the step method may not correspond to an exact match in the data table; therefore a call to the interpolate method is invoked to compute a linearly interpolated record.

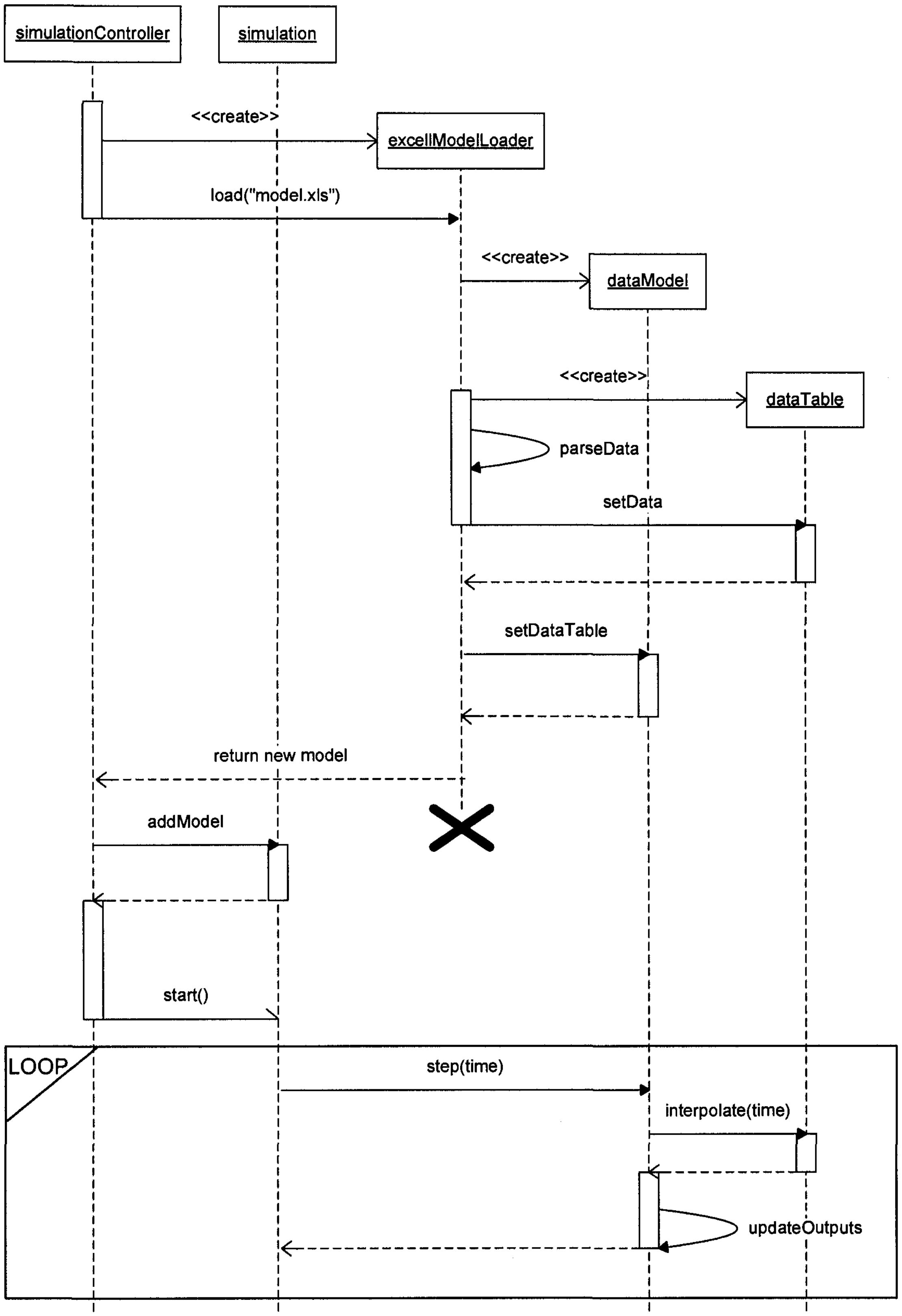


Figure 41 – Sequence diagram of DataModel

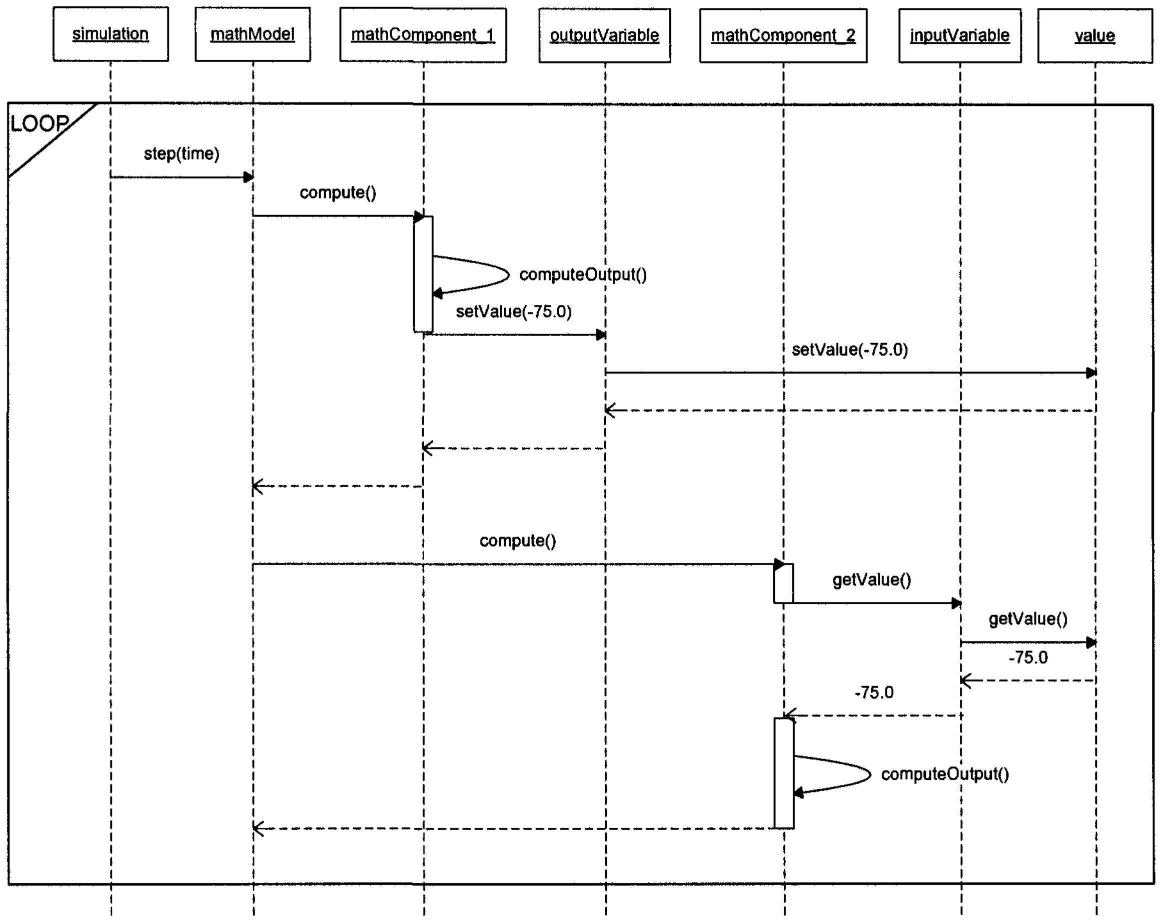


Figure 42 – Sequence diagram of MathModel

Loading a mathematical model is done similar to data model or any other model; it involves instantiating a proper loader and adding the model to the simulation when the loader returns it. However computing the outputs of the mathematical models during the simulation is more complex and involves a system of interconnected math components. As it was discussed in section 4.2.2, in AVRA the task of connecting the output of a component to the input of another is achieved by sharing the Value objects. Figure 42 demonstrates a simple model consisting of two math components. The first component merely receives the time as its input and generates a single output (-75.0 in this example) stored in outputVariable. The second math component receives the inputVariable in addition to the time as its inputs. Earlier at the time of model construction, AVRA arranged it so that the outputVariable of mathComponent_1 and the inputVariable of mathComponent_2 point to a same value object. Therefore when mathComponent_2 tries to retrieve the current value of its input it would always get the most recently computed value as set by mathComponent_2. After retrieving its inputs, mathComponent_2 computes its output values and subsequently updates its output variables which may or may not be connected to inputs of other components (not shown in the figure).

CHAPTER 5 - SIMULATION VIEWS

The view component manages all those entities that contribute to the visualization of the simulation model. In general, the view is continuously observing the outputs of the model and adjusting its graphical visualization accordingly. As a result, a change in the numerical output of a model would result in changes in the 3D scene. Recall from the previous chapter that Model is an abstract class with data or mathematical models as its possible implementations. The view only uses the generic interface of the models as defined in the Model class, therefore as long as it is connected to a Model instance it does not care which concrete implementation it belongs to. From a high-level perspective, a view component simply receives numerical outputs of the model that it represents and translates those numerical values into some visual animations or interactions.

5.1 View Components

Figure 43 demonstrates the view and the components that it depend on. Just as with the models, a view is created by its loader which receives the view description from a file. The view loader parses the view description which is in an XML-based format named MVML. MVML will be discussed in length in chapter 8. Unlike the model loaders, the view loader is a concrete implementation which is specifically designed for loading MVML files; therefore in most cases there is no need for it to be extended to create customized loaders as this would generally convey changes in the underlying core structure of the MVML and the view system. Once the view is constructed and initialized with specified parameters it is added to the simulation that contains its target model.

A view consists of three main types of elements: graphics, connectors and formulas. Since AVRA is designed to generate virtual environments that represent the simulation models, its view consists of numerous 3D models that are loaded and embedded into the virtual scene. These 3D models will move, deform, animate and interact based on the numerical output of the models that they are connected to. The *graphics* objects wrap these 3D models and provide an interface for recovering and manipulating them. The *connectors* are used for establishing an automated data connection between the output of a simulation model and the input of the animation or visualization module that manipulates the graphics. The *formulas* are mathematical entities that are used for scaling the output values of the simulations models so that they are of acceptable range for the visualization modules. Both connectors and formulas will be discussed in detailed in chapter 7.

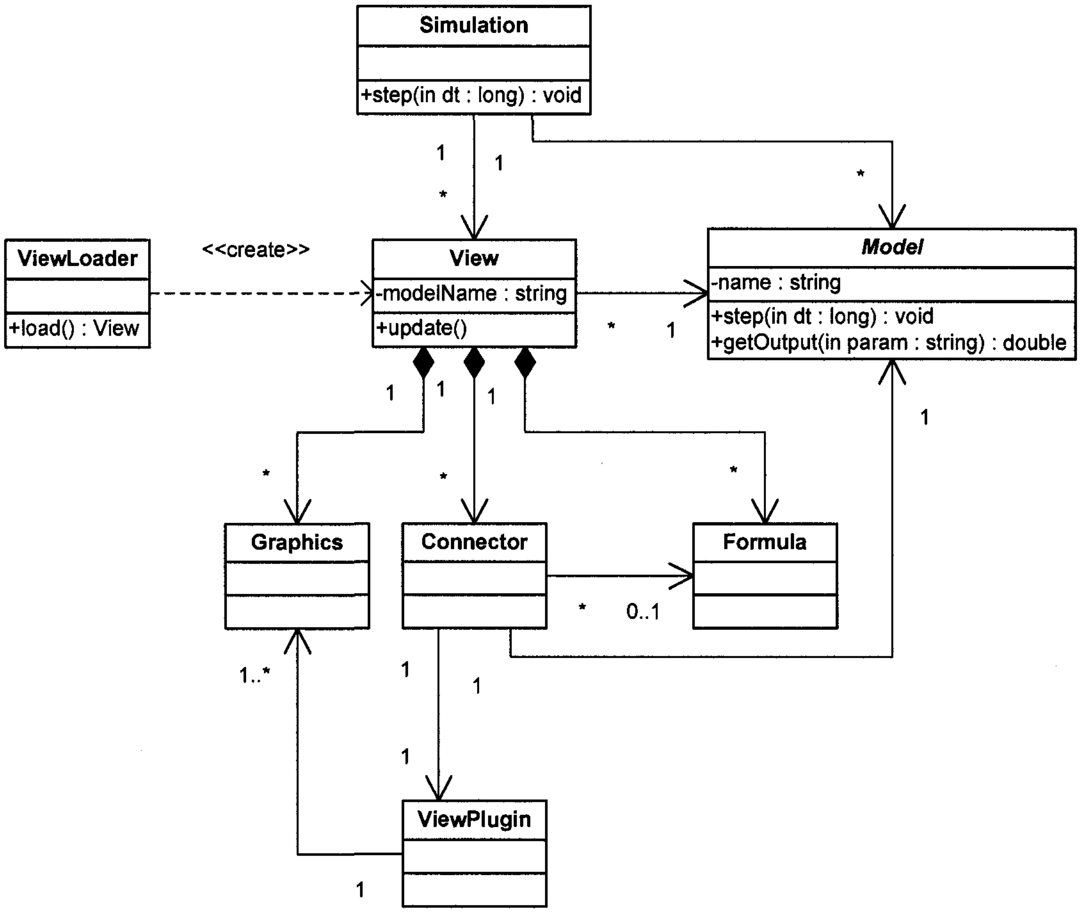


Figure 43 – Class diagram of the view components

5.2 3D Graphical Models

AVRA implementation supports various 3D model formats such as VRML, X3D, 3DS, etc. In addition, AVRA architecture makes it easy to add support for other 3D formats by registering third-party loaders. The following code registers a new VRMLLoader for those 3D models whose filename has a .wrl or .vrml extension.

```
ViewLoader.register3DModelLoader(VRMLLoader.class, "wrl,vrml");
```

In the case above, the code would effectively disable the built-in VRML loader to allow for the third-party loader to load the VRML files instead. In general the latter calls to *register3DModelLoader* always replace the earlier loaders of a same filename extension.

Sometimes the 3D models are simple and are used by the view as a whole. However often these 3D models are complex and may consist of tens or hundreds of smaller parts that are to be animated independently base on different outputs of the simulation model. For example the simulation model in Figure 44 has several output parameters that define a walking robot by generating values for its joint angles. Therefore when describing the view of this model we must connect these outputs to their corresponding graphical parts within the 3D model of the robot. For example in the figure the output out2 is connected to the left_arm_group; that is if the value of out2 is 90 the entire left arm which includes the lower arm and the hand will rotate 90 degrees around its joint. It is therefore crucial that the 3D models have unique names for all graphical parts of interest and that the view descriptions correctly identify those parts using their names.

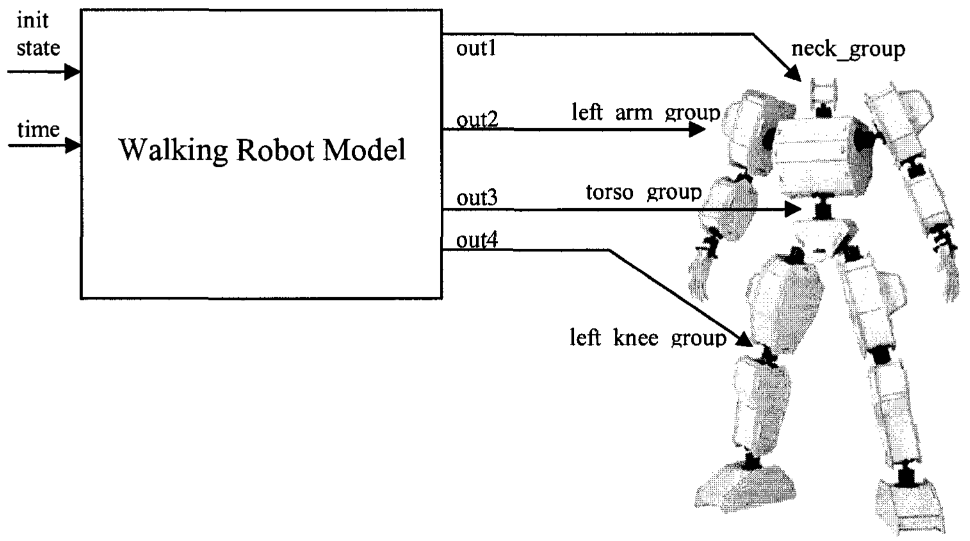
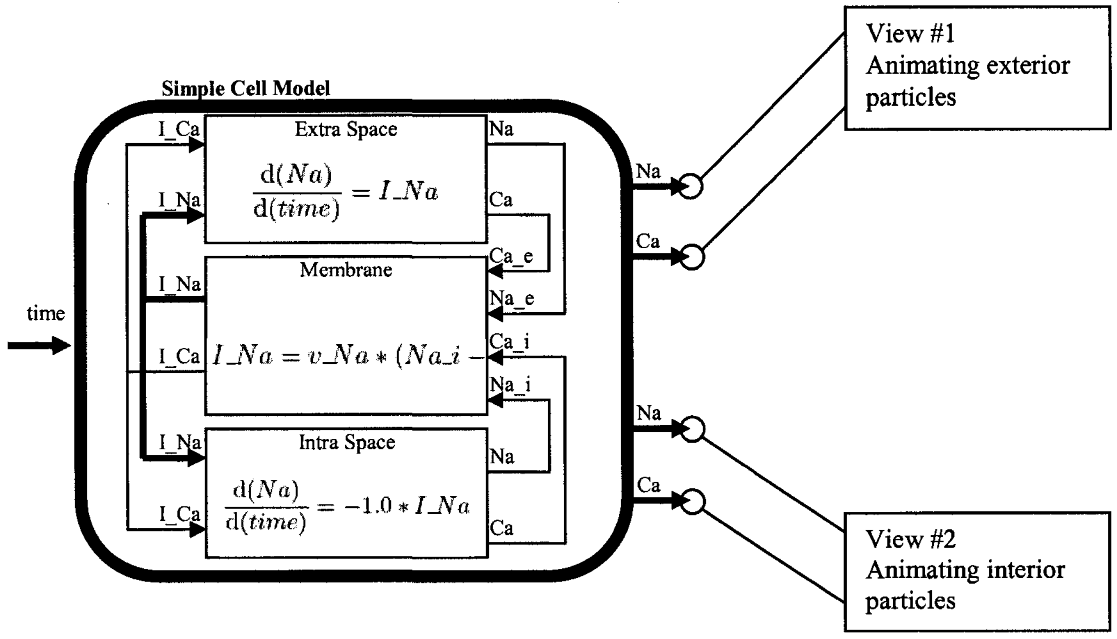


Figure 44 – The connections between the outputs of a simulation model and the named parts of a single 3D model

5.3 Model-View Associations

Each instance of the view is associated with a single model (the model that it will visualize). A view cannot represent multiple models; that is for *n* models there must exist at least *n* unique views in order to visualize all of the models. It is however possible that multiple views point to the same model. This is particularly useful when several view components represent the various aspects of a complex model. For example Figure 45 and Figure 46 demonstrate four different view components representing the same mathematical model of a simple cell but visualizing different aspects of the model. In particular, view #1 and #2 use the outputs of the intra and extra spaces of a neuron cell in order to visualize the local state of those champers at any give time. This is while view #3 and #4 use a different combination of same outputs to animate the flow of Na and Ca particles in and out of those chambers through the membrane. Both views have practical applications and are deemed valuable to scientists who are observing the different aspects of a simulation, yet both visualize a same model with same numerical outputs.



View #1
Animating exterior particles
Simple Cell Model
Extra Space
$\frac{d(Na)}{d(time)} = I\_Na$
Membrane
$I\_Na = v\_Na * (Na\_i -$
Intra Space
$\frac{d(Na)}{d(time)} = -1.0 * I\_Na$
I_Ca
I_Na
Na
Ca
Ca_e
Na_e
Ca_i
Na_i
time
View #2
Animating interior particles

Figure 45 – Two view configurations that visualize the local state of the extra and intra spaces.

