XML-BASED FRAMEWORK FOR WEB-BASED NEUROCARDIOVASCULAR SIMULATION

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND
ELECTRONICS ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BİLKENT UNIVERSITY
IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
İsmail UZUN
August 2004

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Y. Ziya İder (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ömer Morgül

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. İbrahim Körpeoğlu

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet Baray
Director of the Institute

ABSTRACT

XML-BASED FRAMEWORK FOR WEB-BASED NEUROCARDIOVASCULAR SIMULATION

İsmail UZUN

M.S. in Electrical and Electronics Engineering

Supervisor: Prof. Dr. Yusuf Ziya İDER

August 2004

Mathematical modeling and numerical simulation of neurocardiovascular control system has played an important role in better understanding of its function and diagnosis of neurological disorders. Current simulations of neurocardiovascular models are carried out using desktop applications, which lack remote access and information sharing facilities. Although, web-technology has penetrated into all areas of research and professional life during the past two decades, opportunities provided by the web technology has not been fully exploited in this area. Moving from desktop to web, utilizing web technology, promises global access, platform independence, information sharing and easy maintainability features. Considering these features, the demand on a framework that enables web-based simulation of neurocardiovascular system models becomes more obvious.

In this thesis, we have proposed and implemented an XML-based framework that enables web-based simulation of neurocardivascular models. In this context, we implemented an XML-based description language for structured description of neurocardiovascular models, a Java-based simulaton package and supportive software to form a web-based architecture. XML is becoming the universal standard for exchange of structured data over the web. Therefore, we make use of XML to propose the generic description language NeuroCardioVascular Markup Language (NCVML), such that it supports description of a wide range of model set. We expect neurocardiovascular model descriptions to be encoded in NCVML form and to be carried over the web in this format. The java-based simulation package, NCVJSim, contains a built-in library with peculiar components and a simulator part. The library could be extended in time such that the library evolves in time. Additionally, making use of Java Dynamic Class Loading & Java Reflection Mechanisms, we implemented the feature of incorporating user implemented Java classes during run-time. Finally, to achieve web-based access and computing, Java Servlet Technology and HTML are utilized.

Our proposed framework is developed to serve all types of models, thus, it is not restricted to a particular mathematical neurocardiovascular model.

Keywords: Web-based simulation, XML, description language Neurocardiovascular system, Neurocardiovascular models.

ÖZET

WEB-TABANLI NEUROCARDIOVASCULAR SİMÜLASYONU İÇİN XML-TABANLI BİR ÇERÇEVE

İsmail UZUN
Electrik-Elektronik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Yusuf Ziya İDER
Ağustos 2004

Neurocardivascular kontrol sisteminin matematiksel modellemesi ve numerik simulasyonu, sistemin işleyişinin daha iyi anlaşılması ve sinirsel bozuklukların teşhis edilmesinde önemli bir rol oynamaktadır. Şu anda neurocardiovascular modellerin simulasyonu, uzaktan erişim ve bilgi paylaşımı gibi özelliklerden yoksun masaüstü uygulamaları ile gerçekleştirilmektedir. Son 20 yılda, web teknolojisinin profesyonel yaşam ve araştırma hayatının içine yerleşmesine rağmen, bu alanda web teknolojilerinin sunduğu olanaklardan yeterince yararlanılmamaktadır. Web teknolojisi kullanılarak, masaüstünden web'e geçiş, beraberinde global erişim, platform bağımsızlığı, bilgi paylaşımı ve kolay bakım gibi özellikler vaad etmektedir. Bu özellikler gözönüne alındığında, web-tabanlı simulasyona olanak sağlayan bir ortama duyulan ihtiyaç daha açık görülmektedir.

Bu tezde, neurocardiovascular modellerin web-tabanlı simulasyonuna olanak sağlayan XML-tabanlı bir ortam sunulmuş ve gerçekleştirilmiştir. Bu bağlamda, neurocardiovascular modellerin yapısal olarak tanımlanmasını sağlayan bir XML tanımlama dili, Java tabanlı bir simulasyon paketi ve web

mimarisi oluşrumak üzere destekleyici uygulamalar geliştirilmiştir. Günümüzde, XML web üzerinden yapısal veri taşınmasında evrensel standart halini almaktadır. Bu nedenle, neurocardiovascular model kümesinin olabildiğince geniş bir kümesindeki modellerin tanımlanabilmesine olanak sağlayabilecek, genel bir tanımlama dili olan NeuroCardioVascular Modelleme Dili (NCVML)'nin geliştirilmesinde XML'den yararlanıldı. Neurocardiovascular model tanımlarının NCVML ile kodlanmaları ve internet üzerinde bu formatta taşınmaları beklenmektedir. Java tabanlı simulasyon paketi NCVJSim, kendine has bileşenler içeren bir kütüphane ile bir simulatör'den oluşmaktadır. Kütüphane, zamanla gelişebilecek şekilde genişleyebilir bir yapıdadır. Buna ek olarak, Java Dinamik Bağlama ve Java Reflection Mekanizmalari'ndan yararlanılarak, işletim aşamasında kullanıcı tarafından yazılmış Java sınıflarının işletime dahil edilmesi imkanı kazandırılmıştır. Son olarak, web tabanlı erişim ve web tabanlı işlem sağlamak üzere, Java Servlet teknolojisi ile HTML'den yararlanılmıştır.

Sunulan bu çerçeve, tüm modellerin simulasyonuna yönelik geliştirilmiş olup, belirli bir modelin gerçekleştirimi ile sınırlı değildir.

Anahtar sözcükler: Web tabanlı simulasyon, XML, tanımlama dili Neurocardiovascular sistem, Neurocardiovascular model.

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest thanks and gratitude to my advisor, Prof. Dr. Yusuf Ziya İDER for his supervision, his support and for his valuable ideas.

I would also like to thank to my manager Dr. Mesut GÖKTEPE for his guidance, and support to continue my graduate education.

Finally, I would like to thank to Prof. Dr. Ömer Morgül and Assoc. Prof. Dr. İbrahim Körpeoğlu for their valuble ideas and for sharing time to read and comment on this thesis.

To my brother Yusuf,

To Hilvi,

Contents

List of Figures

List of Tables

List of Abbreviations

ABP	-	Arterial Blood Pressure
Ach	-	Acetylcholine
ANS	-	Autonomic Nervous System
API	-	Application Programming Interface
CCC	-	Cardiovascular Control Center
CGI	–	Common Gateway Interface
CV	-	Cardiovascular
CVS	-	Cardiovascular System (CVS)
DOM	-	Document Object Model
DTD	-	Data Type Definition
GUI	-	Graphical User Interface
HTML	–	HyperText Markup Language
HTTP	-	Hypertext Transfer Protocol
JSP	-	Java Server Pages
JVM	-	Java Virtual Machine
NCV	-	Neurocardiovascular
NCVJSim	-	Neurocardiovascular System Java Simulator
NCVML	-	NeuroCardioVascular Markup Language
NCVMLDOMParser	–	NCVML Document Object Model Parser
NCVS	-	Neurocardiovascular System
NE	-	Norepinephrine
OO	–	Object Oriented
OOAD	-	Unification of Object Oriented Analysis & Design
Pas	–	Systemic Arterial Pressure

Pav	-	Average Pressure Value
SA node	-	Sinoatrial node
UML	-	Unified Modelling Language
W3C	-	World Wide Web Consortium
WWW	-	World Wide Web
XML	–	eXtensible Markup Language