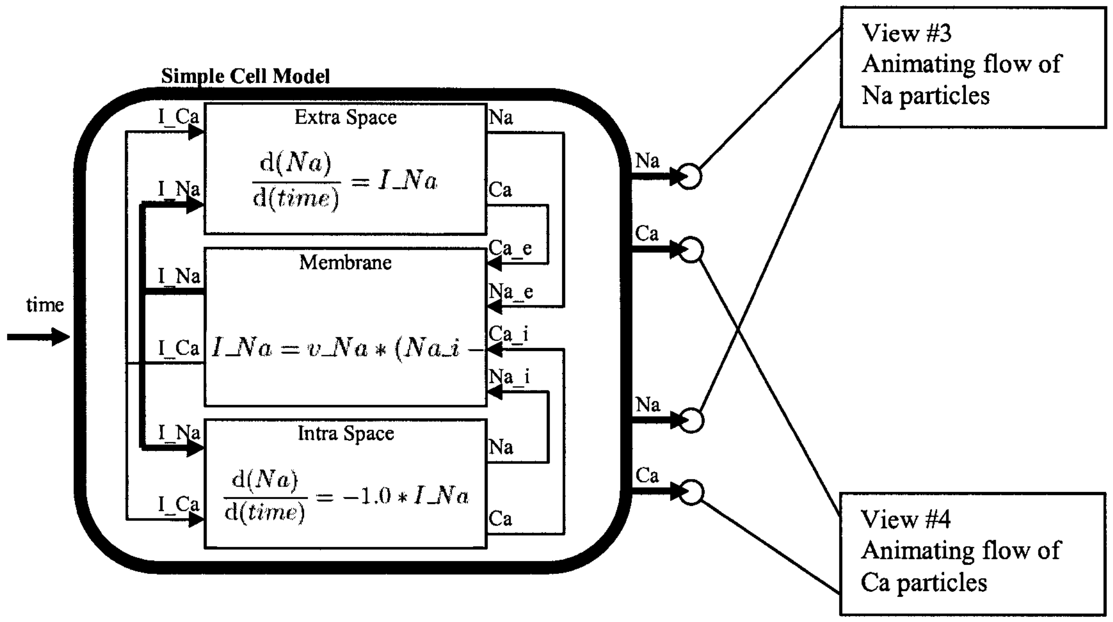


Figure 46 – Two view configurations that visualize the inter-space dynamics across the membrane.

This section presented a high-level overview of the mapping of view components to their target models. Due to decoupling and reusability issues view components are not directly connected to the model outputs. Instead this is achieved in AVRA through view plugins and connectors. The architecture of view plugins and connectors along with their various benefits are discussed in debt in chapter 6 and chapter 7.

CHAPTER 6 - VIEW PLUGINS

In order to accurately visualize the output of a simulation model, the animation and interaction effects must be updated continuously as new data arrive from the model. As a result, the use of static pre-generated animation is not adequate for this type of problems. AVRA introduces reusable view plugins to address this issue.

6.1 Behavior of View Plugins

View plugins are compiled entities that take advantage of the functionalities of the embedded physics engine to produce highly customizable animations and interactions that truly reflect the state of the simulation models in real-time. In essence, a given view plugin programmatically generates a dynamic animation based on a pre-determined set of data that are received through its well-defined interface. For example, the interface of a view plugin that animates the heart-beats may provide methods for setting the rate and magnitude of the heart-beats.

View plugins may be defined and implemented to be very specific or very generic. For example a view plugin that is used in visualizing the heart-beats may be an authentic heart-beat animator that accepts heart-specific parameters or a generic deformation module that simply deforms any set of 3D geometries based on some generic parameters. The choice of how specific or generic the implementation of a view plugin is depends on the requirements of the simulation that needs it. Naturally the more generic view plugins have the advantage of being reusable. Figure 48 shows two different plugins that can be used for visualizing the heart-beats. The HeartBeatAnimator simplifies the task as it only requires the rate attribute whereas the DeformationLaw requires additional attributes and instances. More on this will be discussed in section 6.3.

The animation that is generated by a given view plugin may be anywhere from predictable to randomly variable. It is unlikely however that the generated animation is completely random; instead it is usually random within some pre-defined thresholds. For example in the simulation of a biological organ that generates some ions at a particular rate it is generally preferred that the new ions be distributed randomly but within the boundaries of the organ.

6.2 View Plugins as Extensions of Physics Plugins

As it was discussed in section 3.2, the xPheve physics engine is used in this framework. The integrated physics engine serves two purposes: it generates physically realistic behaviors and more importantly, it is the basis of the view plugins in AVRA. The view plugins are in fact extensions of the *law plugins* as specified in xPheve. This architecture allows the physics engine to take control of the management of the 3D visualization hence avoiding possible conflicts between the physics engine and the simulation engine while conveniently reusing the visualization services that are already offered by xPheve. The separation of the visualization module (physics engine) and the simulation engine makes sense because the visualization updates generally occur at a slower rate (once per frame) than the numerical computations which may require several iterations per frame depending on the size of the simulation steps.

Figure 47 shows a class diagram that demonstrates how view plugins are developed as law plugins in AVRA. As it was discussed earlier, each view component is typically associated with one or more view plugins through connectors. In the actual implementation though, the view components are associated with the super class PhysicalLaw whose subclasses may or may not be implementation of view plugins. This intentional abstraction allows view to access not only view plugins that are specifically developed for AVRA, but also the more generic physical laws that are available within xPheve libraries such as CollisionLaw, DeformationLaw, NewtonsLaw and more.

In order to define a custom view plugin that addresses the specific needs of an AVRA application one needs to extend the PhysicalLaw and implement the enforce method. During the run-time, the enforce methods of law and view plugins are periodically called by the physics engine. For view plugins, the enforce method should contain codes that perform some animation or visualization based on the inputs received from the simulation model. The examples shown in Figure 47 show ColorInterpolator, GateAnimator and ParticleAnimator which are used in our case study prototypes as discussed in section 9.2.

When a law or view plugin is constructed it must be added to the physics engine using the addLaw method. This assure that once the physics engine is started it periodically (usually once per frame) activates the view plugin giving it a chance to update its visualization.

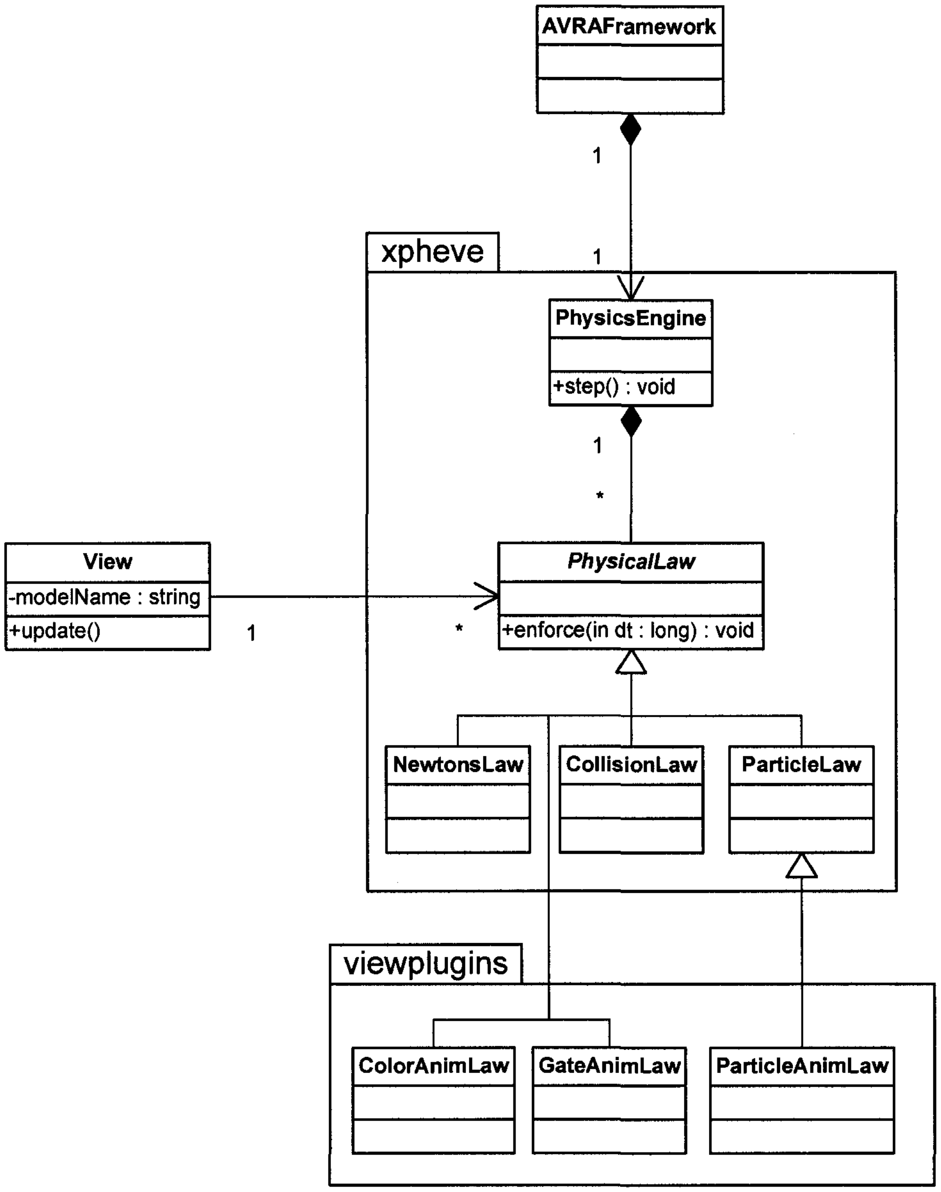


Figure 47 – Class diagram of view plugins as extensions of physical laws

6.3 Interfaces and Internal Structure

From an implementation point of view, a view plugin must abide by three simple rules:

- it must inherit from the physics engine's PhysicalLaw;
- it must have a default constructor;
- and it must have a public interface for receiving the necessary attributes.

Internally a view plugin typically has implementation codes for generating the 3D visualization based on the values of its attributes.

The reason that we need a default constructor for each view plugin is that when the view loader is constructing an instance of the view plugin it cannot understand the conceptual meaning of the constructor parameters. For example, a view plugin that has *age* and *id* as its two integer attributes and receives them as parameters of its constructor does not leave any information for the view loader regarding which attributes is passed first and which is passed second. Therefore in AVRA attributes can only be set through the accessor methods or pubic properties which must abide by the standard naming conventions.

Attributes are implemented through properties in some programming languages and through accessor methods (setter and getters) in others. As our implementation of AVRA is in Java we define and implement accessor methods for each attributes that must be accessible by its users. For example in Figure 48 the HeartBeatAnimator has setter and getter methods for both geometry and rate. It is important to follow the standard property format (such as JavaBeans' naming convention for setters and getters if a Java implementation of AVRA is used) because both view loader and connectors depend on it (more discussions in section 6.5).

As it was discussed in section 6.1, Figure 48 shows two alternatives for visualizing heart-beats. The DeformationLaw is more generic and as such requires extra attributes namely bounds and magnitude. The bounds attribute specifies the region for which deformation must take place and the magnitude reflects the scale of deformation. Additionally, using the more generic DeformationLaw requires four instances that contain four different bounds, one for each chamber of heart.

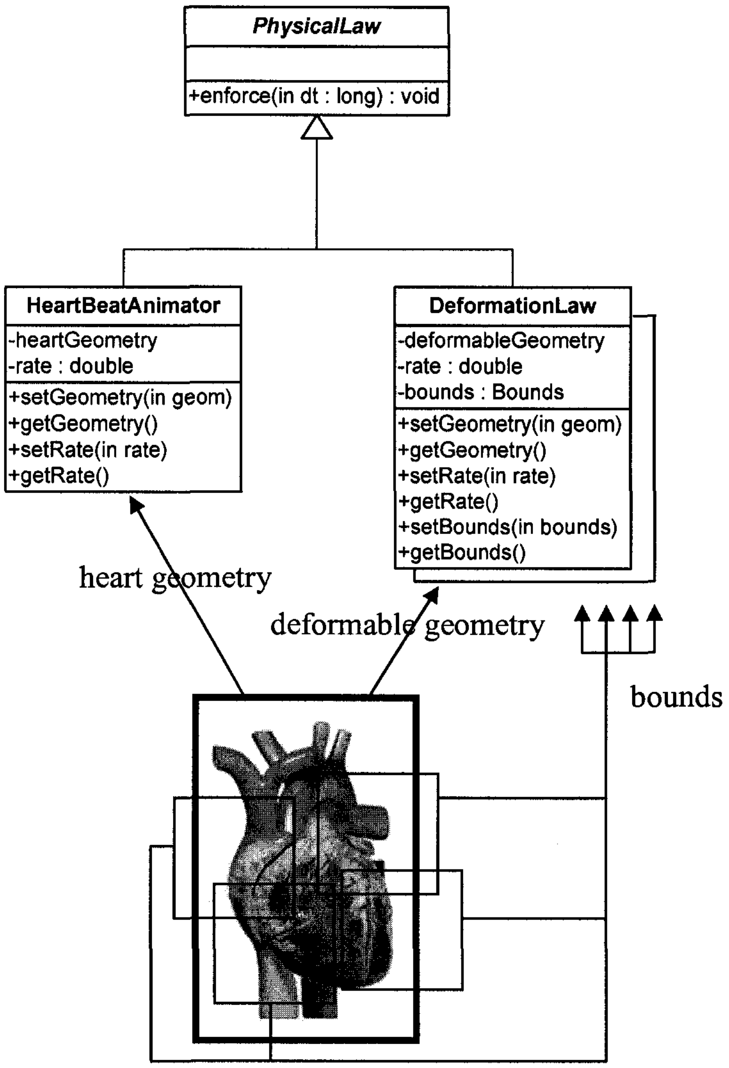


Figure 48 – Examples of specific view plugin and generic law plugin when animating heart-beats

6.4 Construction and Initialization

In AVRA, view plugins are dynamically constructed by the view loader as it parses the view description from MVML files (see chapter 8). In MVML, the view plugins are simply identified by their package names and class names. For example a view plugin with class name of HeartBeatAnimator and package name of discover.avra.views is identified by "discover.avra.views.HeartBeatAnimator". It is therefore important that no two view plugins in an AVRA application have same package and class name.

A view plugin typically goes through these steps:

1. It is dynamically constructed by the view loader and attached to an instance of View.
2. It receives values for its attributes.
3. It is connected to one or more outputs of one or more simulation models.
4. It is integrated with the physics engine.
5. It is initialized during which the visualization (i.e. animation) components are constructed and initialized.
6. It is periodically updated during which some parts of the visualization components are updated or rebuilt based on the new output(s) received from the simulation model.
7. It is destroyed when the higher level application loads a new view configuration (new MVML description) or the application is terminated.

Figure 49 demonstrates the activities within the view loader as it constructs a new view plugin and assigns values to its attributes. Upon receiving attributes the value description is parsed in order to identify what type of value is being assigned to the target attribute. There are three possible types of values: *primitive values*, *graphics parts* and *model outputs*.

Primitive values are the simplest form of values; they convey constant values typically in the form of numbers, strings or boolean (true or false). The primitive values are initially parsed as strings then converted to either boolean or numerical value if possible. If the value is a number it is initially constructed with *double* precision. However before passing these double precision values to the setter methods of the view plugin they may be casted to less precise primitive types such as floats or integers depending on the type of parameter that the setter method accepts. Primitive values can also constitute a list that consists of any combination of numbers, booleans and strings. This is necessary for those setter methods that receive multiple parameters.

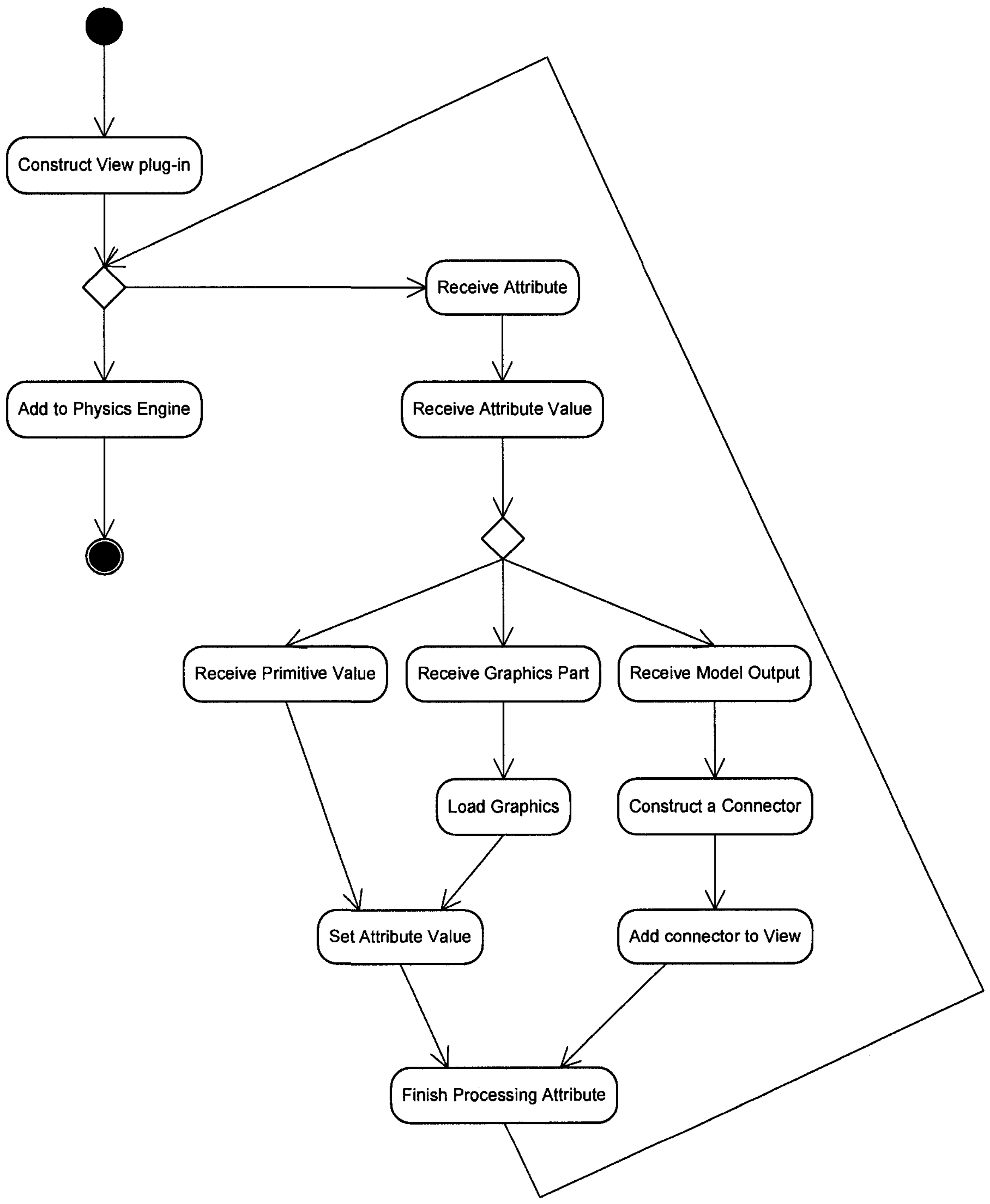


Figure 49 – Activity diagram of View Loader when constructing a View Plugin

Each view component is typically associated with one or more *Graphics* elements. These graphics that are typically loaded before instantiating the view plugin may be passed to setter methods of the view plugin in their entirely or partially. For example, in the case of a more complicated graphics model such as the robot shown in Figure 44, a given view plugin is typically responsible for animating specific parts of the model such rotating the left knee based on the output of the walking simulation model. If a part name is available in the view description of the simulation then the view loader extracts the scene graph node [and all its children] that has matching name. This scene graph branch is then sent to the view plugin as parameter of the setter method that represents the target attribute.

The third type of attribute value is *model output* which represents continuous flow of numerical values from a specific output of a specific simulation model. In the view description, the model, its output and, if applicable, the math component that generates that output are identified by their names. The view loader extracts the target output variable and connects it to view plugin using a new instance of connector. Connectors are responsible for continuously feeding the view plugin with output data from models. The details of how this is done are discussed in the next chapter.

Once a primitive value or a graphics part is extracted the corresponding setter method is invoked by the view loader in order to initialize the attributes of the view plugin. This step is not necessary for model output attributes as those attributes will be initialized and updated later by the connectors when output data from models become available. Instead the view loader adds the constructed connectors to the view component which is responsible for managing connectors as discussed in the next chapter.

6.5 Reflection

The procedure for dynamic construction of view plugins and assigning values to their attributes is achieved in AVRA through reflection. *Reflection* is the process by which a computer program is able to observe and control its own structure and behavior at run-time [13]. In AVRA two types of components use reflection, the view loaders and the connectors. View loaders use reflection for constructing view plugin dynamically and assigning attributes to them. The reason why view plugins need to be constructed dynamically is that AVRA does not know ahead of time which classes of view plugin are available. These plugins may be attached to a given AVRA project as external libraries hence the need for view loaders to query available libraries and identify available view plugins.

```
PhysicalLaw viewPlugin;
try {
  // find and load plugin
  Class c = Class.forName("avra.viewplugins.HeartBeatAnimator");

  // construct a new instance of the plugin
  Object plugin = c.newInstance();

  // check to see if it is a valid view plugin (extends PhysicalLaw)
  if (plugin instanceof xPheve.PhysicalLaw)
    viewPlugin = (PhysicalLaw) plugin;
  else
    throw new RuntimeException("Invalid View Plugin.");

} catch (ClassNotFoundException e) {
    throw new RuntimeException("Could not find View Plugin.");
} catch (InstantiationException e) {
    throw new RuntimeException("Could not instantiate View Plugin.");
} catch (IllegalAccessException e) {
    throw new RuntimeException("Could not access the default
       constructor of the View Plugin.");
}
```

Figure 50 – Example of reflection in Java for loading and instantiating view plugins

Once a requested view plugin is identified by the view loader, it is instantiated using its default constructor. An example of how this is done in AVRA is shown in Figure 50. In Java reflection revolves around a special class named Class. The static forName method of Class is used for loading any given library, driver or class. In the example, "avra.viewplugin.HeartBeatAnimator" which is the full package address of the HeartBeatAnimator is passed to forName method. This will cause the Java Virtual Machine (JVM) to search for and load this class. If successful, the loaded class is returned and stored in variable c. ClassNotFoundException is thrown if JVM is unable to find the class.

The newInstance method can be called on any concrete Class instance in order to dynamically instantiate the loaded class by calling its default constructor. If the class cannot be instantiated because it is an abstract class or an interface or it does not have a default constructor, then an InstantiationException is thrown. If the class is concrete and it has a default constructor, but it is not accessible (i.e. the constructor is defined private or protected) then an IllegalAccessException is thrown. If the instantiation is successful an instance of the plugin is returned in the form of the super-class Object. At this stage the validity of the view plugin is verified by making sure that it is a subclass of the xPheve's PhysicalLaw. If passed, then view loader type casts the plugin to PhysicalLaw and returns it as a verified view plugin.

```java
// Parameters extracted from the MVML description
String attName = "Rate";
String attValue = "1.25";

// Search for the corresponding setter method
String methodName = "set" + attName;
Method[] methods = viewPlugin.getClass().getMethods();
Method method;
for (int i=0; i<methods.length; i++) {
      if (methods[i].getName().equalsIgnoreCase(methodName)) {
            method = methods[i];
            break;
      }
}

// Prepare the method parameters
ArrayList list = new ArrayList();
list.add(Double.valueOf(attValue));
Object[] params = list.toArray();

// Execute the method
method.invoke(viewPlugin, params);
```

Figure 51 – Example of reflection in Java for setting the value for an attribute

The next major step is initializing the view plugin with values for its attributes. As it will be discussed in chapter 8, these values are embedded in the view description of MVML which are parsed and extracted by the view loader. Figure 51 assumes that attribute *Rate* and its corresponding value of 1.25 are already extracted from the MVML description by the view loader. The example shows how reflection is used in AVRA to dynamically set the attribute of the view plugin by generating call to viewPlugin.setRate(1.25) where viewPlugin is the sample plugin that was instantiated in Figure 50.

Extracting the method name in this example assumes Java conventions and is accomplished by simply inserting "set" before the attribute name; in this case setRate is the result. The same Class that was used previously to load and instantiate our view plugin is used in this example for retrieving the list of methods available in viewPlugin object. The getMethods() is used for this task; it returns an array of methods. A simple for loop is used for comparing the names of the methods in search of setRate. In our implementation the contents of MVML are not case sensitive hence the eqaulsIgnoreCase method is used for comparing method names. When the method is found it is assigned to the *method* variable and the loop ends.

The next step involves preparing an array that contains values for the parameters that are passed to the target method. Even if a method accepts just one parameter, we still need to pass it as an array of one element as it is the case in this example. The example assumes that attValue must be passed as a *double*. When the *params* array is ready, we can execute the setter method by calling its invoke method. As its first parameter the invoke method receives the object whose method we are about to execute and as its second parameter it receives the params array. The call made at this point is equivalent to hard coding a call to viewPlugin.setRate(1.25).

The view loader must also consider possibilities of setter methods with multiple parameters and multiple setter methods with same names but different parameters. Figure 52 demonstrates how these cases are handled by the view loader in AVRA. Initially the primitive values are extracted from the XML description as strings and stored in an array of strings. Then the loader queries the methods available in the target view plugin using the reflection capabilities of its Class object.

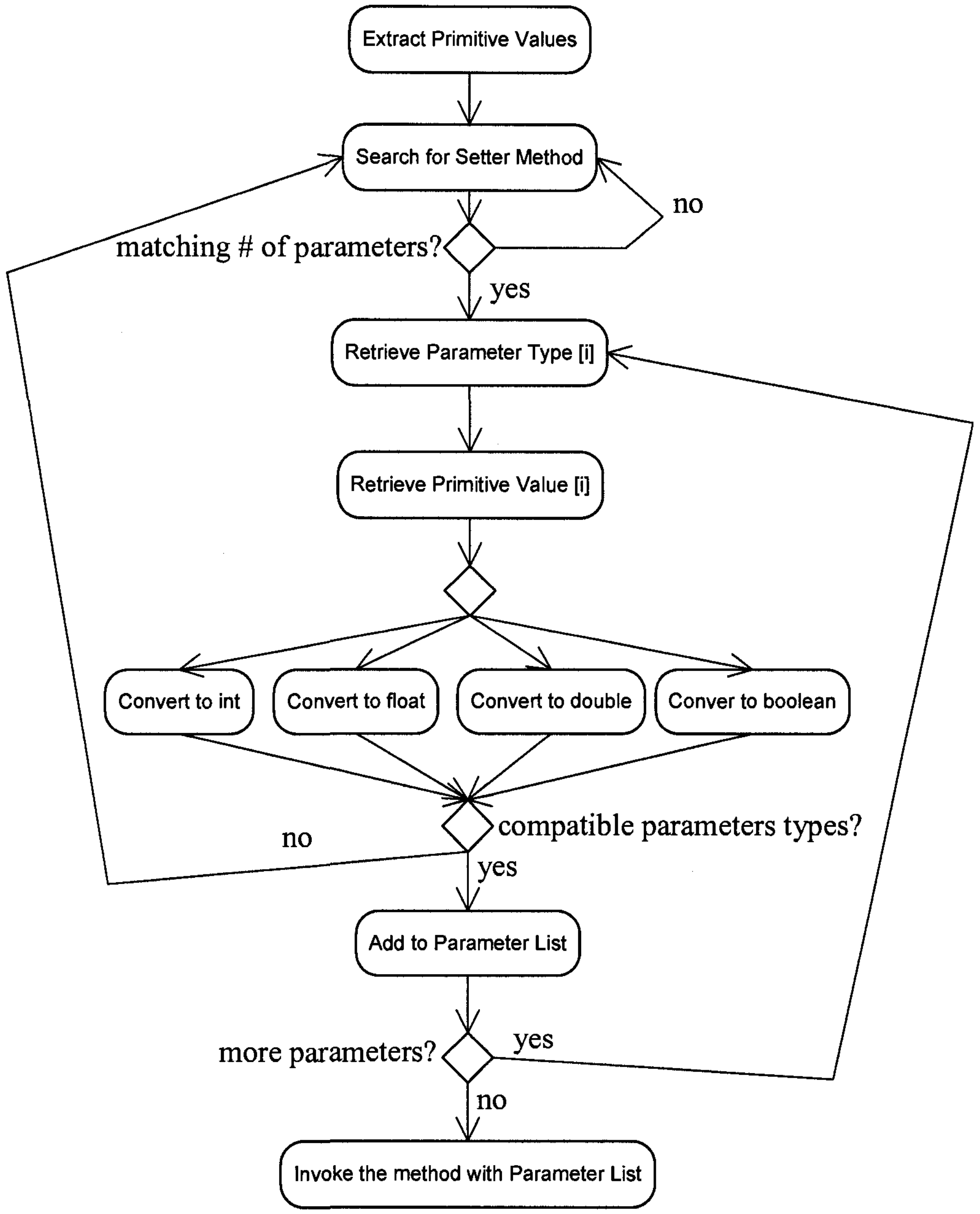


Figure 52 – Activity diagram for invoking methods with multiple parameters

Once a method with set[AttributeName] as its name is found, the view loader queries its list of parameters. If the number of parameters that the method takes is not equal to the number of the primitive values that were extracted from the MVML description, then the control is sent back to top of the loop in search of another method with same name. If the number of parameters does match, then for each parameter the corresponding primitive value is converted from string to what the method accepts. Values may be converted to int, long, float, double or boolean, If the conversion fails then the control is sent back to top of the loop in search of another method; otherwise the converted value is added to the parameter list with is an array of Objects. Upon successful conversion of all parameter values, the method will be invoked and the control is returned.

CHAPTER 7 - MODEL-VIEW CONNECTORS

Up to this point, both model and view components are constructed independently of each other without any sort of connection or awareness. A model-view connector allows the establishment of the much needed connection between model and view components without violating their independence. Connectors are constructed and initialized by the view loader at the time when a view description is being parsed. In particular, when the description of a view plugin is parsed, for each attribute of the view plugin that is dependant on an output of the simulation model, a model-view connector is automatically created. This connector allows the view plugin to continuously receive the current output value of the model to which it is attached.

7.1 Conceptual Connections

The ultimate objective is to connect the outputs of a model to some 3D graphics in such a way that a change in the output of model would result in changes/animations in the 3D graphics. For example, in Figure 53 the outputs of a model are connected to two view plugins that generate particular animations. The model in this example is that of a neuron cell with two outputs: gate_M which represents the state of the gate that controls Sodium flow and i_K which represents the rate of Sodium flow from outside of the cell to inside of it. The output gate_M is connected to a view plugin that controls the 3D model of a cylindrical gate and i_K is connected to a particular generator view plugin that generates and moves 3D models of a Sodium ion. With this configuration, a change in gate_M causes a change in angle □of the virtual gate and the value of i_K is interpreted as the rate at which particles from the top region moves to the bottom region through a virtual opening. As the simulation engine is running the values of gate_M and i_K are continuously recomputed by the model and eventually make their way to the view plugins to which they are connected.

If the model is a DataModel these values are retrieved from a data table and if the model is a MathModel they are calculated at run-time. Regardless of how these outputs are generated they are available to any entity that requires them. In AVRA the view plugins are the entities that are most interested in capturing these values however they cannot access them directly. The reason for this is that view plugins are generic entities that are not aware of the interface nor the existence of the model components. View plugins simply modify, animate or interact with their target 3D graphics in the virtual environments based on their current parameters. The question is which components are responsible for setting those parameters. As it was discussed in section 6.4, the view loader is responsible for initializing the parameters of the view plugins using their setter methods but only those parameters that will not change during the course of the simulation. For those parameters that need to be continuously updated this is achieved through connectors.

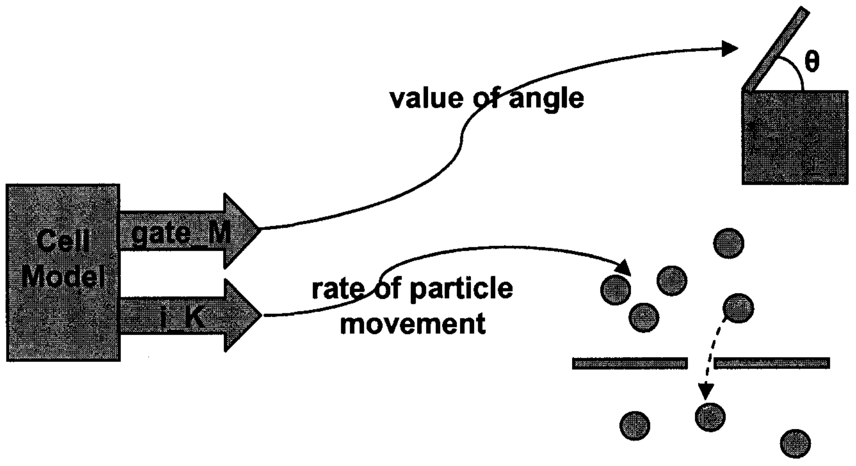


Figure 53 – A conceptual connection between a model and its views.

7.2 Data Flow

Figure 54 shows the conceptual diagram of a connector as it bridges between the model and the view plugin. In particle this figure demonstrates that the value of gate_M is received from the simulation model and then forwarded to the view plugin as an angle causing the view plugin to rotate its target geometry by that angle.

When the view loader realizes that an attribute of a view plugin depends on the output of a model, it creates an instance of a connector. Each connector is associated with four other components: a view plugin, a setter method, an output variable and a function. Each connector instance belongs to one and only one view component. During the simulation, the view periodically calls the update method of all its connectors causing the connectors to invoke calls to the setter method of their target view plugin passing the value of their output variable as the method parameter.

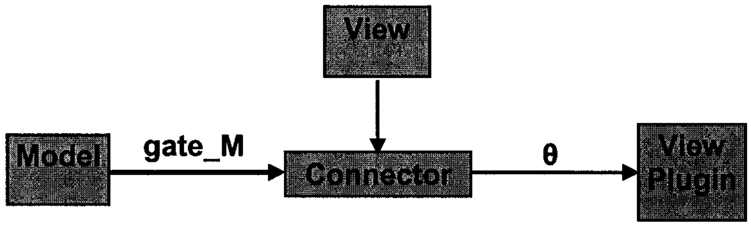


Figure 54 – A connector allows one of the outputs of a model to transmit its data to a view plugin.

7.3 Intercommunication in Simulation and Visualization Threads

Figure 55 demonstrates how a connector is used for retrieving an output value from a model and making it available to its target view plugin. As it was discussed earlier, AVRA is designed to handle numerical simulation and visual updates in two different threads. These threads are represents by loop tags in the sequence diagram. The numerical simulation is controlled by the simulationEngine which causes the model to recompute its outputs and the view to update its connectors. In this example the connector is connected to the value instance which holds the target output of the model. At each update cycle the connector retrieve the current value of this output and sends it to the target viewPlugin as a parameter of the setValue method in this case. This method invocation is done dynamically through use of reflection as will be explained in the next section.

In the visualization thread, the physicsEngine is in control of updating the scene by periodically activating its Law components some of which are the view Plugins that models are connected to. In this example physicsEngine activates the viewPlugin by calling its enforce method causing the viewPlugin to generate/update its animation based on the latest output received from the connector, in this case -75.

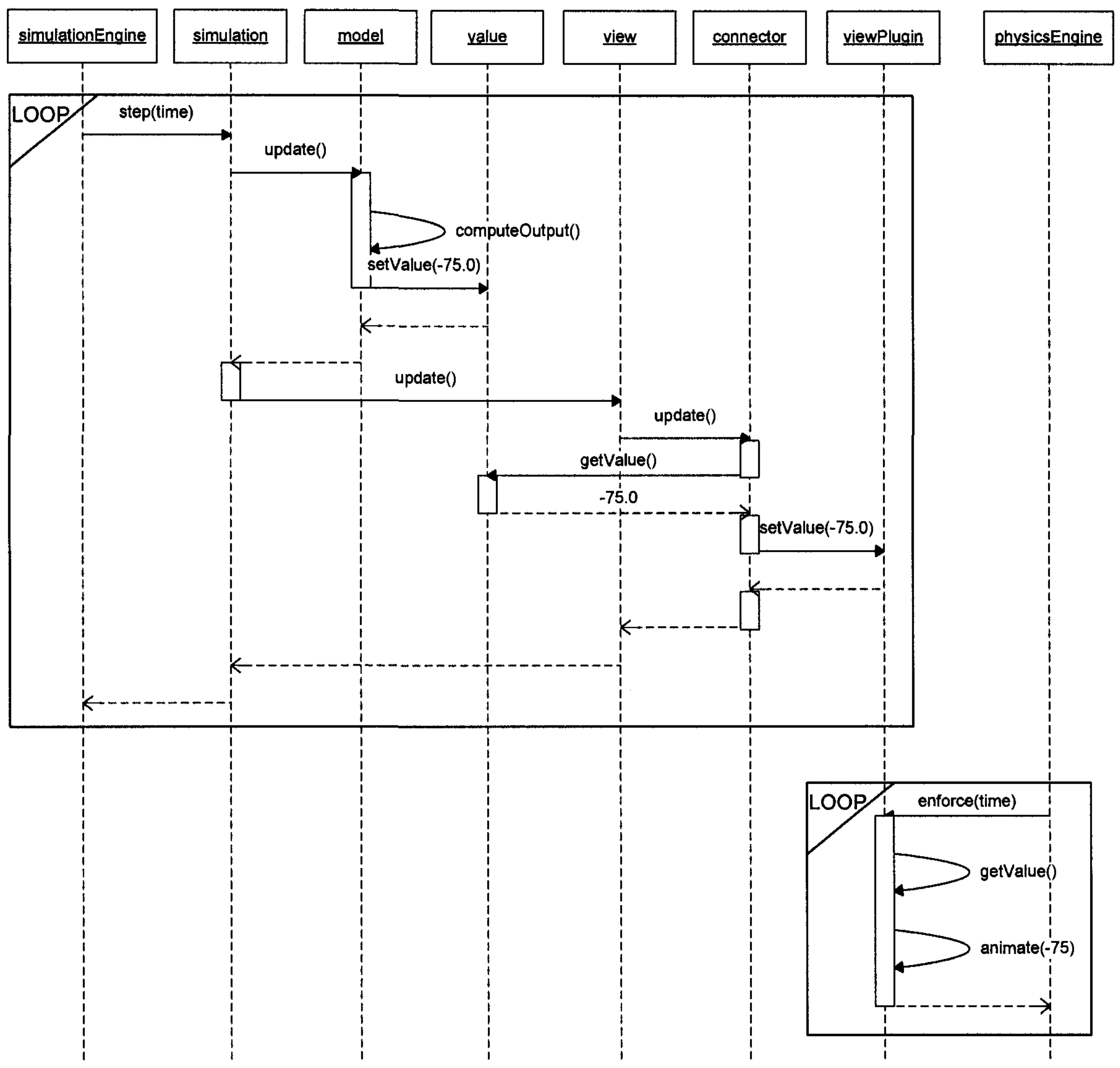


Figure 55 – Sequence diagram of View Plugin as it receives outputs of model and generates an animation accordingly in the next physics engine update.

7.4 Scaling Functions

In the above example the output value of the model is sent to the view plugin as is; however in most cases the value needs to be scaled or otherwise translated to a value that is adequate for the view plugin. For example, the value -75 may represent a low voltage in a neuron cell and the viewPlugin may be a colorInterpolator that visually changes the color of the cell in order to visualize the change in voltage. In neuron cells the voltage typically ranges from -75 to +35. In this case a voltage of -75 should be translated to 0 which +35 should be translated to 1 so there the color interpolator can interpolate between the start and end colors according to this value. This translation is another responsibility of the connectors. Each instance of the Connector is associated with a Function object that is extracted from the MVML description by the view loader and (more on the XML format of Function element is discussed in 8.1.3). Figure 56 summarizes the data flow and its evolution from the time is generated in the Model until the time it arrives at the View Plugin.

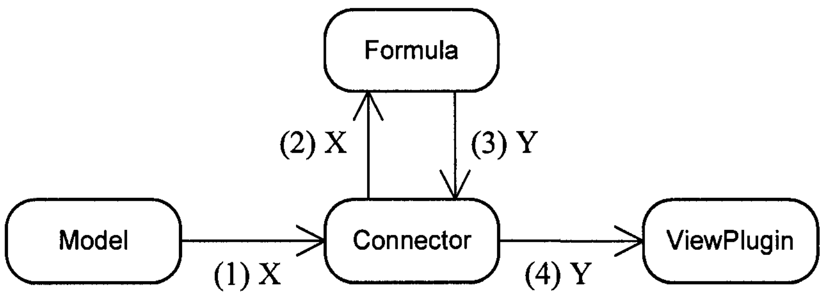


Figure 56 – Data flow from model to view plugin as it is translated from X to Y

7.5 System of Connectors

Each view may contain several connectors however since any instance of view represents only one model these connectors are all connected to the various outputs of a same model. Their outputs on the other hand can be channeled to several instances of view plugins. Each connector is typically associated with a different view plugin although it is possible that two connectors point to different inputs of a same view plugin; this is useful when multiple parameters of a same view plugin depend on multiple outputs of the model as is the case with View Plugin #2 in Figure 57. At each simulation step the view invokes the update method of all its connectors causing new values to be fetched from the model and sent to view plugins.

In AVRA, the connectors are updated by the view at each simulation update cycle. The updating order is as followings: first the update methods of all active simulation models are invoked. Then the update calls to connectors are executed in order to supply the view plugins with the new values.

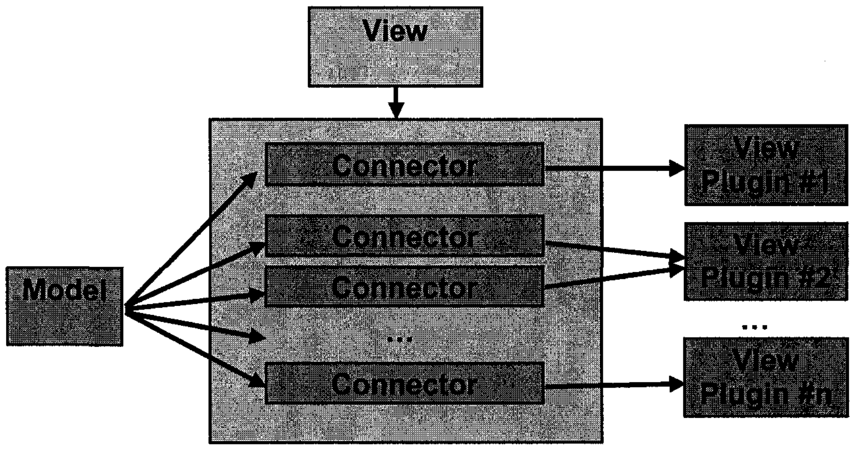


Figure 57 – A view component owns a collection of one or more connectors which receive data from the various outputs of a single model and direct them to their corresponding view plugin.