Chapter 1

Introduction

1.1 Motivation

Mathematical modeling and numerical simulation of physiological systems have been extensively used for understanding their underlying mechanisms and for developing methods for manipulating them. In particular, modeling and simulation of the neurocardiovascular control system have paved the way for important findings regarding better comprehension of the system, development of better diagnostic and monitoring methods, as well as for the formulation of procedures that have helped to cure or remove cardiovascular and neurological disorders [1, 2, 3, 4]. Such studies have in the past evolved as separate endeavors of individual researchers and laboratories. Sharing of information has traditionally been through publications and conferences. However, opportunities provided by web technology for more effective collaboration and information sharing has not been fully exploited. As web technology penetrates into all areas of research and professional life, the demand on a framework that enables web-based simulation of neurocardiovascular system

models becomes more obvious. Such a framework should satisfy the following requirements:

- It should provide access to state-of-art knowledge regarding models of the neurocardiovascular control system. Specifically, it should include a built-in library for well-known models and provide a wealth of components and blocks peculiar to the system under consideration.
- The built-in library should be extendible. It should provide a mechanism for users to introduce to the framework their own components and blocks so that the web based framework evolves in time.
- The users should be able to construct and simulate their own models using built-in-functions as well as by the addition of their own blocks and components.
- Introduction of new components and blocks must be guided through a standardized procedure that uses a common language.
- The whole framework should be web based.

We envisage that such a framework should comprise the following components: 1) A comprehensive XML-based description language that lets users describe their models in an understandable way, 2) A simulation package to compute numerical results, and 3) Supporting software components to perform communication between client-side environment and simulation package, and user interfaces (i.e. GUI) to be able to perform these operations over the web utilizing WWW infrastructure.

XML-Based Description Language: Users must define and represent their models before they submit them to the system for simulation. The description language to be developed should allow for different models to be represented,

should be simple and easy to understand even for users who do not have a strong programming background, and should be extensible for future enhancements and must support transfer of data over the web. With the guidance of those requirements given above, we have defined an XML-based Description Language for Neurocardiovascular Models, which we have named as NeuroCardioVascular Markup Language (NCVML) to let users describe their neurocardiovascular models.

XML is the dominating modelling and data exchange standard on the web at present. Its easiness and extendible features make XML an adequate technology for development of a neurocardiac model description language.

Simulation Package: After detailed analysis of neurocardiovascular models, components of models can be grouped into two types: circuit components and blocks. Circuit components which are frequently used in neurocardiovascular models are flow resistance, pressure source, current source, heart valves, compliance and inertance. These circuit components, in general, have non-linear and time dependent properties. Blocks, on the other hand, are characterized by their inputs, outputs, and transfer functions. Again, for the neurocardiovascular system, nonlinear and time variant blocks are common. The simulation package must be capable of simulating models including both circuit components and systems at the same time, and thus must be a "hybrid" simulator. Some current popular simulation tools do not satisfy this requirement. One of the most popular simulation tools, "Simulink", is capable of simulating models composed of blocks. Thus, with Simulink each component must be provided as block with its transfer function. This constitutes a big shortage for our simulation methodology. Thus such simulators are not capable of simulating hybrid models. Besides, Simulink doesn't have Java interfaces that we can call from our framework. Another

well-known circuit simulator, “PSpice” is capable of simulating hybrid models but PSpice doesn’t have proper Java interfaces to let our framework to incorporate its functionalities. There are also hybrid simulators which have been developed for specific systems. One example is the product of DesignSoft, Tina Pro [5] which aims at simulating power distribution systems. In such systems the main plant is modelled as an equivalent circuit whereas the interaction of various circuit components and sources are modeled using transfer function blocks. The built-in library and the workings of Tina Pro are of-course designed to meet the specific requirements of power system simulations. For the above mentioned reasons we developed our own simulator instead of utilizing any package.

In this study a hybrid simulation environment is developed for the neurocardiovascular system. Java Programming Language is used in order achieve platform independence which is essential for any web based operation. Furthermore definition of library functions and methods are realized through “Object Oriented Programming” techniques.

Supporting Software: To enable pervasive access to the framework, client side environment should be a simple implementation running on a browser. It should enable users to upload models as well as initiate simulations. To provide two way communication between client-side environment and server-side simulation package, web server and as an extension Java based Servlet is implemented and included in the system.

1.2 Organization of the Thesis

This thesis is organized as follows: In Chapter 2, models of the NCV system are reviewed and the peculiar features of this system, which must be taken into consideration, are identified. In particular, the components, blocks and other features, which must be included in the NCV simulation framework, are listed. In Chapter 3, we describe NCVML, the XML description language that we have defined for describing neurocardiovascular models. In this section the built-in-library for the components and blocks most widely used in neurocardiovascular modeling and simulation are also introduced. Chapter 4 introduces the simulation package that we have developed with Java to perform numerical simulations (NCVJSim package). Chapter 5 provides the detailed description of the software architecture of the overall framework. We then present and explain an application example in Chapter 6, to show the feasibility of the system and working principles of the framework. We conclude our paper in Chapter 7.

Chapter 2

Review of Neurocardiovascular Models

2.1 Overview

Neural mechanisms play an important role in the regulation of arterial blood pressure, blood volume and other aspects of cardiovascular function. Neurocardiovascular system describes how the brain affects the activity of the cardiovascular system to regulate its function. Modeling and simulation of the neurocardiovascular control system have paved the way for important findings regarding better comprehension of the system, development of better diagnostic and monitoring methods, as well as for the formulation of procedures that have helped to cure or remove cardiovascular and neurological disorders [1, 2, 3, 4].

A complete neurocardiovascular model consists of the mechanical cardiovascular plant and the regulatory baroreflex mechanism that modulates (controls) hemodynamic variables of cardiovascular system to regulate its function. Cardiovascular system and cardiovascular system models, baroreflex mechanisms and baroreflex models will be described in the following two sections respectively. After these two sections the discussion will go on by merging cardiovascular models and baroreflex models to describe a complete neurocardiovascular model. The main pupose of this review of neurocardiovascular models is to determine and identify components and blocks of neurocardiovascular model simulation. These components and blocks will be the bases on which the XML-based simulation platform will be constructed.

2.2 Models of the Cardiovascular Plant

Cardiovascular system (or circulatory system) is responsible for distributing blood with oxygen and many other substances that are vital for the human body. The cardiovascular system consists of three main parts: the heart, systemic circulation and the pulmonary circulation. The task of distributing blood to every part of the body is performed through circulation of blood through blood vessels. Heart is responsible for pumping blood into vessels of systemic and pulmonary circulations. Systemic circulation is responsible for distribution of the oxygen-rich blood to all parts of the body and collection of the carbon-dioxide rich blood back to the heart. Pulmonary circulation, on the other hand, is the process of carrying carbon-dioxide rich blood to the lungs and transferring the oxygen-rich blood from the lungs back to the heart.

Several types of models have been proposed for mechanical

cardiovascular modeling: one type of models simulates the response of a distributed vascular system model to beat-to-beat pumping of blood by the heart. Another very well known type of models simulates vascular system using lumped components. The pioneer model for the latter type is the 2-element Windkessel model, which is the simplest possible model, but still retains important features of the system. In lumped parameter modeling of the cardiovascular system, the human cardiovascular system is divided into several compartments for modeling and simulation purposes. In general the circulatory system is represented by four compartments: the left and the right ventricular pumps, and vascular segments of the pulmonary and systemic circuits connected with heart chambers in series. The number of lumped elements in each compartment differs from model to model, although the basic principles are the same.

The pulmonary circuit consists of the pulmonary artery, major arteries, arterioles, capillaries and venules, major veins and pulmonary vein. The systemic circuit is compartmentalized into aorta, major arteries, arterioles, capillaries, venules and the vena cava. Models differ in their level of simplifications. For example some simplified models, like Guyton's model proposed in 1980, contain only one combined arterial chamber and one venous chamber in both parts of circulation. In the following part, a simple model of the CVS will be presented and pertinent variables and quantities will be defined.