7.6 Implementation with Reflection

In section 6.5 details of how reflection is used in AVRA for dynamic instantiation and initialization of view plugins were discussed. In addition to the view loader, connectors also take advantage of reflection. This is necessary because connectors do not have compile-time awareness of the interface of the view plugins to which they will be connected. For connectors the reflection is used in two phases. First during their construction in view loader, reflection is used to query the interface and find the corresponding setter methods. Second during the simulation run-time the connectors use reflection to dynamically invoke those methods and pass the parameters as they are made available by the model.

It is possible that there may be conflicts of data types between what the view plugin receives and what the model sends. Although all values are typically numerical, these components may use any combination of int, float, double and long primitive types. This is particularly true because most simulation models compute more precise values (i.e. long and double) where as most view plugin do not need such precisions because they are bound by the pixel limitations and rendering delays; hence they use lower precision data types. It is therefore one of the responsibilities of the model-view connectors to perform such conversions between the incoming and outgoing data. This data conversion in connectors is done similarly than that in the view loader as explained in section 6.5.

CHAPTER 8 - MODEL-VIEW MARKUP LANGUAGE

AVRA introduces MVML, an XML-based markup language for describing the view of a simulation. As it was discussed earlier, in the context of this thesis the view describes how the numerical outputs of a simulation are visualized in a virtual environment. MVML is generic and flexible; it can be used to describe the visualization of a wide variety of simulations.

8.1 MVML Specification

Any given MVML file consists of a single *view* tag that is considered the root of the view description hierarchy. The view tag has a *model* attribute which should reference the model for which this view provides visualization. The model is presumably loaded in AVRA within a hash table that uses the model names as its keys. Therefore in a given AVRA application no two models should have a same name. Current specification of AVRA requires that models be loaded before the views. If a view is loaded before the model that it represents then an exception is thrown. The reason for this is that at the time of loading a view the view loader creates instances of connector and connectors depend on the output variables of models, variables that are unavailable if the model is not constructed yet. Example below shows an empty view that references "Hodgkin-Huxley-cell" model [32] (see section 2.2.3 to see the explanation of a simplified version of Hodgkin-Huxley mode):

```
<view model_name="hodgkin_huxley_squid_axon_1952">

        ...

</view>
```

Within the view tags, the markup contents of MVML can be divided in three categories of tags:

- Graphics: The graphics tag contains such information of the filename, initial visibility state and initial position and orientation. Upon initialization of the framework, a complete VR scene is built based on this description.

- Plugin: For those 3D graphics that have a dynamic state, view plugins are used to connect their visual state with the numerical state of some simulation model.

- Function: Sometimes there is no direct relation between the numerical output of a simulation model and the graphical state of a 3D model and the value needs to be scaled by some function. Such functions may be defined here in the form of MathML syntax [23].

The following section will describe the various elements of MVML in more details. The DTD specification of MVML is available in Appendix D.

8.1.1 Graphics Markup

When the view loader parses a given MVML description, it first constructs a 3D scene by loading the 3D files using appropriate loaders. For example if the filename attribute of the graphics tag reads "cell.wrl" then the VRML loader is used to load the 3D model of the cell and if it read "cell.x3d" then the X3D Loader is used instead. During the initialization phase, only those 3D models with their visibility set to true will be added to scene.

```
<graphics name="cell"

      file="models\\cell.x3d" visible="true"

      position="0 0 0" rotation="0 0 1 0"

/>
```

Some graphics such as the body of the cell in the above example always exist in the scene. However sometimes the existence of a given graphics geometry depends on the numerical output of a simulation model. That is, if the output of the model is within a certain range the graphics object may either be invisible, visible or duplicated into several instances. For example the view description in Figure 58 consists of a cell body that is initially visible and two particle graphics that are initially invisible because later with the help of view plugins they will be duplicated into many visible particles as the simulation runs.

```
<view model_name=" hodgkin_huxley_squid_axon_1952">
      <graphics name="cell" file=".\\models\\Cell.x3d"
            visible="true" position="0 0 0" rotation="0 0 1 0" />
      <graphics name="Na" file=".\\models\\Na.x3d"
            visible="false" />
      <graphics name="K" file=".\\models\\K.x3d"
            visible="false" />

</view>
```

Figure 58 – A MVML description of virtual scene that consists of a cell at origin and Na and K particles that are initially invisible.

8.1.2 View Plugin Markup

The next step is to create instances of the view plugins, initialize their attributes and connect them to the corresponding simulation models. The *class* attribute of the view tag is used to identify the plugin resource (see Figure 59). Our example used java packaging notations such as "xpheve.view.Rotation" to identify the Rotation class as the plugin of interest.

Inside the view tag there may be one more instances of *attribute* tags each with a *name* that identifies which attribute of the view plugin we are referring to. The value of this attribute shall be extracted by parsing the text string child of the attribute tag. For example the notation below sets 0.6 as the value of the MaxValue attribute:

```
<attribute name="MaxValue">

      0.6

</attribute>
```

When the view loader parses the above attribute, it invokes a call to setMaxValue of the view plugin, passing 0.6 as its argument. While most attributes accept simple data such as numbers, there are two special types of attributes that almost always must be set: the graphics and the model output. Since view plugins ultimately visualize particular aspects of a model, they must have reference to the graphics or part of graphics that needs to be updated as the simulation progresses. The target graphics can be identified by the use tag which takes graphics and part names as it attributes:

```
<attribute name="Graphics">

      <use graphics="cell" part="GateH" />

</attribute>
```

In order to establish correspondence between the behavior of the view plugin and the numerical output(s) of a model, one or more attributes of the view plugin must be assigned the *valueof* tag element. Valueof allows identifying a particular output of a model through its component name and variable name. When parsing the XML portion below, the view parser will create a model-view connector that connects this view plugin to the output h of sodium_channel_h_gate component of cell_membrane model:

```
<valueof component="sodium_channel_h_gate" variable="h" />
```

8.1.3 Function Markup

There is not always a direct correspondence between an output of a model and an attribute of a view plugin. Keep in mind that view plugin are generic entities that are designed to be reusable components. As such their parameters are not implemented based of a specific model.

As we saw earlier the voltage output of a neuron cell is typically in the range of -75 to +35. If this variation is to be visualized using the ColorInterpolator then the values must be rescaled to generate a number in the range of 0.0 to 1.0; this can be accomplished using functions.

MVML uses MathML notations (see section 2.2.2) for representing mathematical expression. For example the code below in MVML represents a function named *negate* which takes x as parameter and returns –x:

```
<function name="negate" >
      <math xmlns='http://www.w3.org/1998/Math/MathML'>
            <apply>
                  <times/>
                  <cn type='real'>-1.</cn>
                  <ci>x</ci>
            </apply>
      </math>
</function>
```

In the context of MVML, variable x has special meaning within functions; it always refers to the parameter of the function. This also conveys that in MVML functions can receive one and only one parameter. This is not a limitation as functions are mean to be simple means of scaling single values.

Figure 59 shows a complete MVML description for creating a single view plugin and assigning attributes to it. Note that it uses function negate to negate the output of simulation model before sending it to view plugin.

```
<plugin class="xpheve.laws.AnimatedParticleLaw">
      <attribute name="NumOfSourceParticles">
            150
      </attribute>
      <attribute name="rate">
            <valueof function="negate"
                  component="sodium_channel"
                  variable="i_Na" />
      </attribute>
</plugin>
```

Figure 59 – A MVML description of a view plugin that generates and moves particles based on the output i_Na of the model.

8.2 View Loader

In chapters 5, 6 and 7 there have been discussions on how the view loader handles specific tasks. This section completes the discussion on the process of parsing MVML files and generating views, view plugins and connectors.

8.2.1 Extracting the DOM Structure

Figure 60 shows a high-level perspective of the view loader and various failures that the view loader detects when loading a given MVML file. When it comes to parsing XML-based files there are two possible standardized ways: using SAX and using DOM. SAX (Simple API for XML) is a memory efficient technique that uses call backs as it reads the XML contents [33]. DOM (Document Object Model) on the other hand generates a detailed tree structure that gives the application a complete snap shot of the entire XML file [34]. This tree structure is easy to traverse and is the most popular choice for small and medium XML files. In AVRA, we use DOM trees because MVML generally do not grow very large and the implementation of the view loader is made easier with DOM trees.

The DOM tree construction will fail if there are severe structural problems with the XML content of the MVML. If this happens the view loading is aborted. If the DOM tree is successfully generated, the model name is extracted from the tree. At this point the view loader expects that the corresponding model is already loaded. It tries to find it using a simple hash table which, given a name, quickly returns the corresponding model. If model is not found the view loading is aborted. If the model is available, it is retrieved and the traversal of the DOM tree commences.

During this traversal the view loader may face problems with loading plugins, finding setter methods that corresponding to given attributes or being unable to properly match data types of the parameters. Any of these issues results in immediate abortion as the view loader cannot proceed without resolving them. If successful, the resulting view is constructed and returned along with the graphics and connectors that it contains.

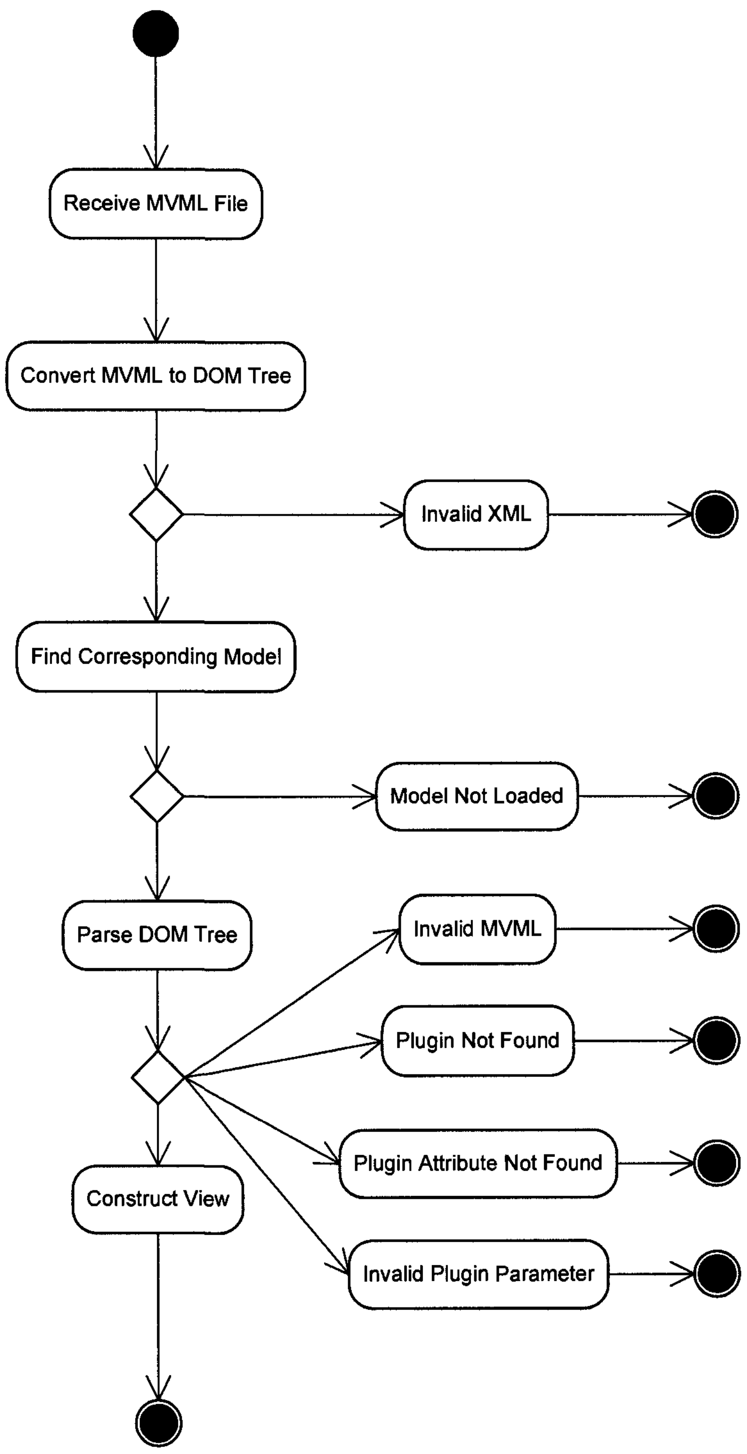


Figure 60 – High-level activity diagram of View Loader with possible Exceptions

8.2.2 Generating AVRA Components

In section 6.4, we saw how the view loader retrieves the primitive values and invokes the setter methods of a view plugin, passing these values as its parameters. Figure 61 is the complete activity diagram of view loader that outlines the construction of connectors, graphics, and functions in addition to that of primitive values.

As the highest level of the DOM tree the view loaders identifies whether a function or graphics or view plugin tag is currently being processed. For each of these three, the corresponding tag is processed and the control is returned to top of the activity diagram to process the next MVML tag. When processing graphics tags, the corresponding graphics loader is instantiated and is used to load the graphics file. If visible the 3D graphics is then added to the scene immediately, otherwise it is saved in memory within a hash data structure.

Functions are extracted by MathMLLoader into a data structure that allows quick evaluations and substitutions during the simulation run-time. The evaluations and substitutions of functions are often triggered by the connectors for the purpose rescaling the outputs of the simulation models.

If a view plugin is in construction, then all its child attributes must be processed one at a time until no more attributes is left at which point control is passed to the higher tree branch for processing the next MVML tag. This process continues until all children of MVML DOM tree are served.

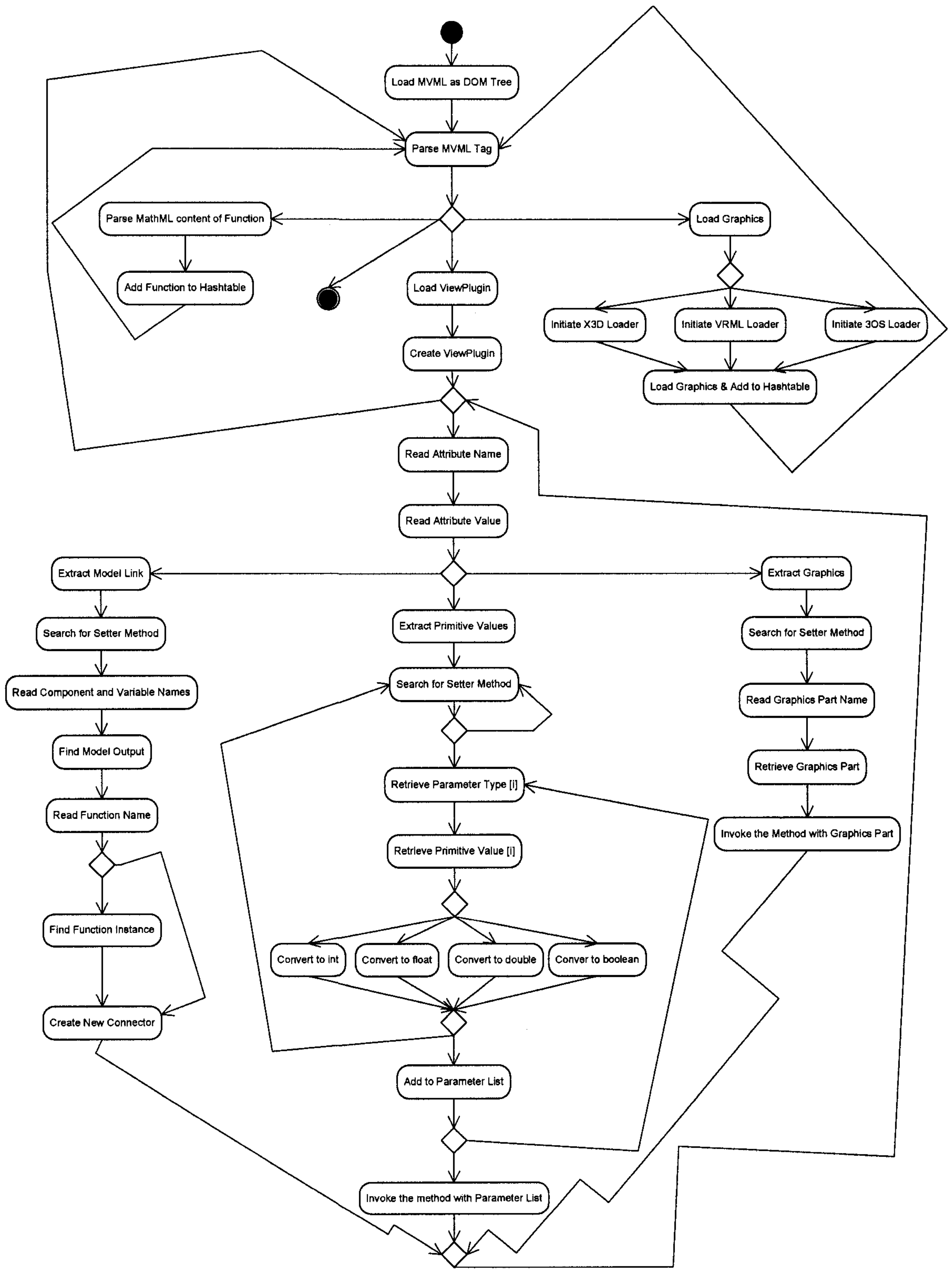


Figure 61 – Complete activity diagram of view loader when parsing MVML

CHAPTER 9 - APPLICATIONS & RESULTS

9.1 AVRA Implementation

We implemented a prototype of AVRA which includes most of the features discussed in the previous chapters. The implementation is on a Java platform and uses Java3D for its 3D graphics rendering and JNI (Java Native Invocation) for haptic support.

The current implementation of ModelLoader accepts CellML models as input and dynamically constructs internal simulation components. Upon starting the simulation engine the output of those models are continuously calculated. Figure 62 shows a sample numerical output that the system generates when a simple action potential model of a neuron is loaded. The initial state of intra and extra sub spaces are set as follows:

Component	**Variable**	**Initial Value**
IntraSubSpace	Na	10
	Ca	20
ExtraSubSpace	Na	30
	Ca	40

Appendices B1, B2 and B3 show snap shots of the numerical simulation of a cell with the above initial state, at time 0, at 1 second and at 1 minute time.

A complete view loader according to the MVML specification is also implemented. It is capable of loading X3D and VRML graphical models and generating a dynamic visualization of the model. The rest of this chapter will demonstrate how this implementation of AVRA can be used in the development of several simulation case studies.

```
Model = basic_ep_model
Component = environment************************
Local:
Inputs:
Outputs:
time = 0.0
Component = extra_cellular_space************************
Local:
Ca = 40.0
Na = 30.0
Inputs:
I_Ca = -3.0E-7
time = 0.0
I_Na = -2.0E-7
Outputs:
Ca = 40.0
Na = 30.0
Component = intra_cellular_space************************
Local:
Ca = 20.0
Na = 10.0
Inputs:
I_Ca = -3.0E-7
time = 0.0
I_Na = -2.0E-7
Outputs:
Ca = 20.0
Na = 10.0
Component = cell_membrane************************
Local:
v_Na = 1.0E-8
v_Ca = 1.5E-8
Inputs:
Ca_e = 40.0
Na_e = 30.0
Na_i = 10.0
Ca_i = 20.0
Outputs:
I_Ca = -3.0E-7
I_Na = -2.0E-7
```

Figure 62 – Sample numerical output of the simulator when receiving basic_ep_model as its input model.

9.2 *Case Study 1: Action Potential Simulation*

This section discusses how AVRA can be used to construct a VR application that simulates action potential, a common biological phenomena that causes electrical charges to travel within neurons of humans and animals.

Without getting into details, action potential is a direct result of the movement of ions across the membrane of a neuron cell. At rest, a neuron cell has a negative charge. Upon occurrence of a stimulus, ion gates open and the positive ions rush into the cell, make it positively charged. This is called action-potential.