2.2.1 Cardiovascular Model Example

In Figure 2.1, the circuit representation of a cardiovascular model is given [6]. In this model, the cardiovascular system is composed of the following four compartments: left ventricular, right ventricular, systemic circulation and

pulmonary circulation.

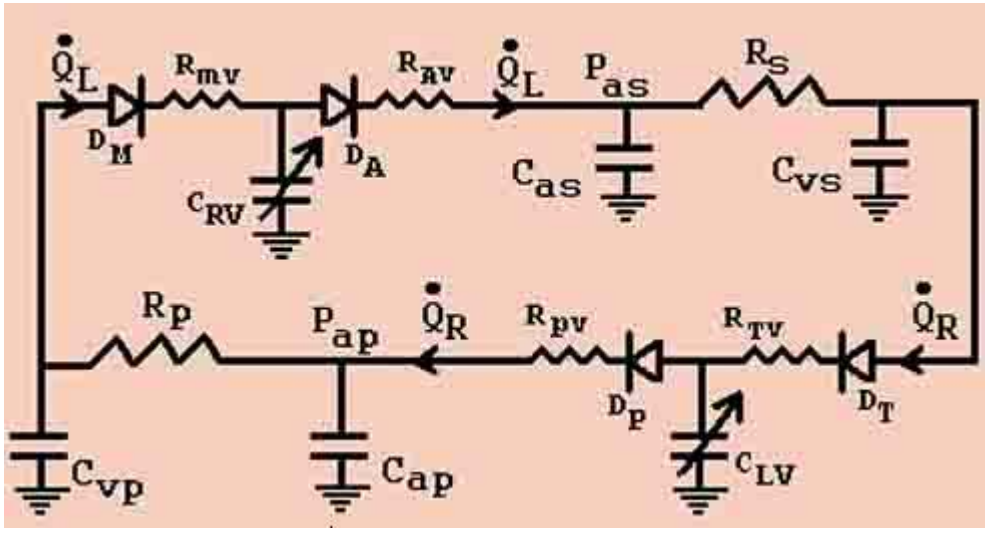


Figure 2-1: A cardiovascular system model of moderate complexity

The system parameters of this model are as follows (inputs);

T_H : Heart Beat Period (sec)
T_S : Systolic Period (sec)

C_{LV} : Compliance of the left ventricle (cc/mmHg)
C_{RV} : Compliance of the right ventricle (cc/mmHg)

R_S : Systemic resistance (mmHg.sec/cc)
R_P : Pulmonary resistance (mmHg.sec/cc)

C_{AS} : Systemic arterial compliance (cc/mmHg)
C_{VS} : Systemic venous compliance (cc/mmHg)
C_{AP} : Pulmonary arterial compliance (cc/mmHg)
C_{VP} : Pulmonary venous compliance (cc/mmHg)

R_{MV} : Mitral valve resistance

R_{AV} : Aortic valve resistance

R_{TV} : Triscupid valve resistance

R_{PV} : Pulmonary valve resistance

B : Total blood volume except the ventricles (cc)

The system variables of this model are as follows (outputs);

P_{AS} : Systemic arterial pressure (mmHg)

P_{AP} : Pulmonary arterial pressure (mmHg)

P_{VP} : Pulmonary venous pressure (mmHg)

P_{VS} : Systemic venous pressure (mmHg)

Q_{AS} : Systemic arterial volume (cc)

Q_{AP} : Pulmonary arterial volume (cc)

Q_{VP} : Pulmonary venous volume (cc)

Q_{VS} : Systemic venous volume (cc)

Q_{LV} : Left ventricle volume (cc)

P_{LV} : Left ventricle pressure(mmHg)

Q_{RV} : Right ventricle volume (cc)

P_{RV} : Right ventricle pressure(mmHg)

F_{MV} : Flow through the mitral valve (cc/sec)

F_{AV} : Flow through the aortic valve (cc/sec)

F_{TV} : Flow through the triscupid valve (cc/sec)

F_{PV} : Flow through the pulmonary valve (cc/sec)

In the above model several components are used. These are resistances, valves, and compliances. Properties of these components are described in detail in the following section.

Resistance: Due to their characteristics, blood vessels are supposed to resist to blood flow. Assume an ideal segment of a cylindrical vessel with rigid wall as shown below in Figure 2.2. The pressure difference between the two ends of the vessel is a function of flow [1]. In most cases this dependence can be modeled as a linear relation. The constant of proportionality between pressure difference and flow is then called resistance (to flow) as in the following equation 2.1.

$$R = (P_1 - P_2) / F \qquad (2.1)$$

where P1, P2 are the blood pressures at two ends of the vessel and F is the flow through it.

In the above CVS model, for example R_S represents the resistance of the combined (in series) artriol-capillary-venule system in the systemic circulation. Since these are vessels with small diameters they exert considerable resistance to flow. On the other hand major arteries and veins have large diameters and they do not have much resistance to flow.

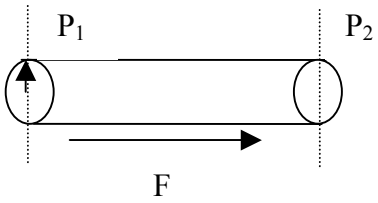


Figure 2-2: Idealized segment of a vessel with radius r

Valve: Valves enable one-way flow according to the pressure difference between its nodes as shown in Figure 2.3. Mitral valve, tricuspid valve, pulmonary and arterial valves are modeled as "Valve" component. It is a nonlinear component and is modeled very similar to a diode. Due to their characteristics, valves resist to the blood flow. For this reason, in most models valve component is modeled including a resistance.

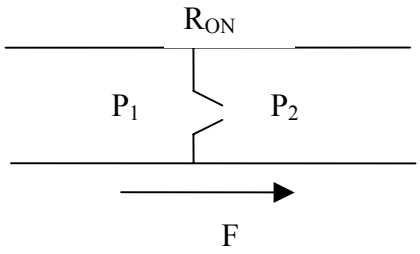


Figure 2-3: The schematic diagram of a valve

The relation between the flow between two nodes of the valve and the pressure difference is as follows:

$$F = \begin{cases} 0 & , P_1 \le P_2 \\ (P_1 - P_2) / R_{ON} & , P_1 > P_2 \end{cases} \qquad (2.2)$$

where R_{ON} is the resistance of the valve to the flow in the given direction and P_1 and P_2 are the pressures at each side of the valve. This equation basis on the assumption that the flow of blood in reverse direction is zero even if $P_2 > P_1$. However, valves are normally not so ideal and there might be a small amount of flow in reverse direction when $P_2 > P_1$ is big enough. Another flow equation considering this phenomenon is as follows:

$$F = \begin{cases} (P_1 - P_2) / R_{OFF} & , P_1 \le P_2 \\ (P_1 - P_2) / R_{ON} & , P_1 > P_2 \end{cases} \qquad (2.3)$$

where R_{ON} is the resistance of the valve in the forward direction of flow and R_{OFF} is the resistance of the valve in the reverse direction flow. Typical values of R_{OFF} and R_{ON} show that $R_{OFF} >> R_{ON}$

In Figure 2.4. the aortic valve component of the CVS model given in Figure 2.1 is figured. As can be seen from the figure, valve and resistance are grouped to model a valve component.