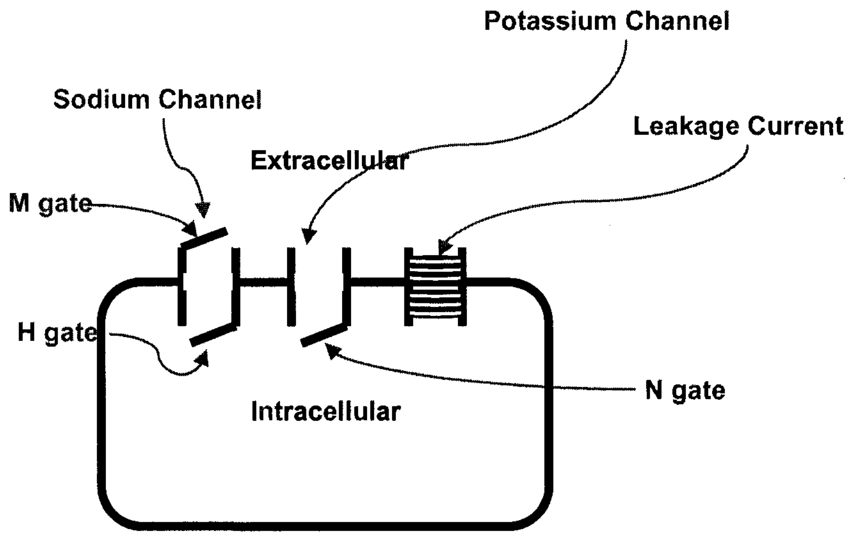


Figure 63 – The diagram of the membrane and intra and extra spaces connected through ionic gates

Since the 1950s, there have been many biological experiments that resulted in mathematical formulas that describe the nature of ionic movements in and out of the cells. These mathematical formulas describe such attributes as the energy, voltage, rate of ionic movements, density of ions and state of the ionic gates at any given time. AVRA allows users to selectively connect the outputs of a model to the inputs of some visual animation. In this particular example, the goal is to load the hudgin-huxley model of action potential in order to visualize the ionic state of a cell in 3D during action potential. The mathematical model of hudgin-huxley is conveniently available in CellML format (www.cellml.org). Figure 63 shows the various components involves in the hudgin-huxley model. Based on the values of time and stimulus as input, this model provides the outputs listed in Table 4.

Output	Description
m	Activation coefficient of Na channel
h	Inactivation coefficient of Na channel
n	Activation coefficient of K channel
i_Na	Sodium current.
i_K	Potassium current.
i_L	Leakage current.
V	Membrane voltage.

Table 4 – The main outputs of Hudgin-Huxley model

The 3D models are created using FluxStudio and exported to X3D format. They consist of three files: Na.x3d which is the 3d model of a single Sodium ion, K.x3d which is the 3D model of a single Potassium ion, and Cell.x3d which is the 3D model of a cell consisting of seven parts as shown in Figure 64. To bring this 3D model to life, all we need to do is to create a MVML description of the simulation and load it in the framework. In our example, the objective is to connect the model outputs as outlined in Table 4 to the animation of the various parts of the 3D scene. Figure 65 shows example of a MVML that connects V, m and i_Na to their corresponding animation modules. The resulting simulation is shown in Figure 66 and Figure 67. As the outputs of the hudgin-huxley model changes, the gates of Na and K channels open and close and the Na and K particles move back and forth across the membrane of the cell through these gates.

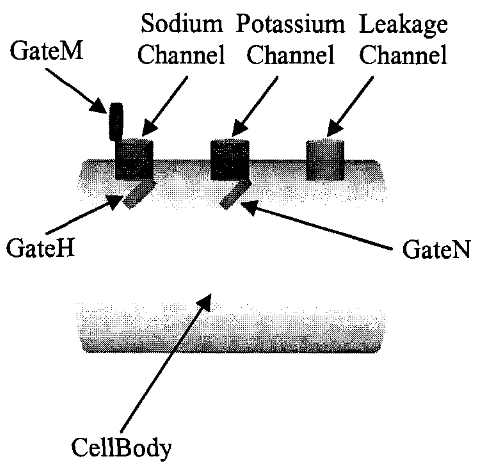
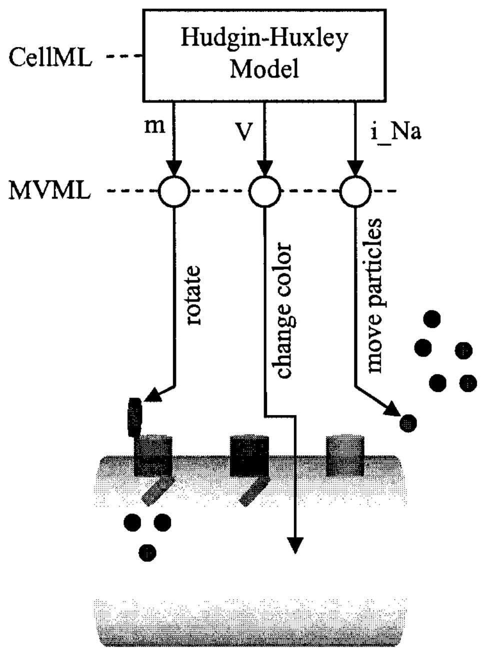


Figure 64 – The 7 named body parts of the 3D model of a cell membrane (cell.x3d). These part names must be referenced by their exact names in MVML description of the scene.



```
<view model_name="hudgin_huxley">
  <graphics name="cell" file="Cell.x3d"
    visible="true" position="0 0 0"
    rotation="0 0 1 0" />
  <graphics name="Na" file="Na.x3d"
    visible="false" />

  <plugin class="xpheve.laws.RotationLaw">
    <attribute name="Graphics">
      <use graphics="cell" part="GateM" />
    </attribute>
    <attribute name="value">
      <valueof
        component="sodium_channel_m_gate"
        variable="m" />
    </attribute>
  </plugin >

  <plugin class="xpheve.laws.ColorLaw">
    <attribute name="Graphics">
      <use graphics="cell" part="CellBox" />
    </attribute>
    <attribute name="value">
      <valueof component="membrane"
        variable="V" />
    </attribute>
    <attribute name="MaxColor"> 1 1 0
    </attribute>
  </plugin>

  <plugin class="xpheve.laws.AnimParticleLaw">
    <attribute name="NumOfSourceParticles">
      150
    </attribute>
    <attribute name="rate">
      <valueof component="sodium_channel"
        variable="i_Na" />
    </attribute>
  </plugin>
</view>
```

Figure 65 – Example of MVML description which connects the V, m, and i_Na outputs of the model to the three view plugins responsible for controlling the rotation, color and particle movements of the 3D model parts.

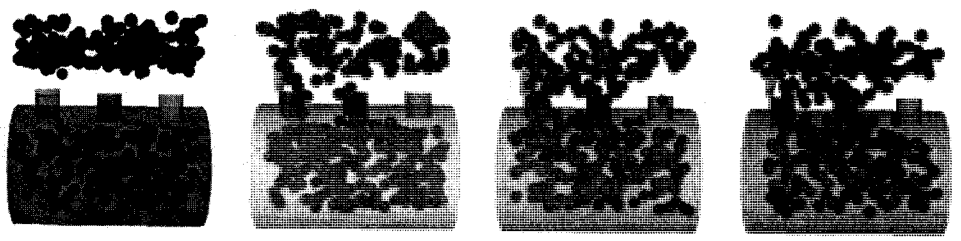


Figure 66 – Screenshots of the simulations during occurrence of action potential. The color of cell body changes to yellow to denote proportional increase in Voltage (V). Particles move in and out to denote Sodium (red) and Potassium (blue) currents (i_Na, i_K).

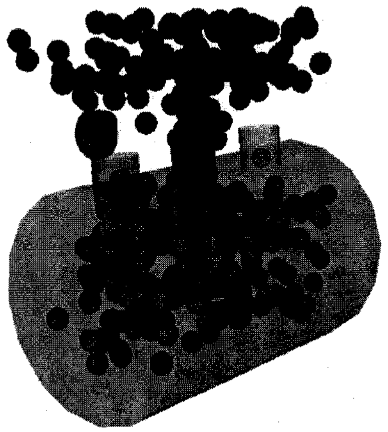


Figure 67 – The generated simulation from alternative perspective.

9.3 *Case Study 2: Neuron Simulation with Multiple Graphics*

This case study is an extension of the previous case study to demonstrate the reusability and extensibility that AVRA provides for VR simulations. In this case study, the axon of a neuron is modeled by constructing a VR simulation that contains a system of multiple action potential chambers as implemented in the previous case study. Neurons and in particular axons function by transmitting electricity through a sequence of action potentials that is triggered on one side and eventually arrives on the other side of the axon.

Figure 68 demonstrates how the physiology of axon can be simulated using a system of multiple action potential chambers. Case study 1 presented the simulation of a single action potential chamber with a membrane and interior and exterior spaces. In Figure 68, six instances of this model is replicated to transfer electrical signals over a longer distance. In this scenario an external stimulus triggers the action potential in the left chamber. This would result in a sequence of five additional action potentials that ends with the right-most chamber. Within a neuron this set of five chambers could resemble the axon which although behaves as a set of discrete chambers but is in fact a single entity. This is where AVRA can be conveniently configured to display the desired visualization. Since the view description and the simulation models are independent of each other in AVRA, we can visualize axon as a single entity or a set of multiple chambers regardless of the simulation model structure.

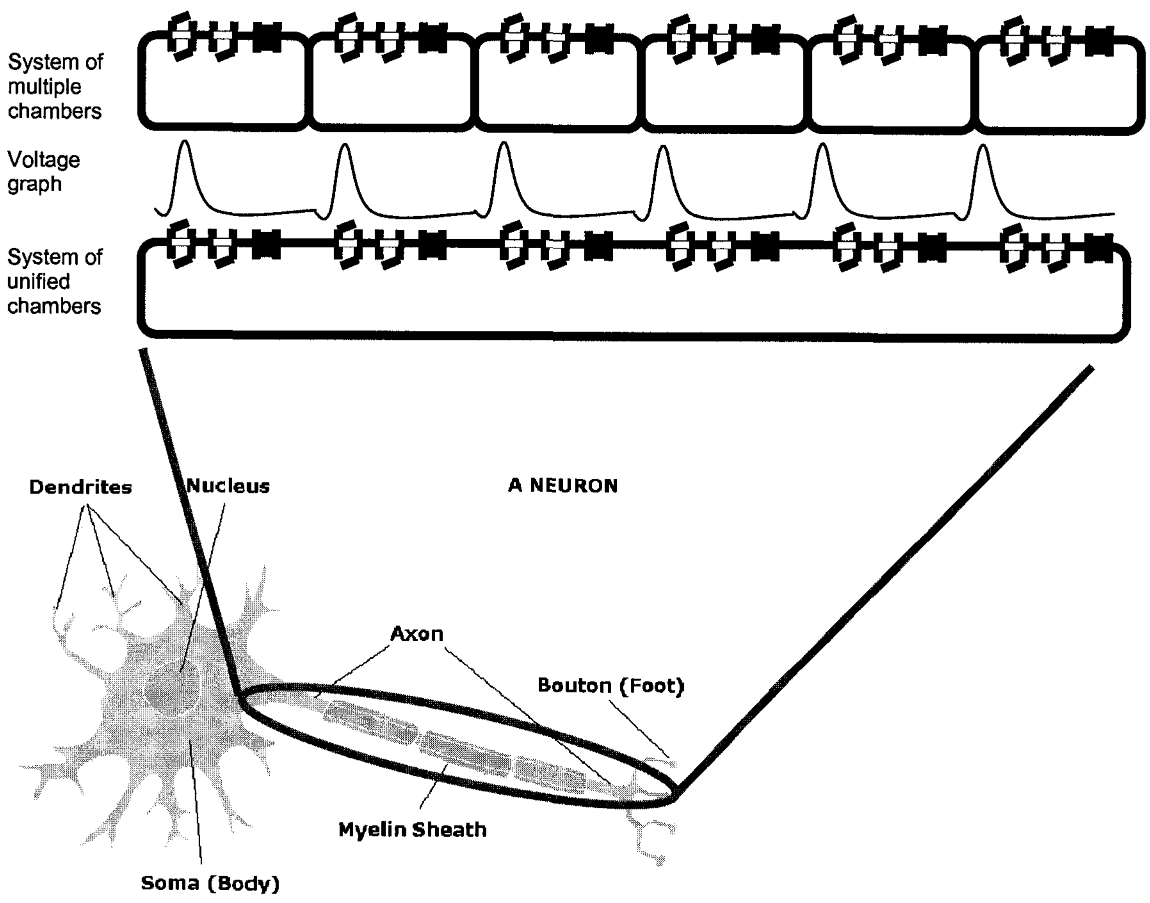


Figure 68 – Simulating axon with a system of multiple action potential chambers

```
<model name="axon_model">

  <import xlink:href="hodgkin_huxley.xml">
    <units name="millisecond" units_ref="millisecond" />
    ...

    <component name="membrane_1" component_ref="membrane" />
    <component name="sodium_channel_1" component_ref="sodium_channel" />
    <component name="sodium_channel_m_gate_1" component_ref="sodium_channel_m_gate" />
    <component name="sodium_channel_h_gate_1" component_ref="sodium_channel_h_gate" />
    <component name="potassium_channel_1" component_ref="potassium_channel" />
    <component name="potassium_channel_h_gate_1"
               component_ref="sodium_channel_h_gate" />
  </import>

  <import xlink:href="hodgkin_huxley.xml">
    <units name="millisecond" units_ref="millisecond" />
    ...

    <component name="membrane_2" component_ref="membrane" />
    <component name="sodium_channel_2" component_ref="sodium_channel" />
    <component name="sodium_channel_m_gate_2" component_ref="sodium_channel_m_gate" />
    <component name="sodium_channel_h_gate_2" component_ref="sodium_channel_h_gate" />
    <component name="potassium_channel_2" component_ref="potassium_channel" />
    <component name="potassium_channel_h_gate_2"
               component_ref="sodium_channel_h_gate" />
  </import>

  <connection>
    <map_components component_1="membrane_1" component_2="membrane_2" />
    <map_variables variable_1="V" variable_2="I_stim" />
  </connection>

  ...

</model>
```

Figure 69 – The CellML code for importing two instances of Hodgkin-Huxley and connecting them together through V (voltage) and I_stim (stimulus current) variables.

Figure 69 demonstrates how CellML allows reusing of the models by importing them into new models. In this example the simulation model is that of an axon that contains several action potential chambers. The import tags are used to import and reuse the hodgk-huxley model for action potential seen in the previous case study. This mode is imported twice in order to create two instances of it. The current specification of CellML does not allow importing an entire model; rather it allows importing components selectively. This is convenient if only some of the components need to be reusable but inconvenient if duplicating an entire model is desired. For the latter, one needs to import all components one by one as shown in Figure 69 demonstrates. The final task is to mathematically connect the output V of the first model to the input I_stim of the second model. This would allow an action potential in the first model to stimulate the second model and therefore trigger a cascading effect. The remaining four action potential chambers can be added in a similar manner in order to complete our model of axon with six chambers.

Now that the mathematical model of the axon is ready, we can create its MVML description for specifying how the axon is to be visualized. Figure 70 highlights a partial MVML description that visualizes three state variables of the action potential model in two of the axon chambers. As it can be seen in the figure the only difference between this MVML and that of the previous case study is the duplicate of view plugins that account for outputs of two different model entities. In particular there are two view plugins for visualizing the M gates, two view plugins for visualizing the voltage and two view plugins for generating particles for sodium ions. These duplicated view plugins received their inputs from different model entities.

There are also two separate instances of Cell.x3d graphics namely Chamber_1 and Chamber_2. The view plugin that receives its input from the components of model_1 always reflects the result into the graphics parts of Chamber_1. Similarly model_2 is visualized within the boundaries of Chamber_2. Note that the AnimParticleLaw plugins do not reference Chamber_1 nor Chamber_2. Instead they both reference Na which the graphical representation of Sodium. This is because this view plugin generates its own duplication of the graphical models; therefore both chambers can share a same graphics for that.

```
<view model_name="axon_model">
  <graphics name="Chamber_1" file="Cell.x3d" visible="true" position="0 0 0" />
  <graphics name="Chamber_2" file="Cell.x3d" visible="true" position="1 0 0" />

  <graphics name="Na" file="Na.x3d" visible="false" />

  <!-- Chamber 1 Visualization -->
  <plugin class="xpheve.laws.RotationLaw">
    <attribute name="Graphics">
      <use graphics="Chamber_1" part="GateM" />
    </attribute>
    <attribute name="value">
      <valueof component="sodium_channel_m_gate_1" variable="m" />
    </attribute>
  </plugin >

  <plugin class="xpheve.laws.ColorLaw">
    <attribute name="Graphics">
      <use graphics="Chamber_1" part="CellBox" />
    </attribute>
    <attribute name="value">
      <valueof component="membrane_1" variable="V" />
    </attribute>
    <attribute name="MaxColor"> 1 1 0 </attribute>
  </plugin>

  <plugin class="xpheve.laws.AnimParticleLaw">
    <attribute name="Graphics">
      <use graphics="Na" />
    </attribute>
    <attribute name="NumOfSourceParticles"> 150 </attribute>
    <attribute name="rate">
      <valueof component="sodium_channel_1" variable="i_Na" />
    </attribute>
  </plugin>

  <!-- Chamber 1 Visualization -->
  <plugin class="xpheve.laws.RotationLaw">
    <attribute name="Graphics">
      <use graphics="Chamber_2" part="GateM" />
    </attribute>
    <attribute name="value">
      <valueof component="sodium_channel_m_gate_2" variable="m" />
    </attribute>
  </plugin >

  <plugin class="xpheve.laws.ColorLaw">
    <attribute name="Graphics">
      <use graphics="Chamber_2" part="CellBox" />
    </attribute>
    <attribute name="value">
      <valueof component="membrane_2" variable="V" />
    </attribute>
    <attribute name="MaxColor"> 1 1 0 </attribute>
  </plugin>

  <plugin class="xpheve.laws.AnimParticleLaw">
    <attribute name="Graphics">
      <use graphics="Na" />
    </attribute>
    <attribute name="NumOfSourceParticles"> 150 </attribute>
    <attribute name="rate">
      <valueof component="sodium_channel_2" variable="i_Na" />
    </attribute>
  </plugin>
</view>
```

Figure 70 – The MVML description for visualizing the change in sodium gate, sodium current and voltage of an axon that consists of two chambers

Figure 71 demonstrates the resulting infrastructure that is generated by AVRA upon loading the axon model and its view description. The axon model conceptually contains two internal models (all its sub-components are duplicated). AVRA generates a single view component with six view plugins for this simulation. The view plugins 1.1 through 1.3 represent RotationLaw, ColorLaw and AnimParticleLaw in that order. They visualize the various outputs of Model_1 (Chamber_1 components) that is reflected on the graphics of the first chamber (from left).

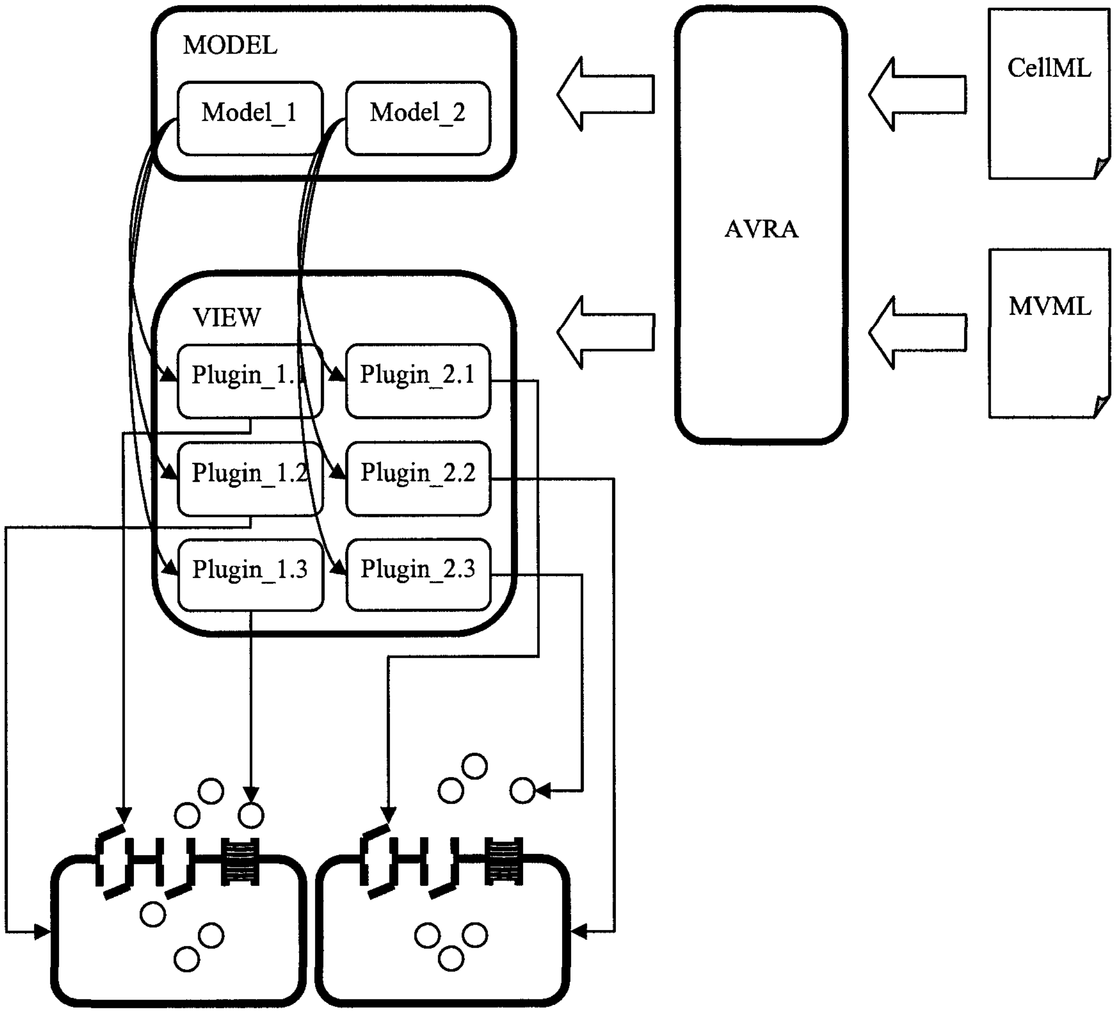


Figure 71 – The system of interconnected components that is automatically generated and maintained by AVRA when CellML and MVML files of case study 2 is loaded

9.4 Case Study 3: Neuron Simulation with Unified Graphics

The problem with the resulting simulation in the previous case study is that although it is an effective visualization which displays the detailed physiological phases within chambers of axon, it is not visually realistic. In neurons, the axon is not a set of separate chambers that are attached to each other. It is rather a unified entity that only behaves as if it is consisted of separate chambers. Physiologically the axon is a long segment that allows action potential at particular points where its surface is not covered by Myelin Sheath (see Figure 68). In addition, the axon membrane gates look nothing like the mechanical rotating door that was used in the previous case studies.