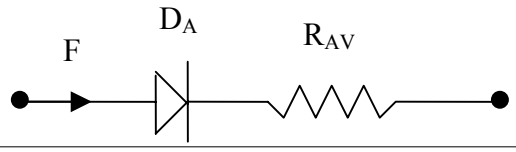


Figure 2-4: Component Model of the Aortic Valve

Compliance: Compliance can be described as the measure of the ability of a vessel to change its volume in response to applied pressure between its interior and exterior. It is calculated as the ratio of volume change to internal pressure change (dQ/dP). In another words, compliance is the inverse of elastance. Systemic and pulmonary arteries and veins can be modeled as "compliance" components. These are nonlinear components, but can be modeled as having a linear characteristic for most practical purposes. Arterioles, capillaries and venules do not have elastic walls and therefore their compliances are negligible.

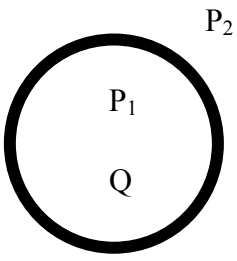


Figure 2-5: A cross-section of a cylindrical vessel.
Q is the volume of the vessel, P_1 and P_2 are interior and exterior pressures respectively

For the schematic given in Figure 2.5, the compliance can be calculated as eq. 2.4.

$$C = Q / \Delta P \quad \Rightarrow \quad C = Q / (P_1 - P_2) \tag{2.4}$$

In the above model (Figure 2.1), left and right ventricles are modeled as "Variable Compliance" components. Thus, left and right ventricular components show time-dependent variations that contraction of the heart and circulation of blood into vessels. In this model, the time-dependent variation of the variable compliance of left heart is defined by the inverse of "Stiffness" variable. Stiffness of the left ventricle is defined with a mathematical function as follows:

S_{LV} : Stiffness of Left Ventricular

$C_{LV} = 1/S_{LV}$ where (2.5)

$$S_{LV} = \begin{cases} S_{LD} + S_{LS} * \sin(2\pi t / 2T_S) & \text{for } 0 < t < T_S \text{ and,} \\ S_{LD} & \text{for } T_S < t < T_H \end{cases} \tag{2.6}$$

where S_{LD} and S_{LS} are constants.

C_{RV} is given with the same function except that $S_{LV,}$ S_{LS} are replaced with constants $S_{RD,}$ S_{RS}

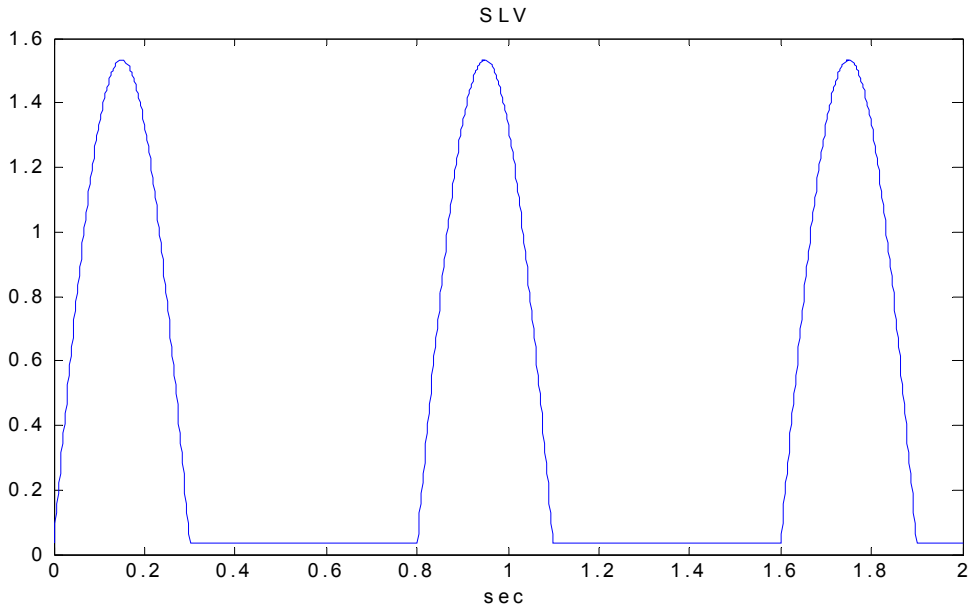
The graphical representation of S_{LV} and C_{RV} are given in Figure 2.6 (a). and 2.6.(b) respectively for T_H = 0.8 sec, T_S = 0.3, S_{LD} = 0.033 sec, S_{LS} = 1.5 sec

for time period of 2 sec.

The heartbeat has two phases, called systole and diastole. Systole refers to the contraction phase, when the ventricular pressure rises, thus blood flows out of ventricle to arteries, and diastole refers to the relaxation phase, when ventricle pressure is low, thus ventricules fill with blood.

As parameter values of the variable compliances C_{LV} and C_{RV} vary inversely proportional to the mathematical relations of S_{LV} and S_{RV} given above, the system modeled in Figure 2.1 functions as follows: at the beginning of the systole phase, the compliance of left ventricle, C_{LV} decreases sharply (Figure 2.6). As a result, P_{LV} increases sharply. This lead aortic valve to open while the mitral valve closes. During this time, the pressure difference between P_{LV} and P_{AS} reaches to its highest value. Hence, blood flows into aorta from left ventricle. After decreasing for a short period, C_{LV} reaches to its lowest value and remains nearly constant for a short time period. On the other hand, P_{LV} starts to decline after it reaches to its highest value. In the last time interval of systole, C_{LV} starts to increase dramatically and P_{LV} declines sharply. When P_{LV} reaches to its lowest value the systole phase (T_S) terminates. At that moment, P_{LV} is smaller than both P_{AS} and P_{VP}. This causes aortic valve to close and mitral valve to open which assigns that diastole phase starts. Until the end of the diastole phase blood flows from left atrium into left ventricle while C_{LV} and P_{LV} remain constant. Typical values for durations of systole and diastole times are 0.3 and 0.5 sec respectively. The left heart and right heart are modeled very similar, and the compliance of C_{RV} is the same as C_{RV} with different values of parameters. Hence, the functioning of right heart is very similar to the right heart functioning described above. As C_{LV} and C_{RV} vary periodically, each heart beat cycle, this process repeats and the circulation of the blood through vessels is realized.

(a)



(b)

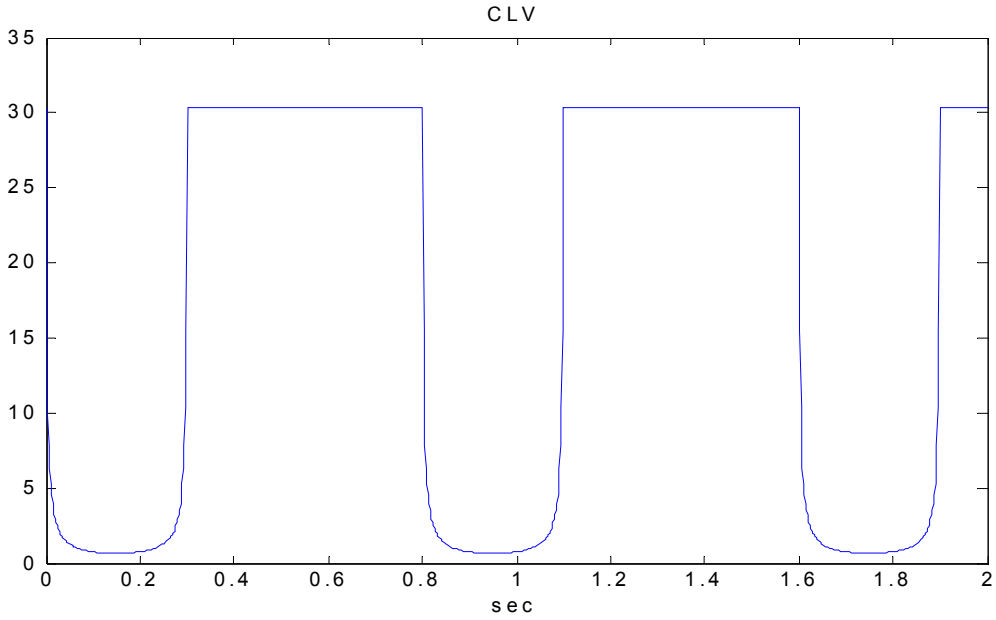


Figure 2-6: Simulation outputs of the left and right compliances

(a) Graphical representation of SLV function with parameters TH = 0.8 sec, TS = 0.3, SLD = 0.033 sec, SLS = 1.5

(b) Graphical representation of CLV = 1/SLV

According to steady flow dynamics, the volume of blood must be conserved. This rule is called "Conservation of Mass" rule. Conservation of mass rule leads us to the following differential equations for our CVS model:

$$dQ_{AS}/dt = F_{AV} - ((P_{AS} - P_{VS})/R_S) \quad (2.7)$$

$$dQ_{VS}/dt = ((P_{AS} - P_{VS})/R_S) - F_{TV} \quad (2.8)$$

$$dQ_{AP}/dt = F_{PV} - ((P_{AP} - P_{VP})/R_P) \quad (2.9)$$

$$dQ_{VP}/dt = ((P_{AP} - P_{VP})/R_P) - F_{MV} \quad (2.10)$$

$$dQ_{LV}/dt = F_{MV} - F_{AV} \quad (2.11)$$

$$dQ_{RV}/dt = F_{TV} - F_{PV} \quad (2.12)$$

Utilizing the equation eq. 2.4, we obtain;

$$dP_{AS}/dt = (1/C_{AS})*(F_{AV} - (P_{AS} - P_{VS})/R_S) \quad (2.13)$$

$$dP_{VS}/dt = (1/C_{VS})*((P_{AS} - P_{VS})/R_S) - F_{TV} \quad (2.14)$$

$$dP_{AP}/dt = (1/C_{AP})(F_{PV} - ((P_{AP} - P_{VP})/R_P)) \quad (2.15)$$

$$dP_{VP}/dt = (1/C_{VP})*(((P_{AP} - P_{VP})/R_P) - F_{MV}) \quad (2.16)$$

$$dP_{LV}/dt = (1/C_{LV})*(F_{MV} - F_{AV}) \quad (2.17)$$

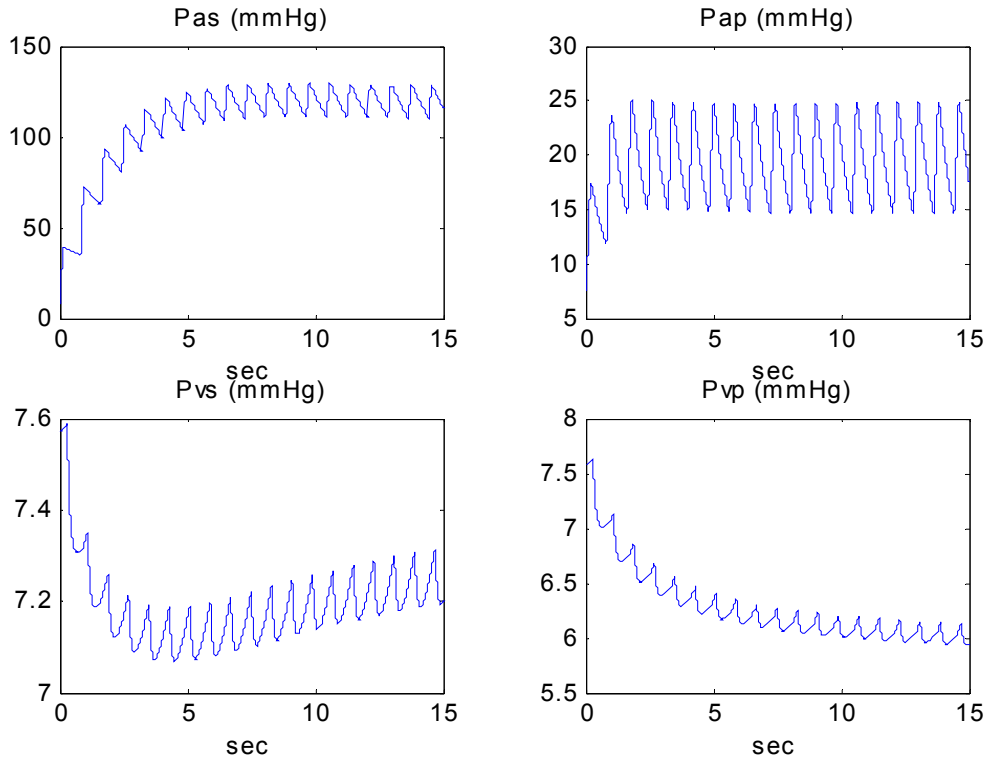
$$dP_{RV}/dt = (1/C_{RV})*(F_{TV} - F_{PV}) \quad (2.18)$$

Solving the differential equations obtained above numerically by using Euler integration with small time increments, we can obtain instant pressure and volume values of left ventricle, right ventricle, systemic artery, systemic venous, pulmonary artery, and therefore any flow value passing through a component.

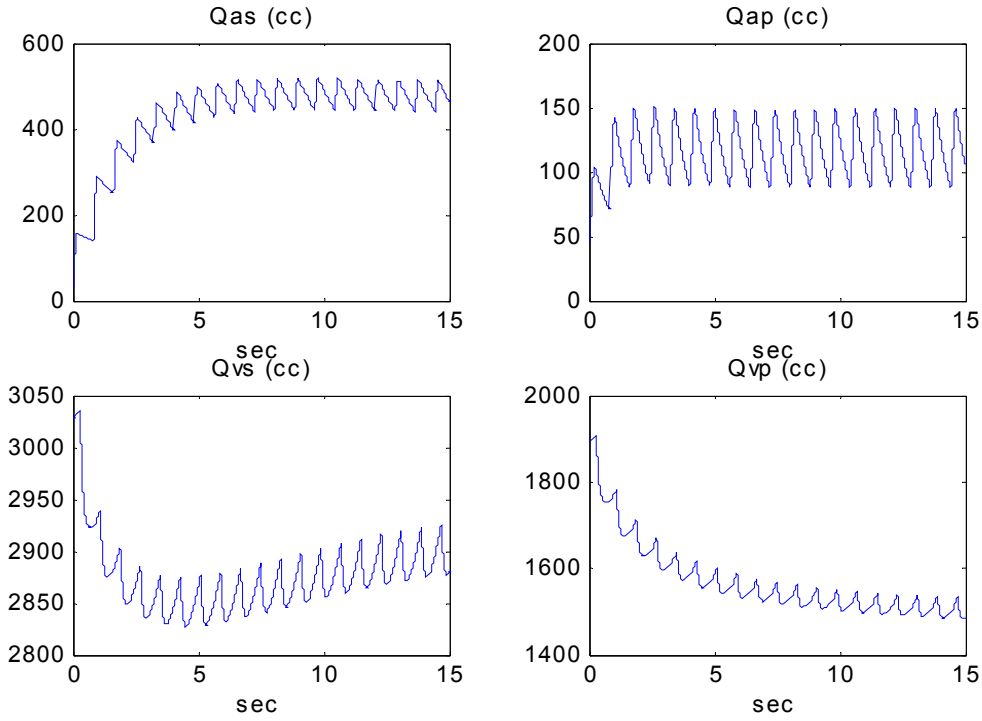
The output of the MATLAB implementation of simulating the CVS

mechanical model explained so far, with the given parameters are depicted in Figure 2.7. Simulation parameters are as follows: Simulation Duration = 15 sec, Step Size = 0.002 sec, TS=0.3 sec, TH=0.8 sec, Rmv=0.004 mmHg.sec/cc, Rtv=0.004 mmHg.sec/cc, Rav=0.06 mmHg.sec/cc, Rpv=0.06 mmHg.sec/cc, Cas=4 cc/mmHg, Cvs=400 cc/mmHg, Cap=6 cc/mmHg, Cvp=250 cc/mmHg, Rs=1 mmHg.sec/cc, Rp=0.125 mmHg.sec/cc, Blood volume except ventricles =5000 cc, SLS=1.5, SLD=0.033, SRS=0.3, SRD=0.033