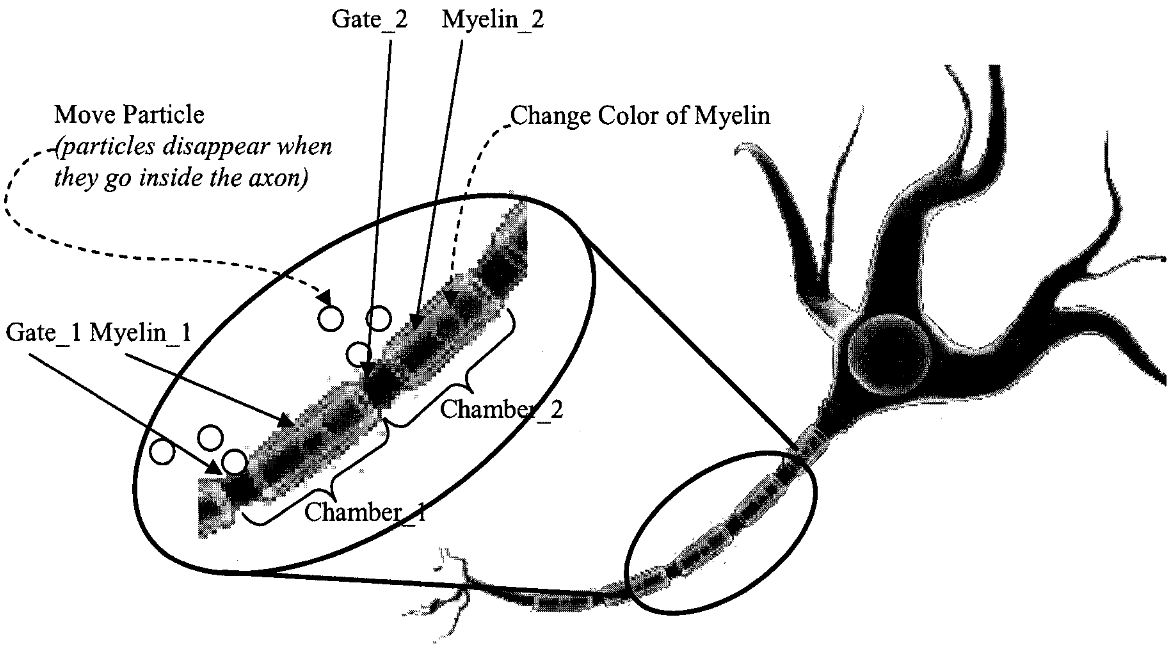


Figure 72 – The visualization of neuron that uses a single 3D model of a neuron and changes the colors of its Myelin Sheath to visualize change in the voltage and animates the Na particles to visualize change in Sodium concentration.

This case study visualizes multiple instances of a simulation model within a same graphical model. That is instead of loading multiple graphics to represent the chambers a single graphics of a neuron will be loaded in the scene. Since in reality the membrane gates are too small to be visible at this scale we choose to omit them in this case study. There the resulting simulation will merely show the visualization of voltage and those sodium particles that are outside of the axon. Figure 72 shows the desired visualization for this case study. Note that the sodium ion particles are made abnormally large for easier visualization.

```
<view model_name="axon_model">
  <graphics name="Neuron" file="neuron.x3d" visible="true" position="0 0 0" />
  <graphics name="Na" file="Na.x3d" visible="false" />

  <!-- Chamber 1 Visualization -->
  <plugin class="xpheve.laws.ColorLaw">
    <attribute name="Graphics">
      <use graphics="Neuron" part="Myelin_1" />
    </attribute>
    <attribute name="value">
      <valueof component="membrane_1" variable="V" />
    </attribute>
    <attribute name="MaxColor"> 1 1 0 </attribute>
  </plugin>

  <plugin class="xpheve.laws.AnimParticleLaw">
    <attribute name="Graphics">
      <use graphics="Na" />
    </attribute>
    <attribute name="NumOfSourceParticles"> 150 </attribute>
    <attribute name="rate">
      <valueof component="sodium_channel_1" variable="i_Na" />
    </attribute>
    <attribute name="intermediatePoint"> 1 1 0 </attribute>
  </plugin>

  <!-- Chamber 2 Visualization -->
  <plugin class="xpheve.laws.ColorLaw">
    <attribute name="Graphics">
      <use graphics="Neuron" part="Myelin_2" />
    </attribute>
    <attribute name="value">
      <valueof component="membrane_2" variable="V" />
    </attribute>
    <attribute name="MaxColor"> 1 1 0 </attribute>
  </plugin>

  <plugin class="xpheve.laws.AnimParticleLaw">
    <attribute name="Graphics">
      <use graphics="Na" />
    </attribute>
    <attribute name="NumOfSourceParticles"> 150 </attribute>
    <attribute name="rate">
      <valueof component="sodium_channel_2" variable="i_Na" />
    </attribute>
    <attribute name="intermediatePoint"> 1 1 0 </attribute>

  </plugin>
</view>
```

Figure 73 – The MVML description for visualizing the change in sodium current and voltage of an axon that consists of a unified graphical model

The MVML description for achieving the above is shown in Figure 73. This version of MVML references a single 3D model of a neuron (neuron.x3d). The two ColorLaw instances changes the color of different parts of a same graphical model namely Mylein_1 and Mylein_2. The AnimParticleLaw directs particles in and out of the sodium gates. Although the gates themselves are invisible in this case study, the AnimParticleLaw still needs to know about its whereabouts in order to animate the particles accordingly. In this case the first AnimParticleLaw receives the coordinates of Gate_1 (1, 1, 0) and the second one receives the coordinates of the Gate_2 (2, 2, 0) as their intermediate points.

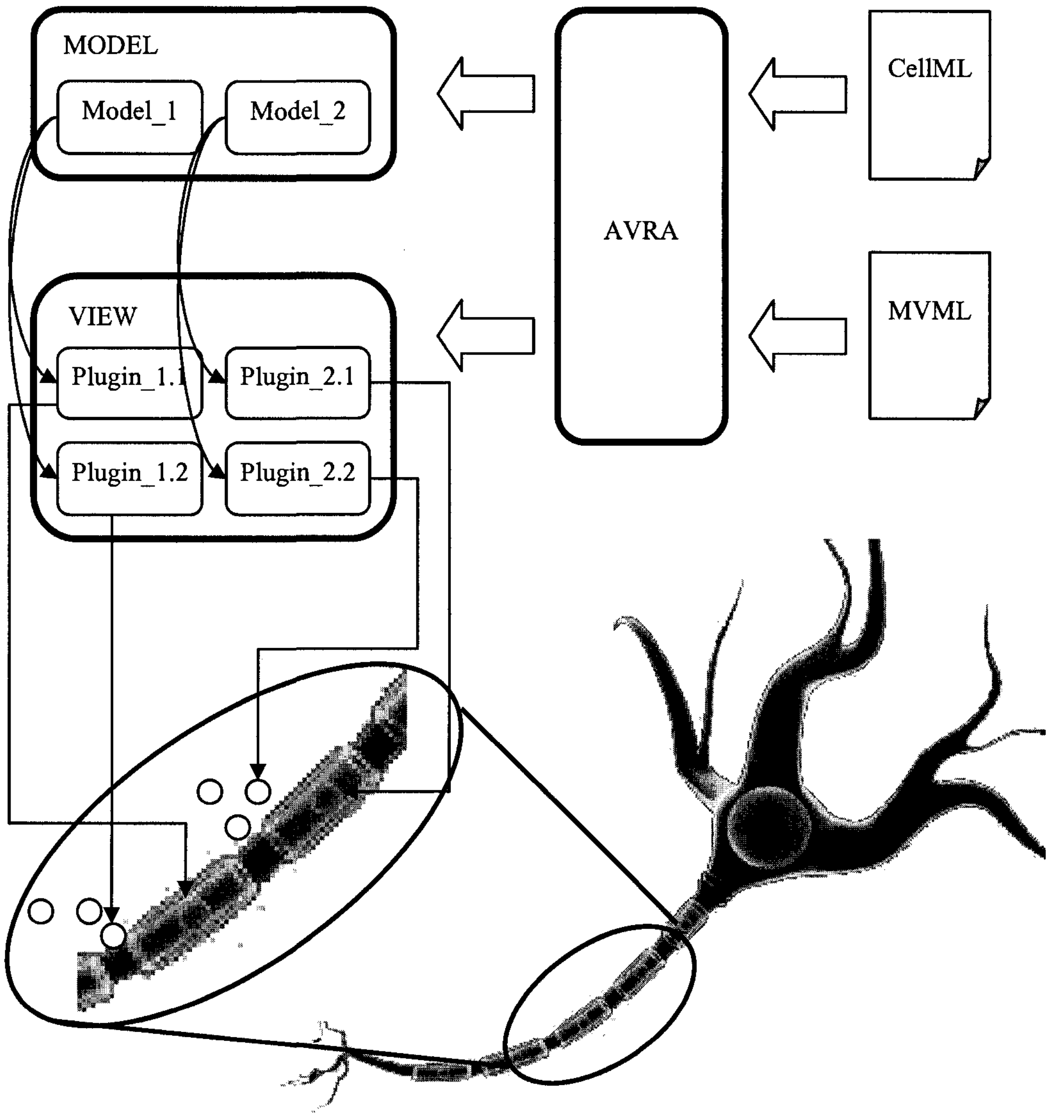


Figure 74 – The system of interconnected components that is automatically generated and maintained by AVRA when CellML and MVML files of case study 2 is loaded

Figure 74 demonstrate how AVRA constructs model and view components when loading the above MVML description. The main different between this structure and that of the previous case study is that we have only four view plugins and they control the visual attributes of a single unified model although different parts of it.

9.5 Case Study 4: Simulation of Unhealthy Neuron

Up until now we used a single simulation model (although several instances of it) in order to simulated a neuron axon that is consisted of five healthy segments (chambers) that are exact mathematical duplicate of each other. In this case study we will see how AVRA allows multiple simulation models to work in harmony with each other.

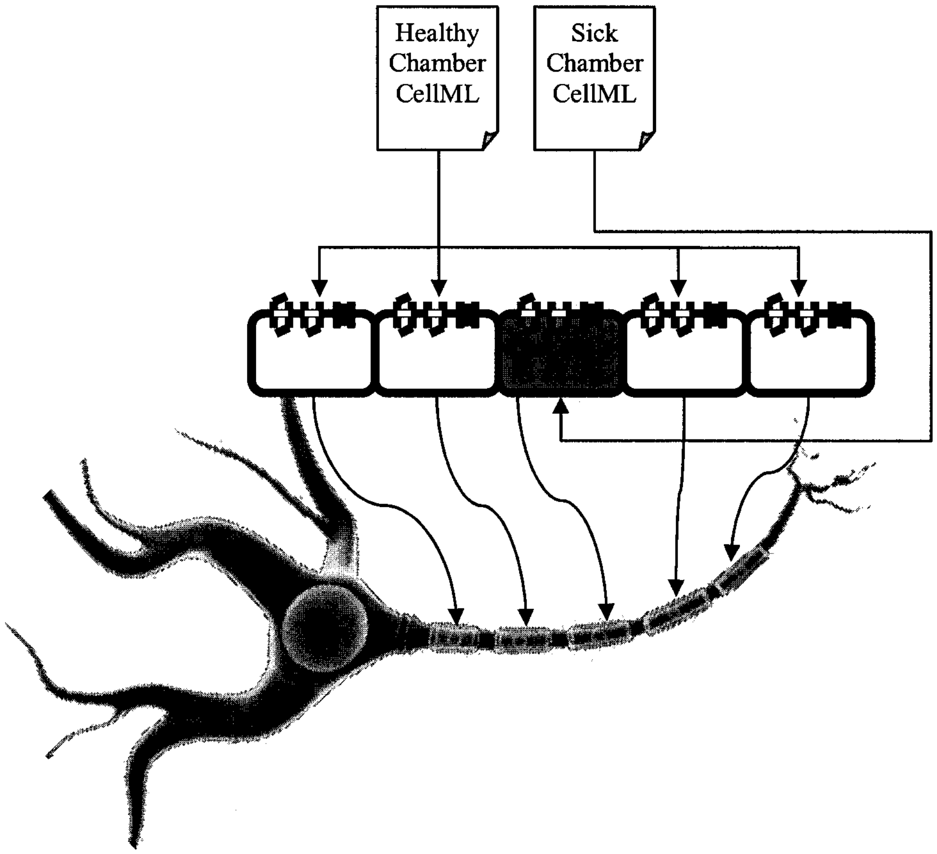


Figure 75 – Using multiple simulation models to visualize healthy and sick segments of a neuron axon

```
<model name="axon_model">

  <import xlink:href="hodgkin_huxley.xml">
    <units name="millisecond" units_ref="millisecond" />
    ...

    <component name="membrane_2" component_ref="membrane" />
    <component name="sodium_channel_2" component_ref="sodium_channel" />
    <component name="sodium_channel_m_gate_2" component_ref="sodium_channel_m_gate" />
    <component name="sodium_channel_h_gate_2" component_ref="sodium_channel_h_gate" />
    <component name="potassium_channel_2" component_ref="potassium_channel" />
    <component name="potassium_channel_h_gate_2"
               component_ref="sodium_channel_h_gate" />
  </import>

  <import xlink:href="sick_chamber.xml">
    <units name="millisecond" units_ref="millisecond" />
    ...

    <component name="membrane_3" component_ref="membrane" />
    <component name="sodium_channel_3" component_ref="sodium_channel" />
    <component name="sodium_channel_m_gate_3" component_ref="sodium_channel_m_gate" />
    <component name="sodium_channel_h_gate_3" component_ref="sodium_channel_h_gate" />
    <component name="potassium_channel_3" component_ref="potassium_channel" />
    <component name="potassium_channel_h_gate_3"
               component_ref="sodium_channel_h_gate" />
  </import>

  <connection>
    <map_components component_1="membrane_2" component_2="membrane_3" />
    <map_variables variable_1="V" variable_2="I_stim" />
  </connection>

  ...

</model>
```

Figure 76 – The CellML code for importing one instances of Hodgkin-Huxley model and one instance of Sick-Chamber model and connecting them together through V (voltage) and I_stim.

In this case study an axon chamber is considered sick if it has a different underlying mathematics than that of Hodgin-Huxley model as used in the previous case studies. For example if the rate of change in sodium density is calculated differently or if some gates do not open, the axon segment is considered abnormal or sick. These malfunctions may result in action potentials being fired irregularly or not at all. Figure 76 demonstrates the desired configuration in which different simulation models are connected to different graphical parts of a neuron.

Figure 76 demonstrate a new version of our axon_model in CellML which is reconfigured to import two different models: a healthy Hudgkin-Huxley model as model #2 and an arbitrary sick chamber model as model #3. In this case study we created the sick_chamber.xml as a duplicate of hudgkin_huxley except the sodium gate M does not open in this model. Since the interface of these two models (i.e. the variables and components names) is exactly the same, we can reuse the MVML descriptions created in the previous case studies exactly as is and without any modifications. This is because the view plugin of the 3rd chamber receives the output of model #3 from its various components and model #3 happened to be our sick model as per Figure 76.

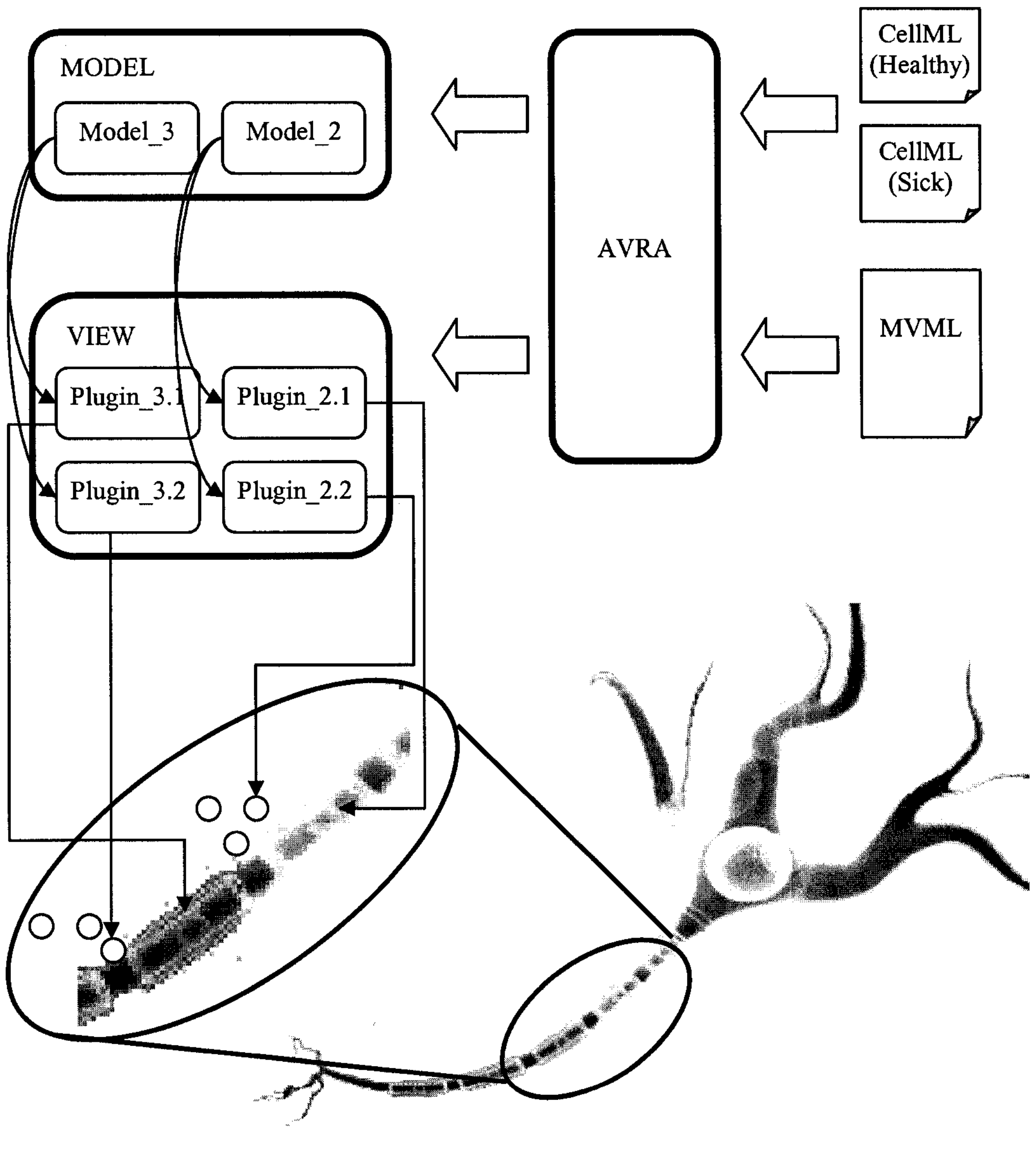


Figure 77 – The system of interconnected components that is automatically generated and maintained by AVRA when the simulation model that contains healthy and sick CellML is loaded

Figure 77 shows the result infrastructure that is similar than the previous study but present a different visualization due to the changes in the underlying simulation models that are loaded as CellML files. In this structure the view plugins that control the appearance of the 3rd axon segment does not lid up the corresponding graphics because the corresponding model computes zero as the current voltage no matter how much stimulus it receives from the previous chamber. In addition since this particular chamber does not generate action potential it will not stimulate the next healthy chamber hence the communication link in the axon is mathematically and visually cut off.

9.6 Case Study 5: Interactive Simulation of a Neuron

Interactive simulations can take advantage of the various features in AVRA but they require direct manipulation of view plugins from the higher level applications. This case study demonstrates how applications that are built on top of AVRA can add haptic interaction to the neuron simulation as generated in the case study 4.

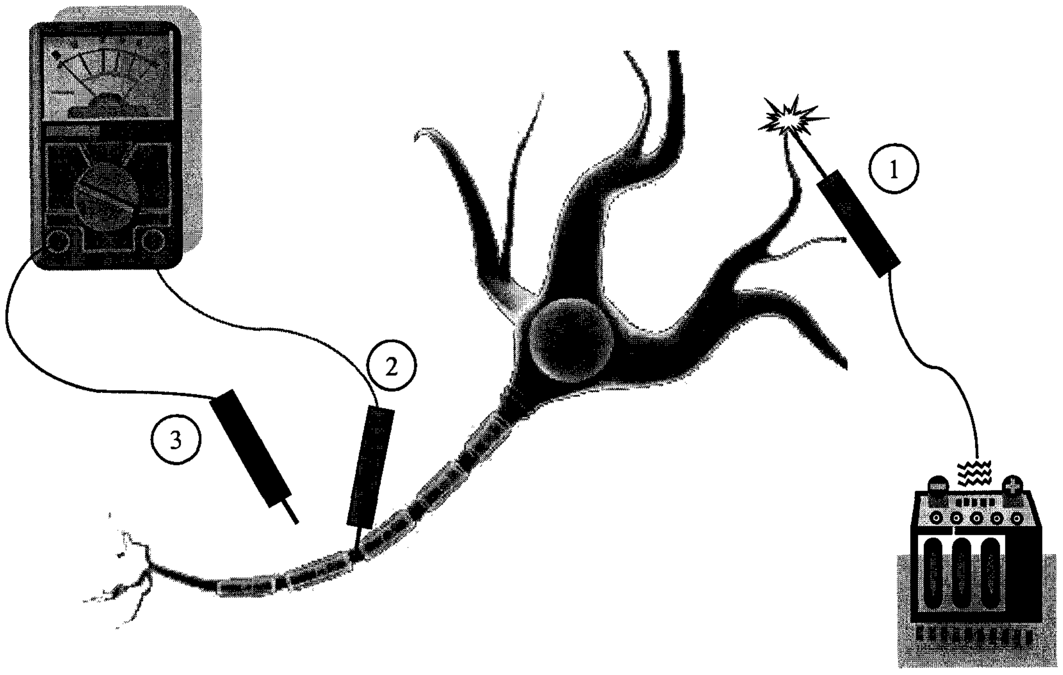


Figure 78 – Interaction with the model of a neuron with three probes to stimulate action potential and measure voltage

The desired scenarios of this application include manually triggering action potential through electrical shocks and measuring the voltage across the membrane of an axon through a specialized mulitmeter. Figure 78 demonstrates the desired interaction with the model of neuron as planned for this case study. In particular, whenever the probe 1 collides with the neuron, the nearest axon chamber receives an stimulus that subsequently results in an action potential. In the figure, the probe collides with the nucleus of the neuron which is closes to axon chamber #1, therefore that chamber will receive an stimulus. Measuring the voltage across a membrane is achieved by placing one probe of multimeter inside and the other outside of the chamber as shown in the figure with probes.

In this case study haptic devices are used to control the three probes of the multimeter and the power source. The challenge of this scenario is that the simulation model inputs and the visual output depend on a more complex dynamic than what can be specified in MVML. For example, the numerical input value for *stimulus* in the axon_model depends on the visual location of the probe 1. Also the visual state of the multimeter depends on which axon chamber is pinned by probe 2 and the whereabouts of probe 3. AVRA allows the higher level applications to define such complex behaviors and conveniently integrate them with the rest of the simulation.

```
<view model_name="axon_model">
  ...
  <graphics name="Multimeter" file="multmtr.x3d" visible="true" position="5 5 0" />

  ...

  <!-- Multimeter -->
  <plugin class="xpheve.laws.RotationLaw">
    <attribute name="Graphics">
      <use graphics="Multimeter" part="handle" />
    </attribute>
    <attribute name="value">
      <valueof component="membrane_x" variable="V" />
    </attribute>
    <attribute name="minValue"> -100 </attribute>
    <attribute name="maxValue"> +100 </attribute>
  </plugin>

</view>
```

Figure 79 – The MVML description for visualizing the change in sodium current and voltage of an axon that consists of a unified graphical model

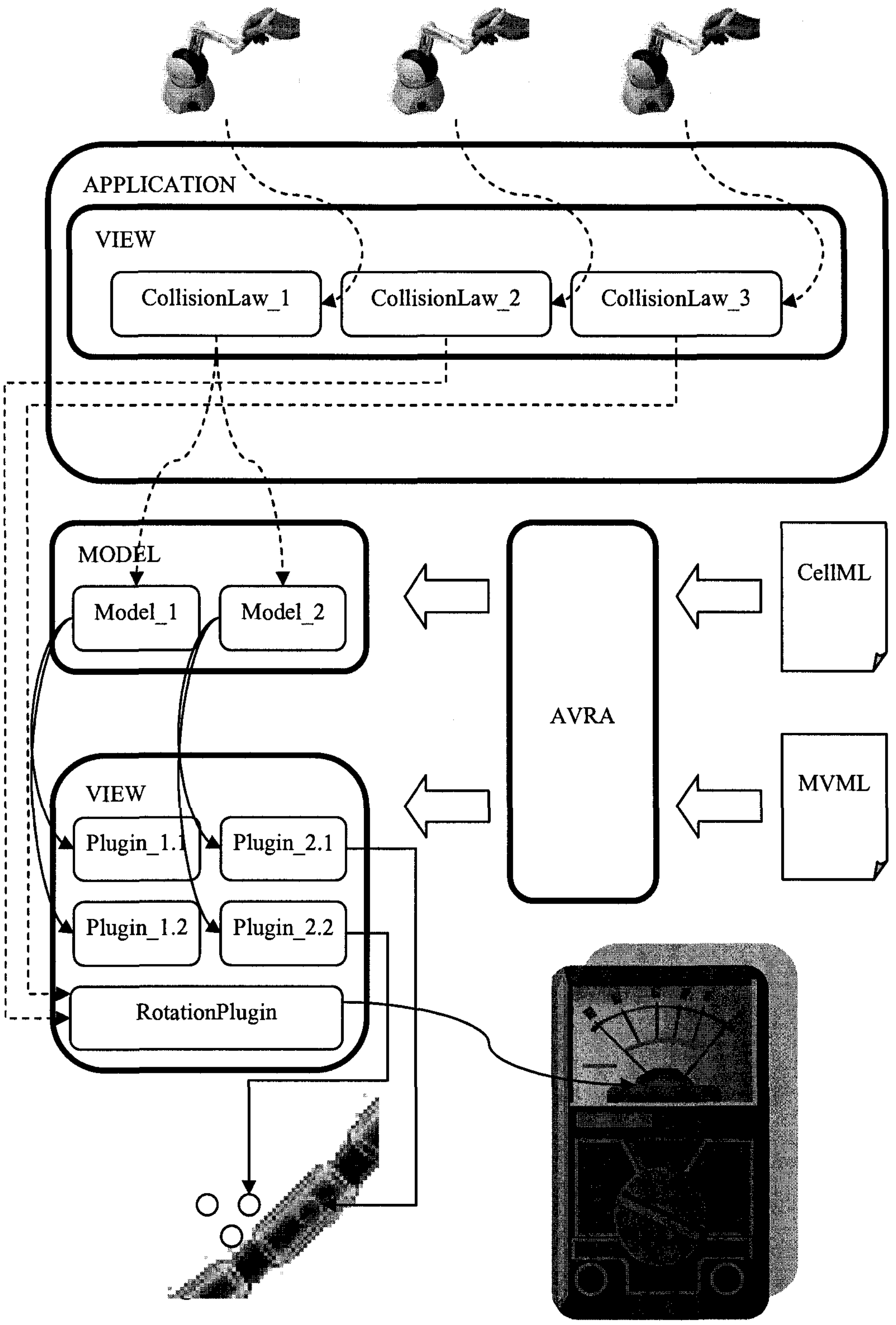


Figure 80 – The inter-communication between AVRA and high-level applications to account for interactions.

Figure 79 shows the MVML code for inserting a 3D multimeter in the virtual scene and creating a view plugin and rotate the voltmeter handle based on the voltage amount received from component *membrane_x* which should be changed at run-time to membrane_1...membrane_5 by the high-level application depending on which membrane the probe is colliding with.

Figure 80 demonstrates a possible infrastructure that enables the high-level application to implement the above scenario using the existing AVRA capabilities as used in the previous case study. In effect the application adds three additional view plugins for the purpose of detecting collisions between the probes that are controlled by haptic devices and the 3D model of a neuron. When CollisionLaw_1 detects a collision, it identifies the nearest axon chamber and directly manipulates the input value of the stimulus variable in its model_1 or model_2. CollisionLaw_2 and CollisionLaw_3 track the position of multimeter probes. They also detect the nearest axon chamber but in this case they control the inputs of the RotationPlugin based on that. Essentially they tell the RotationPlugin to base the rotation angle on V values receives from membrane_1 or membrane_2, whichever is closer.

In the above scenario, the haptic interaction can be implemented directly by the higher-level applications; or alternative it can be achieved with a 3rd party library such as HAML [54] that is configured on top of the AVAR architecture.

9.7 Results

9.7.1 Key Advantages

The case studies in this chapter outlined some of the key advantages that AVRA has when developing VR applications that visualize the conceptual outcomes of their target simulations. This section summarizes the key novelties and advantages that were verified by these case studies and then compares them with the existing frameworks.

The key advantages of AVRA as verified by the case studies are:

- **Model-View Mapping in Virtual Environments:** AVRA provides a comprehensive infrastructure that dynamically constructs numerical and graphical models and automatically connects them together based on the contents of the MVML descriptions.

- **Model Update:** The simulation models that are loaded in AVRA can be conveniently updated with updated mathematical specifications or numerical data without any changes in the view description or the application. The simulation models can also be replaced with other models with minor changes in the MVML description of the view if the component and variable names are different.

- **View Update:** After the first construction of MVML, its view description contents can be conveniently reconfigured with new parameters or updated with new graphics and view plugins without changes in the simulation model or the application.

- **Multi-Model Visualization:** AVRA supports visualization for multiple models including models that are different than each other and those that are duplicates of each other. Each model instance may affect the visual aspect of its dedicated graphics or contribute to a unified share graphics.

- **Multi-View Visualization:** In AVRA, a single simulation model can be associated with several graphics. This is useful when visualizing several aspects of a same simulation model. For example the visualization of a neuron cell can include both graphics of the neuron and a multimeter than measures its voltage.

- **Dynamic Model-View Mapping for High-Level Applications:** AVRA allows applications that are built on top of it to access the internal model and view components that are automatically generated upon loading MVML files. In particular AVRA allows these applications to dynamically reconfigure the model-view mapping. This allows more complex visualization including those that are affected by the user interactions.

Table 5 outlines the case-study coverage of the above 6 features. As it was shown the previous sections, the presented case studies reused much of elements from the previous case studies hence demonstrating the benefits of using AVRA. For example the case study 3 simply updates the view description of case study 2 hence demonstrating that the view can be updated in AVRA without affecting the model or the application.

	Case Study 1	Case Study 2	Case Study 3	Case Study 4	Case Study 5
Model-View Mapping	X	X	X	X	X
Model Update		X		X	
View Update	X	X	X		X
Multi-Model		X		X	
Multi-View					X
Dynamic Mapping					X

Table 5 – The case-study feature coverage matrix

9.7.2 Comparisons with Other Frameworks

In section 2.3 the most popular frameworks for VR applications were discussed. When considering how the case studies of this chapter can be implemented in these frameworks it is apparent that the heavier burden of the development is on the shoulder of the higher-level applications. In essence these frameworks do not contribute directly to the visualization of simulation models. Instead they provide a rich set of libraries that offer a wide variety of services that while valuable in other aspects of VR application, do not make the task of simulation development any easier.

Although VirtualExplorer is explicitly designed for simulation and visualization purposes its approach to this issue is not fundamental. VirtualExplorer simply offers utilities and device interfaces through plugins and allows the simulations to be implemented within applications plugins. These application plugins typically contain all of the high level infrastructure and the low-level implementations for simulation calculation and visualization.

The only existing framework that provides an infrastructure that can be of use in implementing simulations is SCIVE [6]. Similarly to AVRA, SCIVE uses a mapping schema that connects models, graphics, animation, physics and possibly many other components together. In SCIVE this is done through a central knowledge layer which contains all the data that is needed by the components. While this design allows reuse and sharing of data it is merely a managed data sharing infrastructure and leaves the applications with the low level tasks of populating the shared data and mapping them to high-level behaviors such graphical behaviors and visualizations.

CHAPTER 10 - CONCLUSION

As the software technology in any field advances and overcomes its primitive challenges, it is desirable to utilize highly reusable and flexible architectures and frameworks as to avoid redundancies and maximize flexibility and reusability. We have seen this happening in the IT field with the introduction of enterprise technologies such as XML, J2EE, .NET, Ruby on the Rails and others. While all these technologies are different nature, they share a same goal of providing developers with rich frameworks and libraries that support dynamic change in data and view with minimum code change requirements.

The field of VR has fallen behind in this respect and the proposed architecture is one that utilizes the lessons learned and applied in IT sector to address the very same problems that VR is facing today.

This thesis presents a novel framework that applies the most fundamental concepts of software engineering in order to decouple the model and view components and take advantage of the resulting flexibility, extensibility and reusability. AVRA is capable of accepting many types of simulation models and generating virtual environments that effectively visualize those simulations. The scene construction is based on a flexible model-view scheme that uses MVML for describing how the targeted simulation is to be visualized. With MVML description, the outputs of the simulation models can be connected to inputs of view plugins that ultimately generate animations and dynamic behaviors in a virtual environment. Since these animations are directly the result of precise mathematical calculations or data repository, the generated visual effects can sever as a convenient mean of observing simulation result by scientists and engineers. In addition, it can be used as the basis of more complex VR applications including those that require interactions.

From a higher point of view, the proposed architecture allows development of VR-based simulations that allow frequent and dynamic changes in their simulation *model* and view. This is of great significance as the aforementioned two are the most common elements that are subject to frequent changes during the lifetime of the VR-based simulations. The reason for changes in models is that the data and mathematics of the simulation models evolve as new experimental data arrives or the theoretical simulation environment changes. The changes in view is often linked to what is perceived as effective visualization of the simulation and it depends on (1) the aspect of interest, that is which part of the simulation is most important to observe, and (2) the abstract level of interest, that is how high-level or detailed the visualization must be.

AVRA is a framework that utilizes various techniques to allow the implementation of VR-based simulations that are highly flexible in term of their model, view and interaction contents. For this purpose AVRA introduce two novel concepts: *view plugin* and *MVML.*

AVRA depends on view plugins to address flexibility, reusability and extensibility of VR simulation. In AVRA, it is understood that the visual behavior of VR application should be customizable without recoding or even recompiling the project. It is also understood that there should be no limitations in implementing new visualization behaviours. As such view plugins are introduced to provide a unified structure for defining modules that contain compiled codes for specific visualization tasks to be added to a simulation application upon request.

While XML-based languages already exist for defining the underlying data and/or mathematics of the simulations, there are no standards for defining the view and interaction aspects of such applications. As such MVML, a novel XML-based specification, is proposed to address the aforementioned shortcomings. With MVML developers can specify which simulation model to visualize, which view plugins are to be used for the visualization and which parameters to pass to the various modules involved.

The combination of the above contributions ensures that AVRA architecture is one that overcomes the most significant problems that VR-based simulations are facing today.

In addition to its various benefits, AVRA presents challenging opportunities for future developments. In particular AVRA architecture can be expanded to cover the interaction aspects of simulation applications, therefore further reducing the weight of low-level responsibilities in the higher level applications. Another challenging future work will be to expand AVRA in order to step beyond simulation applications, covering other VR-based applications such as those in gaming, training and artificial intelligence.

APPENDICES

Appendix A

A complete CellML File

```
<model name="basic_ep_model" xmlns="http://www.cellml.org/cellml/1.0#"
       xmlns:cellml="http://www.cellml.org/cellml/1.0#"
       xmlns:cmeta="http://www.cellml.org/metadata/1.0#">

  <units name="concentration_units">
    <unit prefix="milli" units="mole" />
    <unit units="litre" exponent="-1" />
  </units>

  <units name="flux_units">
    <unit units="concentration_units" />
    <unit units="second" exponent="-1" />
  </units>

  <units name="rate_constant">
    <unit units="second" exponent="-1" />
  </units>

  <component name="environment">
    <variable name="time" public_interface="out" units="second" />
  </component>

  <component name="intra_cellular_space">
    <!-- the following variables are used in other components -->
    <variable name="Na" public_interface="out" units="concentration_units" />
    <variable name="Ca" public_interface="out" units="concentration_units" />

    <!-- the following variables are imported from other components -->
    <variable name="time" public_interface="in" units="second" />
    <variable name="I_Na" public_interface="in" units="flux_units" />
    <variable name="I_Ca" public_interface="in" units="flux_units" />

    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply><eq />
        <apply><diff />
          <bvar><ci> time </ci></bvar>
          <ci> Na </ci>
        </apply>
        <ci> I_Na </ci>
      </apply>

      <apply><eq />
        <apply><diff />
          <bvar><ci> time </ci></bvar>
          <ci> Ca </ci>
        </apply>
        <ci> I_Ca </ci>
      </apply>
    </math>
  </component>

[CONTINUED NEXT PAGE...]
```

```
  <component name="extra_cellular_space">
    <!-- the following variables are used in other components -->
    <variable name="Na" public_interface="out" units="concentration_units" />
    <variable name="Ca" public_interface="out" units="concentration_units" />
    <!-- the following variables are imported from other components -->
    <variable name="time" public_interface="in" units="second" />
    <variable name="I_Na" public_interface="in" units="flux_units" />
    <variable name="I_Ca" public_interface="in" units="flux_units" />
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply><eq />
        <apply><diff />
          <bvar><ci> time </ci></bvar>
          <ci> Na </ci>
        </apply>
        <apply><times />
          <cn cellml:units="dimensionless"> -1.0 </cn>
          <ci> I_Na </ci>
        </apply>
      </apply>
      <apply><eq />
        <apply><diff />
          <bvar><ci> time </ci></bvar>
          <ci> Ca </ci>
        </apply>
        <apply><times />
          <cn cellml:units="dimensionless"> -1.0 </cn>
          <ci> I_Ca </ci>
        </apply>
      </apply>
    </math>
  </component>

  <component name="cell_membrane">
    <!-- the following variables are used in other components -->
    <variable name="I_Na" public_interface="out" units="flux_units" />
    <variable name="I_Ca" public_interface="out" units="flux_units" />
    <!-- the following variables are imported from other components -->
    <variable name="Na_i" public_interface="in" units="concentration_units" />
    <variable name="Na_e" public_interface="in" units="concentration_units" />
    <variable name="Ca_i" public_interface="in" units="concentration_units" />
    <variable name="Ca_e" public_interface="in" units="concentration_units" />
    <!-- the following variables are only used internally -->
    <variable name="v_Na" initial_value="1.0e-8" units="rate_constant" />
    <variable name="v_Ca" initial_value="1.5e-8" units="rate_constant" />
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <apply><eq />
        <ci> I_Na </ci>
        <apply><times />
          <ci> v_Na </ci>
          <apply><minus />
            <ci> Na_i </ci>
            <ci> Na_e </ci>
          </apply>
        </apply>
      </apply>
      <apply><eq />
        <ci> I_Ca </ci>
        <apply><times />
          <ci> v_Ca </ci>
          <apply><minus />
            <ci> Ca_i </ci>
            <ci> Ca_e </ci>
          </apply>
        </apply>
      </apply>
    </math>
  </component>

[CONTINUED NEXT PAGE...]
```

```
  <connection>
    <map_components component_1="intra_cellular_space" component_2="cell_membrane" />
    <map_variables variable_1="Na" variable_2="Na_i" />
    <map_variables variable_1="Ca" variable_2="Ca_i" />
    <map_variables variable_1="I_Na" variable_2="I_Na" />
    <map_variables variable_1="I_Ca" variable_2="I_Ca" />
  </connection>


  <connection>
    <map_components component_1="extra_cellular_space" component_2="cell_membrane" />
    <map_variables variable_1="Na" variable_2="Na_e" />
    <map_variables variable_1="Ca" variable_2="Ca_e" />
    <map_variables variable_1="I_Na" variable_2="I_Na" />
    <map_variables variable_1="I_Ca" variable_2="I_Ca" />
  </connection>

  <connection>
    <map_components component_1="environment" component_2="intra_cellular_space" />
    <map_variables variable_1="time" variable_2="time" />
  </connection>


  <connection>
    <map_components component_1="environment" component_2="extra_cellular_space" />
    <map_variables variable_1="time" variable_2="time" />
  </connection>


</model>
```