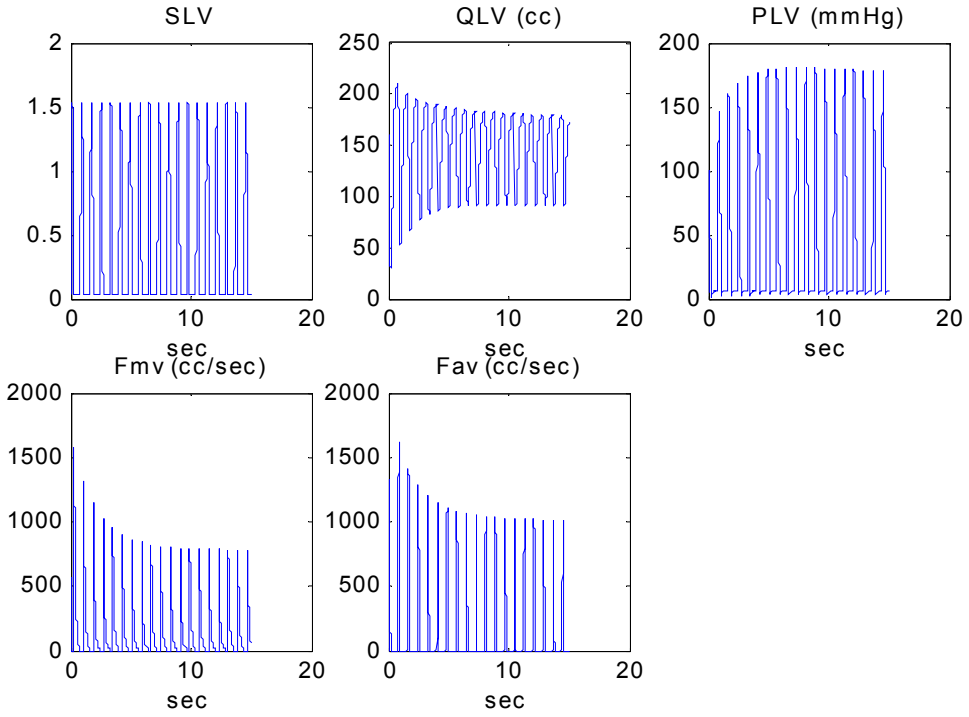
(a)



(b)


Qas (cc)
600
400
200
0
0
5
10
15
sec
Qap (cc)
200
150
100
50
0
0
5
10
15
sec
Qvs (cc)
3050
3000
2950
2900
2850
2800
0
5
10
15
sec
Qvp (cc)
2000
1800
1600
1400
0
5
10
15
sec

(c)


SLV
2
1.5
1
0.5
0
0
10
20
sec
QLV (cc)
250
200
150
100
50
0
0
10
20
sec
PLV (mmHg)
200
150
100
50
0
0
10
20
sec
Fmv (cc/sec)
2000
1500
1000
500
0
0
10
20
sec
Fav (cc/sec)
2000
1500
1000
500
0
0
10
20
sec

(d)

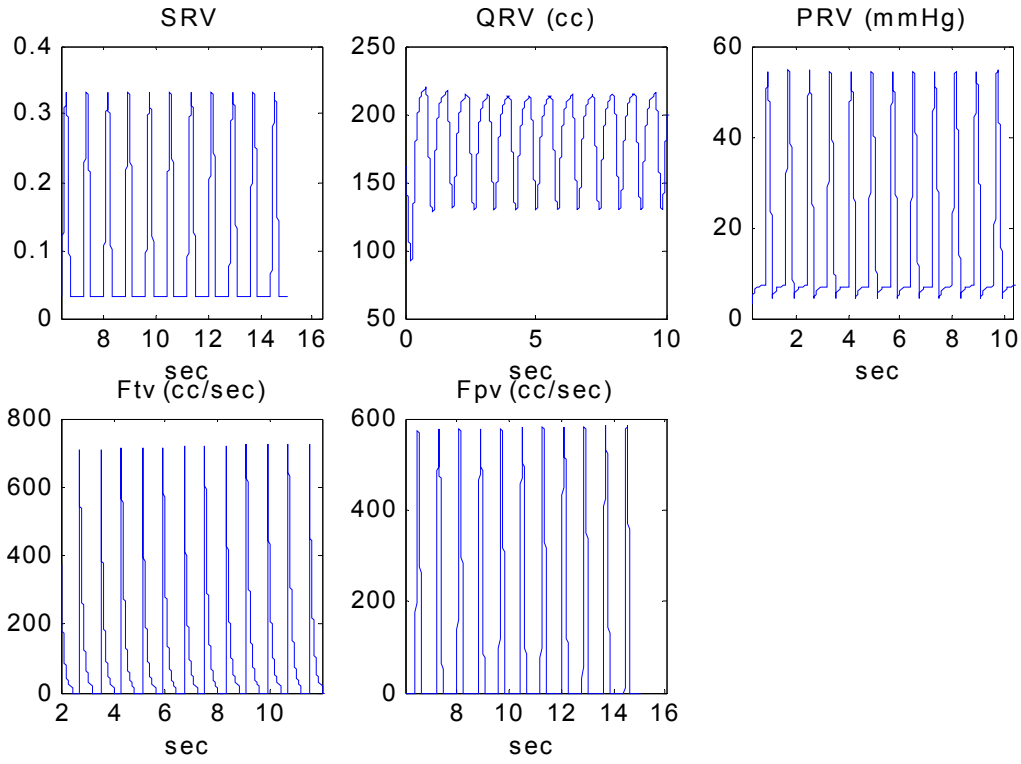


Figure 2.7. Simulation of the CVS model explained above.

(a) Systemic arterial, Pulmonary arterial, Systemic venous, and Pulmonary venous pressure variations versus time.

(b) Systemic arterial, Pulmonary arterial, Systemic venous, and Pulmonary venous volume variations versus time.

(c) Left ventricle stiffness, volume, pressure, blood flow through mitral valve, blood flow through aortic valve variations versus time.

(d) Right ventricle stiffness, volume, pressure, blood flow through tricuspid valve, blood flow through pulmonary valve variations versus time.

2.3 Models of CVS Regulation

The variations in the blood pressure of CVS are regulated by short-term and long-term blood pressure regulation mechanisms. Long-term blood pressure regulation stands for duration of minutes to hours or even to days and is achieved by adjusting blood volume to the required level. On the other hand, short-term blood pressure regulation stands for duration of some seconds to minutes, and achieved by regulating mechanical system parameters of the CVS, such as blood vessel diameter (directly effects resistance), heart rate and heart contractility. Considering long time periods of several hours to days, renal system is effective in long-term blood volume regulation. However, in short-term blood pressure regulation, the Autonomic Nervous System plays the major role. In the rest of this document, the discussion will concentrate on short-term blood pressure regulation. Thereby, long-term blood pressure regulation will not be detailed.