Appendix B1

Numerical Output of a CellML Model

```
Model = basic_ep_model
Component = environment************************
Local:
Inputs:
Outputs:
time = 0.0
Component = extra_cellular_space************************
Local:
Ca = 40.0
Na = 30.0
Inputs:
I_Ca = -3.0E-7
time = 0.0
I_Na = -2.0E-7
Outputs:
Ca = 40.0
Na = 30.0
Component = intra_cellular_space************************
Local:
Ca = 20.0
Na = 10.0
Inputs:
I_Ca = -3.0E-7
time = 0.0
I_Na = -2.0E-7
Outputs:
Ca = 20.0
Na = 10.0
Component = cell_membrane************************
Local:
v_Na = 1.0E-8
v_Ca = 1.5E-8
Inputs:
Ca_e = 40.0
Na_e = 30.0
Na_i = 10.0
Ca_i = 20.0
Outputs:
I_Ca = -3.0E-7
I_Na = -2.0E-7
```

Appendix B2

```
Model = basic_ep_model
Component = environment************************
Local:
Inputs:
Outputs:
time = 1000.0
Component = extra_cellular_space************************
Local:
Ca = 40.0
Na = 30.0
Inputs:
I_Ca = -3.000090000027E-7
time = 1000.0
I_Na = -2.000040000008E-7
Outputs:
Ca = 40.00030000009
Na = 30.00020000004
Component = intra_cellular_space************************
Local:
Ca = 20.0
Na = 10.0
Inputs:
I_Ca = -3.000090000027E-7
time = 1000.0
I_Na = -2.000040000008E-7
Outputs:
Ca = 19.99969999991
Na = 9.99979999996
Component = cell_membrane************************
Local:
v_Na = 1.0E-8
v_Ca = 1.5E-8
Inputs:
Ca_e = 40.00030000009
Na_e = 30.00020000004
Na_i = 9.99979999996
Ca_i = 19.99969999991
Outputs:
I_Ca = -3.000090000027E-7
I_Na = -2.000040000008E-7
```

Appendix B3

```
Model =
basic_ep_mode  <import xlink:href="httpsnt************************
Local:
Inputs:
Outputs:
time = 60000.0
Component = extra_cellular_space************************
Local:
Ca = 40.0
Na = 30.0
Inputs:
I_Ca = -3.005400162E-7
time = 60000.0
I_Na = -2.0024000479999997E-7
Outputs:
Ca = 40.01800054
Na = 30.01200024
Component = intra_cellular_space************************
Local:
Ca = 20.0
Na = 10.0
Inputs:
I_Ca = -3.005400162E-7
time = 60000.0
I_Na = -2.0024000479999997E-7
Outputs:
Ca = 19.98199946
Na = 9.98799976
Component = cell_membrane************************
Local:
v_Na = 1.0E-8
v_Ca = 1.5E-8
Inputs:
Ca_e = 40.01800054
Na_e = 30.01200024
Na_i = 9.98799976
Ca_i = 19.98199946
Outputs:
I_Ca = -3.005400162E-7
I_Na = -2.0024000479999997E-7
```

Appendix C

A complete MVML File

```
<view model_name="hodgkin_huxley_1952_version05">
       <graphics name="cell" file=".\\models\\Cell.x3d" visible="true"
              position="0 0 0" rotation="0 0 1 0" />
       <graphics name="Na" file=".\\models\\Na.x3d" visible="false" />
       <graphics name="K" file=".\\models\\K.x3d" visible="false" />
       <function name="negate" >
              <math xmlns='http://www.w3.org/1998/Math/MathML'>
               <apply>
                <times/>
                <cn type='real'>-1.</cn>
                <ci>x</ci>
               </apply>
              </math>
       </function>
       <function name="add" >
              <math xmlns='http://www.w3.org/1998/Math/MathML'>
               <apply>
                <plus/>
                <ci>x</ci>
                <cn type='real'>50.</cn>
               </apply>
              </math>
       </function>
       <function name="sub" >
              <math xmlns='http://www.w3.org/1998/Math/MathML'>
               <apply>
                <plus/>
                <cn type='real'>300.</cn>
                <apply>
                 <times/>
                 <cn type='real'>-1</cn>
                 <ci>x</ci>
                </apply>
               </apply>
              </math>
       </function>

       <plugin class="xpheve.laws.RotationLaw">
              <attribute name="TransformGroup">
                     <use graphics="cell" part="GateM" />
              </attribute>
              <attribute name="value">
                     <valueof component="sodium_channel_m_gate" variable="m" />
              </attribute>
       </plugin>
       <plugin class="xpheve.laws.RotationLaw">
              <attribute name="MinValue"> 0 </attribute>
              <attribute name="MaxValue"> 0.6 </attribute>
              <attribute name="TransformGroup">
                     <use graphics="cell" part="GateH" />
              </attribute>
              <attribute name="value">
                     <valueof component="sodium_channel_h_gate" variable="h" />
              </attribute>
       </plugin>
```

[CONTINUED NEXT PAGE...]

```
        <plugin class="xpheve.laws.RotationLaw">
                <attribute name="MinValue"> 0.3 </attribute>
                <attribute name="MaxValue"> 0.7 </attribute>
                <attribute name="TransformGroup">
                        <use graphics="cell" part="GateN" />
                </attribute>
                <attribute name="value">
                        <valueof component="potassium_channel_n_gate" variable="n" />
                </attribute>
        </plugin >
        <plugin class="xpheve.laws.ColorLaw">
                <attribute name="Shape">
                        <use graphics="cell" part="CellBox" />
                </attribute>
                <attribute name="value">
                        <valueof component="membrane" variable="V" />
                </attribute>
                <attribute name="MaxColor">1 1 0</attribute>
        </plugin >
        <plugin class="xpheve.laws.AnimatedParticleLaw">
                <attribute name="NumOfSourceParticles"> 150 </attribute>
                <attribute name="NumOfDestinationParticles"> 0 </attribute>
                <attribute name="Scale"> 0.001 </attribute>
                <attribute name="SourceSpaceMin"> -1.4 1.6 -0.9 </attribute>
                <attribute name="SourceSpaceMax"> 1.4 2.3 0.9 </attribute>
                <attribute name="DestinationSpaceMin"> -1.4 -0.6 -0.6 </attribute>
                <attribute name="DestinationSpaceMax"> 1.4 0.4 0.6 </attribute>
                <attribute name="ParticleModel"> <use graphics="Na" /> </attribute>
                <attribute name="rate">
                        <valueof formula="negate" component="sodium_channel"
                                variable="i_Na" />
                </attribute>
                <attribute name="IntermediatePoint"> -1 1 0 </attribute>
        </plugin >
        <plugin class="xpheve.laws.SharedAnimatedParticleLaw">
                <attribute name="NumOfSourceParticles"> 100 </attribute>
                <attribute name="NumOfDestinationParticles"> 0 </attribute>
                <attribute name="Scale"> 0.001 </attribute>
                <attribute name="SourceSpaceMin"> -1.4 -0.6 -0.6 </attribute>
                <attribute name="SourceSpaceMax"> 1.4 0.4 0.6 </attribute>
                <attribute name="DestinationSpaceMin"> -1.4 1.6 -0.9 </attribute>
                <attribute name="DestinationSpaceMax"> 1.4 2.3 0.9 </attribute>
                <attribute name="ParticleModel">
                        <use graphics="K" />
                </attribute>
                <attribute name="rate">
                        <valueof component="potassium_channel" variable="i_K" />
                </attribute>
                <attribute name="IntermediatePoint"> 0 1 0 </attribute>
        </plugin >
        <plugin name class="xpheve.laws.SharedAnimatedParticleLaw">
                <attribute name="NumOfSourceParticles"> 100 </attribute>
                <attribute name="NumOfDestinationParticles"> 0 </attribute>
                <attribute name="Scale"> 0.001 </attribute>
                <attribute name="SourceSpaceMin"> -1.4 -0.6 -0.6 </attribute>
                <attribute name="SourceSpaceMax"> 1.4 0.4 0.6 </attribute>
                <attribute name="DestinationSpaceMin"> -1.4 1.6 -0.9 </attribute>
                <attribute name="DestinationSpaceMax"> 1.4 2.3 0.9 </attribute>
                <attribute name="ParticleModel">
                        <use graphics="K" />
                </attribute>
                <attribute name="rate">
                        <valueof component="leakage_current" variable="i_L" />
                </attribute>
                <attribute name="IntermediatePoint"> 1 1 0 </attribute>
        </plugin >
</view>
```

Appendix D

The DTD of MVML

```
<?xml version="1.0"?>
<!DOCTYPE view [

<!ELEMENT view         (graphics*, plugin*, function*)>
<!ATTLIST view         model_name #PCDATA #REQUIRED>

<!ELEMENT graphics EMPTY>
<!ATTLIST graphics     name #PCDATA #REQUIRED>
<!ATTLIST graphics     file #PCDATA #REQUIRED>
<!ATTLIST graphics     visible (true | false) #IMPLIED>
<!ATTLIST graphics     position #PCDATA #IMPLIED>
<!ATTLIST graphics     rotation #PCDATA #IMPLIED>

<!ELEMENT plugin       (attributes)*>
<!ATTLIST plugin       class #PCDATA #REQUIRED>
<!ELEMENT attributes   (#PCDATA| valueof | use)>
<!ATTLIST attributes   name #PCDATA #REQUIRED>
<!ELEMENT valueof EMPTY>
<!ATTLIST valueof      component #PCDATA #REQUIRED>
<!ATTLIST valueof      variable #PCDATA #REQUIRED>
<!ELEMENT use EMPTY>
<!ATTLIST use          graphics #PCDATA #REQUIRED>
<!ATTLIST use          part #PCDATA #IMPLIED>

<!ELEMENT function     (math)>
<!ATTLIST function     name #PCDATA #REQUIRED>
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
          http://www.w3.org/Math/DTD/mathml2/mathml2.dtd >
]>
```

REFERENCES

[1] G.C. Burdea and P. Coiffet – "Virtual Reality Technology", Wiley-IEEE Press; 2nd edition, 2003

[2] S. Nourian, X. Shen, N. D. Georganas, "xPheve: An Extensible Physics Engine for Virtual Environments", CCECE 2006

[3] P.J. Hunter, W.W. Li, A.D. McCulloch, D. Noble – "Multiscale Modeling: Physiome Project Standards, Tools, and Databases" – IEEE Computer (Vol. 39, No. 11), Pages: 48-54

[4] C.M. Lloyd, M.D.B. Halstead, P.F. Nielsen "CellML: its future, present and past", Progress in Biophysics and Molecular Biology, 2004

[5] G. Heumer, M. Schilling, M. E. Latoschik "Automatic Data Exchange and Synchronization for Knowledge-Based Intelligent Virtual Environments", 2005 IEEE VR

[6] M. E. Latoschik, C. Frohlich, "Semantic Reflection for Intelligent Virtual Environments", IEEE VR 2007

[7] X. Shen, J. Zhou, A. Hamam, S. Nourian, N. R. El-Far, F. Malric, N. D. Georganas – "Haptic-Enabled Telementoring Surgery Simulation" – IEEE Multimedia, 2008

[8] J. Kelso, L. E. Arsenault, S. G. Satterfield, R. D. Kriz – "DIVERSE: A Framework for Building Extensible and Reconfigurable Device Independent Virtual Environments" – IEEE VR 2002

[9] F. Kuester, B. Hamann, K.I. Joy – "VirtualExplorer: A Plugin-Based Virtual Reality Framework" – Proceedings of SPIE, 2001

[10] Y. Xiao, S. Regatti, S. Kutuva – "A Framework for Integrating Virtual Surgery Modules" – University of Akron, Proceedings of Health Informatics 2006 (part of IADIS Virtual Multi Conference on Computer Science and Information System, May 15 – 19, 2006).

[11]T. B. L. Kirkwood, R. J. Boys, CS Gillespie, C. J. Proctor, "Towards an e-biology of ageing: integrating theory and data", Science 2001

[12]C. Shaw, J. Liang, M. Green, Y. Sun – "The decoupled simulation model for virtual reality systems" – ACM SIGCHI 1992, Pages: 321 – 328

[13]J. M. Sobel, D. P. Friedman - "An Introduction to Reflection-Oriented Programming" – Indiana University, http://www.cs.indiana.edu/~jsobel/rop.html

[14]B. Buxton, G.W. Fitzmaurice – "HMDs, Caves & Chameleon: A Human-Centric Analysis of Interaction in Virtual Space" – Computer Graphics: The SIGGRAPH Quarterly, 32(4), 64-68, http://www.billbuxton.com/VRtaxonomy.html (as of 2007)

[15]R. J. Stone – "Haptic Feedback: A Brief History from Telepresence to Virtual Reality" – Robert J. Stone, http://www.springerlink.com/content/e2kr2eu6820tx1lg/fulltext.pdf (as of 2008)

[16]A. Ioannidis, M. Spanoudakis, P. Sianas, I. Priggouris, S. Hadjiefthymiades, L. Merakos – "Using XML and related standards to support Location Based Services" – 2004 ACM symposium on Applied computing, http://portal.acm.org/citation.cfm?id=968226 (as of 2008)

[17]B.W. Smith, S. Andreassen, G.M. Shaw, P.L. Jensen, S.E. Rees, J.G. Chase – "Simulation of cardiovascular system diseases by including the autonomic nervous system into a minimal model" – Computer Methods and Programs in Biomedicine, Volume 86 , Issue 2 (May 2007), Pages: 153-160, http://portal.acm.org/citation.cfm?id=1238539 (as of 2008)

[18]A.D. Au, H.S. Greenfield – "A computer graphics approach for understanding of prosthetic heart valve characteristics" – SIGGRAPH 1972 seminar on Computer graphics in medicine, Pages: 43 – 52, http://portal.acm.org/citation.cfm?id=804973 (as of 2008)

[19]CellML Wiki – "Example of CellML Electrophysiological Models" – http://www.cellml.org/tutorial/electrophysiological (as of 2008)

[20]U. Kuhnapfel, H. Cakmak , B. Chantier , H. MaaB, G. Strauss, C. Trantakis, E. Novatius, J. Meixensberger, K. Lehmann, H. J. Buhr, M. Lawo, G. Bretthauer "Haptic Interface-

Systems for Virtual-Reality Training in Minimally-Invasive Surgery", International Status Conference for Virtual and Augmented Reality, Leipzig, Germany, Feb. 2004

[21]Bray, Tim; Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau (September 2006). Extensible Markup Language (XML) 1.0 (Fourth Edition) - Origin and Goals. World Wide Web Consortium. Retrieved on October 29, 2006. http://www.w3.org/TR/xml/ (as of 2008)

[22]Wikipedia – "Extensible Markup Language (XML)" – http://en.wikipedia.org/wiki/XML (as of 2008)

[23]S. Buswell, S. Devitt, A. Diaz – "Mathematical Markup Language (MathML™) 1.01 Specification (Abstract)" – September, 2006 Revision.

[24]R. Fried, M.B. Preisack,W. Klas, T. Rose – "Virtual Reality and 3D Visualizations in Heart Surgery Education" - The Heart Surgery Forum #2001-03054, http://static.cjp.com/gems/pdfs/2001-03054.pdf (as of 2008)

[25]T.S. Sørensen, J. Mosegaard – "Haptic Feedback for the GPU-Based Surgical Simulator " - Medicine Meets Virtual Reality 14, IOS Press, 2006, http://www.daimi.au.dk/~mosegard/publications/MMVR06GPUHaptics.pdf (as of 2008)

[26]RTI International, Research Triangle Park, NC, http://www.rti.org/

[27]C. Shaw, J. Liang, M. Green, Y. Sun – "The decoupled simulation model for virtual reality systems" – ACM SIGCHI 1992, Pages: 321 – 328, http://portal.acm.org/citation.cfm?id=142750.142824 (as of 2008)

[28]E. Gamma, R. Helm, R. Johnson, J. M. Vlissides – "Design Patterns: Elements of Reusable Object-Oriented Software" – Book, Chapter 4, 1994

[29]D. Luckham – "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems" – Addison-Wesley, Book, Chapter 6, 2002

[30]D. Greer – "Interactive Application Architecture Patterns" – CTRL-SHIFT-B, http://ctrl-shift-b.blogspot.com/2007/08/interactive-application-architecture.html (as of 2008)

[31]B. Lewis, D. J. Berg – "Multithreaded Programming with Java Technology" – Sun Microsystems Press Java Series, Book, Chapter 6

[32]A. Hodgkin, A. Huxley – "A quantitative description of membrane current and its application to conduction and excitation in nerve" – J. Physiol. 117:500–544, 1952

[33]S. Means, M. Bodie – "The Book of SAX: The Simple API for XML" – Book, Chapters 2-7, 2002

[34]J. Marini – "Document Object Model : Processing Structured Documents" – McGraw-Hill/OsborneMedia; 1st edition, Book, Chapter 3, 2002

[35]S. Nourian, X. Shen, N. D. Georganas, "Role of Extensible Physics Engine in Surgery Simulations", HAVE 2005

[36]D Chang, NH Lovell, S Dokos, "Field Markup Language: Biological Field Representation in XML", Engineering in Medicine and Biology Society (EMBS) 2007

[37]H Tramberend – "Avocado: A Distributed Virtual Reality Framework" – Proceedings of the IEEE Virtual Reality, 1999

[38]Eduardo T. L. Corseuil, Alberto B. Raposo, Romano J. M. da Silva, Marcio H. G. Pinto, Gustavo N. Wagner, Marcelo Gattass – "ENVIRON–Visualization of CAD Models In a Virtual Reality Environment" – Eurographics Symposium on Virtual Environments, 2004

[39]KC Gross, CJ Digate, EH Lee, "Event-driven rule-based messaging system" – US Patent 5,283,856, 1994

[40]J.C. de Oliveira M. Hosseini, S. Shirmohammadi, F. Malric, S. Nourian A. El Saddik N.D. Georganas – "Java multimedia telecollaboration" – IEEE Multimedia, 2003

[41]HAVOK – HAVOK Physics – http://www.havok.com/ (as of 2008)

[42]AGEIA – AGEIA PhysX, NovodeX – http://www.ageia.com/ (as of 2008)

[43]R. Shrout – "NVIDIA and Havok Bring SLI Physics to Life" - http://www.pcper.com/article.php?aid=222 (as of 2008)

[44]M. Latoschik, C. Frohlich – "TOWARDS INTELLIGENT VR: Multi-Layered Semantic Reflection for Intelligent Virtual Environments" – IEEE VR 2007

[45]J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. – "Flowvr: a middleware for large scale virtual reality applications" – In Proceedings of Euro-par 2004, Pisa, Italia, August 2004.

[46]W. Bethel, C. Bass, S. R. Clay, B. Hook, M. T. Jones, H. Sowizral, and A. van Dam. – "Scene graph apis: wired or tired?" – ACM SIGGRAPH 99 Conference abstracts and applications, pages 136–138, New York, NY, USA, 1999. ACM Press.

[47]R. Blach, J. Landauer, A. Rsch, and A. Simon. – "A Highly Flexible Virtual Reality System. In Future Generation Computer Systems" – Special Issue on Virtual Environments. Elsevier Amsterdam, 1998.

[48]S. Hansen, T. V. Fossum – "Refactoring model-view-controller" – Journal of Computing Sciences in Colleges, Volume 21 Issue 1, 2005

[49]H. Delingette and N. Ayache – "Hepatic surgery simulation" – ACM Communication 2005, Volume 48, Number 2, Pages 31 – 36

[50]W. Chen, H. Wan, H. Zhang, H. Bao, Q. Peng – "Interactive collision detection for complex and deformable models using programmable graphics hardware" – Proceedings of the ACM symposium on Virtual reality software and technology

[51]Y. Tamura, A. Kageyama, H. Nakamura, N. Mizuguchi and T. Sato – "Collaborative virtual reality space for analyzing numerical simulation results" – Journal of Plasma Physics, 2006

[52]S. Burbeck – "Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)" – 1987 Smalltalk, http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html (as of 2008)

[53]B Kosko – "Neural Networks and Fuzzy Systems" – Book, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[54]M. Eid, A. Alamri, A. E. Saddik – "MPEG-7 Description of Haptic Applications Using HAML" – Haptic Audio Visual Environments and their Applications, 2006. HAVE 2006

[55]A. Bierbaum, C. Just – “Software Tools for Virtual Reality Application Development” – SIGGRAPH, 1998