The mechanical properties of the cardiovascular system are modulated by feedback control mechanisms acting through the central nervous system. Feedback control mechanisms may include several feedback pathways depending on the aim that the model concentrates on. Two major pathways are baroreflex loop for control of arterial pressure and the cardiopulmonary reflex that controls circulatory blood volume, respectively.

The tension-sensitive baroreflex pressure receptors, called baroreceptors are located on the wall of carotid sinuses and on the wall of the aorta (at the aortic arc). These baroreceptors sense changes in ABP (Arterial Blood Pressure) and transmit this information to the Cardiovascular Control Center that is a part of the ANS (Autonomic Nervous System). The blood pressure information is transmitted by baroreceptors to the ANS by means of

firing rate that results in increase as blood pressure increases and result in decrease as blood pressure decreases. The ANS receives other information from other parts of the body regarding circulation as well as pressure changes. Among these information, O_2 and CO_2 concentrations in the arteries and information coming from higher brain centers can be mentioned. ANS neurons process all information they receive and modulate hemodynamic variables of the CVS through two pathways called sympathetic and parasympathetic (vagal) nerves by innervating the heart and blood vessels. Peripheral resistance, heart rate and heart contractility can be mentioned among hemodynamic variables of the CVS. Similarly, the cardiopulmonary volume receptors located in right atrium sense the right atrial pressure changes and transmits this information to the ANS. ANS takes regulatory actions by modifying hemodynamic variables of the cardiovascular system in order to regulate the pressure. These processes are called “Negative Feedback”.

In the following part, the function of the SA node (sinoatrial node) will be described emphasizing the effect of the firing rates of the sympathetic (shown by fs) and the parasympathetic (shown by fp) nerves, stimulating the SA node cells. Later on, the overall feedback control mechanism will be pictured to combine and visualize information given so far. Normal heartbeats originate from the SA node. ANS regulates heart rate by modulating the automaticity of the SA node. Autonomic nerves release the neurotransmitters that influence the automaticity of the SA node cells. The norepinephrine (NE) and acetylcholine (Ach) are the neurotransmitters released from sympathetic and parasympathetic (vagal) nerve endings respectively. The beating rate of the heart depends on the combination of the instantaneous NE and Ach concentrations in the synapses between neural fibers and the SA node cells, shown by NE(t) and Ach(t). The NE(t) depends on the sympathetic pathway

firing rate (fs) and Ach(t) depends on the parasympathetic pathway firing rate (fp) respectively. The heart rate variation in respect to the combined fs and fp is still under investigation and several mathematical models have been released so far [7, 8, 9, 10].

As stated previously, the heartbeat series are generated by SA node. The heartbeat interval is dependent on NE(t) and Ach(t) concentrations. A typical SA node cell membrane potential variation versus time is depicted in Figure 2.8. Consider that this simplistic illustration emphasizes the occurrences of the heartbeat series. The spontantenaous potential peaks denote heartbeats.

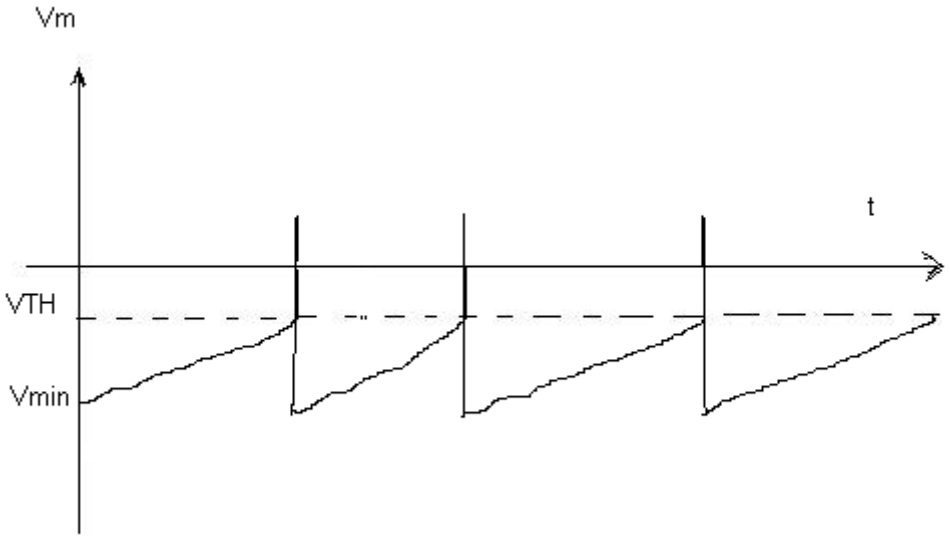


Figure 2.8. SA node cell membrane potential and heartbeat occurrences

For each heartbeat interval, as NE(t) and Ach(t) change, the cell membrane potential V_M varies between a minimum potential value V_{MIN} and a threshold value V_{TH}. At the beginning of each interval, V_{MIN} is reset to the minimum value V_{MIN}. V_M increases slowly approaching to the threshold value V_{TH}. The increase is nonlinear and the instantaneous slope depends on NE(t) and Ach(t). The larger the slope, the earlier the heartbeat is. As V_M reaches to

V_{TH} a heartbeat occurs, V_M makes an instant peak and just after its value is set to V_{MIN} for a new heartbeat cycle. This mechanism of heartbeat generation is modeled by the so-called "Integrate and Fire" models.

The building blocks of the negative feedback mechanism is explained so far. The block diagram of the overall mechanism that takes P_{AS} as input and generates heartbeat series as output is pictured in Figure 2.9 [11].

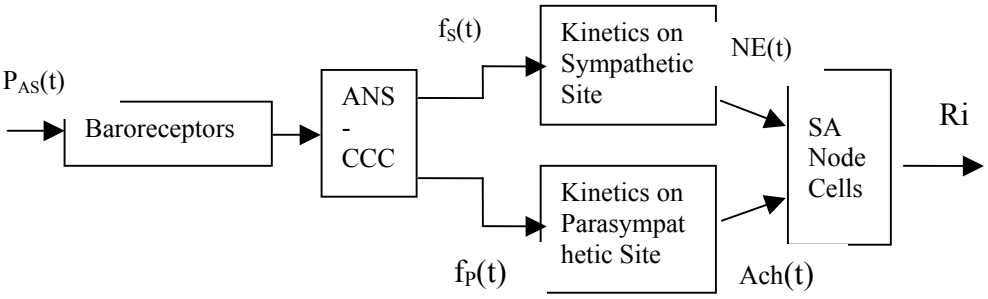


Figure 2.9. Block diagram of the negative feedback control loop. Pas(t) denotes the instant arterial blood pressure, fb(t) denotes the instantaneous firing rate of the baroreceptors, fs(t) and fp(t) denote instantaneous neural activities on sympathetic and parasympathetic nerves, respectively. NE(t) and Ach(t) are the effective concentrations of NE and Ach, respectively. Ri (i=0,1,2,...) denote discrete times at which heartbeat occurs.

Several studies have been carried out to propose a mathematical model that models the functioning of the overall system and many others has been proposed for particular functioning of subsystems. For example, Warner et al. [8-9] proposed a mathematical model to simulate the dynamics of SA node in response to sympathetic and parasympathetic stimulations. But Warner model does not give equations relating blood pressure with sympathetic and parasympathetic firing rates. Another simple model discussed below, gives two

differential equations relating NE(t) and Ach(t) directly with systemic arterial pressure, thus omitting calculation of fb(t), fs(t) and fp(t).

$$dNE(t)/dt = K_{sym}[P_{const} - P_{AS}(t - \tau_{sym})] - NE(t)*k_{sym} \quad (2.19)$$

$$dAch(t)/dt = K_{Ach}*P_{AS}(t - \tau_{Ach}) - Ach(t) * k_{Ach} \quad (2.20)$$

where $K_{sym} = K_{Ach} = 0.3$, $P_{const} = 120$ mmHg, $\tau_{sym} = 2.5$ sec, $\tau_{Ach} = 0.5$ sec, $k_{sym} = 0.239$, $k_{Ach} = 2$

For a given P_{AS} at a given t, by solving two differential equations of eq.2.19 and eq.2.20, NE(t) and Ach(t) can be calculated. To calculate the times where heartbeats occur, the differential equation eq. 2.21 is given which relates the slope of rising portion of $V_M(t)$ to NE(t) and Ach(t). Assuming the last heartbeat has occurred at t = 0, for $t \geq 0$

$$U(t) = V_M(t) - V_{MIN},\ dU/dt = C + c(t),\ U(0) = 0 \quad (2.21)$$

Where C = 180, and c(t) = BRS*[0.25*(15 – Ach(t)) + 0.25*(NE(t) - 20)], BRS is called the baroreflex gain and takes values between 0 and 50.

Heartbeats occur only when U reaches to the value $V_{TH} - V_{MIN}$ where 144 is a typical value. As heartbeat occurs the value of U is reset to 0.

A comprehensive study to postulate a mathematical model of short-term arterial pressure control by the carotid baroreceptors in pulsatile conditions is carried out by Ursino [12]. In this model Ursino describes all steps of baroregulation and gives mathematical equations for each particular sub-block. The overall feedback mechanism proposed in this model is similar to the Figure 2.9 interacting with the mechanical cardiovascular model. However, Ursino makes a distinction among afferent pathway (involving the carotid baroreceptors and the sinus nerve), the efferent sympathetic and parasympathetic pathways. Another difference is that, Ursino gives equations

to calculate the heart period changes in response to sympathetic and parasympathetic nerves activities instead of calculating the NE and Ach concentrations. The mathematical model of carotid baroreflex control system of Ursino model is described in detail below.

2.3.1 Carotid Baroreflex Control System of Ursino Model

Afferent Pathway: Ursino model describes the afferent baroreflex pathway as the series arrangement of a linear derivative first-order dynamic block and a sigmoidal static characteristic

$$\tau_p * dP'/dt = P_{cs} + \tau_z * dP_{cs}/dt - P' \quad (2.22)$$

$$f_{cs} = [f_{min} + f_{max} * \exp((P'-P_n)/k_a)] / [1 + \exp((P'-P_n)/k_a)] \quad (2.23)$$

where τ_p and τ_z are the time constants for the real pole and the real zero in the linear dynamic block (usually $\tau_z/\tau_p > 1$). P_{cs} is the carotid sinus pressure, P' is the output variable of the linear dynamic block (having the dimension of a pressure), f_{cs} is the frequency of spikes in the afferent fibers, f_{max} and f_{min} are the upper and lower saturation of the frequency discharge, P_n is the value of intrasinus pressure at the central point of the sigmoidal functional, and k_a is a parameter with the dimension of pressure, related to the slope of the static function at the central point. In closed-loop conditions, carotid sinus pressure is equal to systemic arterial pressure ($P_{cs} = P_{AS}$). τ_z =6.37 sec, f_{min}=2.52 spikes/sec, f_{max}=47.78 spikes/sec are typical values of the model.

Efferent Sympathetic Pathway: The autonomic activity of the efferent sympathetic pathway is given with the equation 2.24.

$$f_{es} = f_{es,\infty} + (f_{es,0} - f_{es,\infty})*\exp(-k_{es}*f_{cs}) \quad (2.24)$$

where f_{es} is the frequency of spikes in the efferent sympathetic nerves and k_{es}, $f_{es,0}$ and $f_{es,\infty}$ are constants (with $f_{es,0} > f_{es,\infty}$)

Efferent Parasympathetic Pathway: The autonomic activity of the efferent parasympathetic pathway is given with the equation 2.25.

$$f_{ev} = [f_{ev,0} + f_{ev,\infty}.\exp((f_{cs} - f_{cs,0})/k_{ev})] / [1 + \exp((f_{cs} - f_{cs,0})/k_{ev})] \quad (2.25)$$

where f_{ev} is the frequency of spikes in the efferent parasympathetic fibers, k_{ev}, $f_{ev,0}$ and $f_{ev,\infty}$ are constant parameters (with $f_{ev,\infty} > f_{ev,0}$), and $f_{cs,0}$ is the central value in eq. 2.23.

Regulation Effectors: Beside the heartbeat period, in his model, Ursino defines feedback mechanisms for all parameters of the CV plant, which are effected by sympathetic nerves. Ursino lists these parameters as resistances, unstressed volumes, and cardiac elastances. A generic mathematic model applicable to all parameters is postulated to model the baroregulation of these parameters. This mathematical model contains three equations which includes pure latency, a monotonic logarithmic static function and a low-pass first order dynamics as given in equations 2.26, 2.27, 2.28. These equations are the same for all parameters with different constant parameter values.

$$\sigma_{\Theta}(t) = \begin{cases} G_{\Theta} * \ln[f_{es}(t - D_{\Theta}) - f_{es,min} + 1] & \text{if } f_{es} \geq f_{es,min} \\ 0 & \text{if } f_{es} < f_{es,min} \end{cases} \quad (2.26)$$

$$d\Delta\Theta/dt = (1/\tau_\Theta)*[-\Delta\Theta(t) + \sigma_\Theta(t)] \quad (2.27)$$

$$\Theta(t) = \Delta\Theta(t) + \Theta_0 \quad (2.28)$$

where Θ denotes the generic control parameter, σ_Θ is the output of the static characteristic, τ_Θ and D_Θ are the time constants and pure latency, respectively. $f_{es,min}$ is defined as the minimum sypathetic stimulation and $\Delta\Theta$ is the parameter change caused by sympathetic stimulation. Θ_0 is the cumulative value of Θ at the previous step. G_Θ is described as a constant gain factor, which takes positive values for mechanisms working on maximum elastance of the left and right ventricles, systemic splanchnic and systemic extrasplanchnic resistances, and negative for unstressed volumes of splanchnic and extrasplanchnic compliances. Consider that the concepts splanchnic, extrasplanchnic and unstressed volumes are not mentioned in this paper so far. The plant of Ursino model has some differences from our described model. We will not go into details to explain them, but just describe them with a couple of words. Splanchnic and extrasplanchnic concepts are used to describe two parts of systemic circulation, and unstressed volume is the parameter of a special type of compliance component.

The particular implementation of the parameter control described above is detailed for heartbeat period parameter control. The heartbeat period is effected by both sympathetic and parasympathetic stimulations. For this reason, the mathematical model of heart period control has additional (eq. 2.30) and modified (eq. 2.33) equations. Ursino model gives the following equations that relate the heart period changes induced by sympathetic (ΔT_s) and parasympathetic (ΔT_v) stimulations, respectively. The heart period includes a balance between the sympathetic and parasympathetic activities

assuming a linear interaction between sympathetic and parasympathetic responses (2.33).

$$\sigma_{T,s}(t) = \begin{cases} G_{T,s}*\ln[f_{es}(t - D_{T,s}) - f_{es,min} + 1] & \text{if } f_{es} \geq f_{es,min} \\ 0 & \text{if } f_{es} < f_{es,min} \end{cases} \quad (2.29)$$

$$\sigma_{T,v}(t) = G_{T,v}*f_{ev}(t - D_{T,v}) \quad (2.30)$$

$$d\Delta T_S/dt = (1/\tau_{T,s})*[-\Delta T_S(t) + \sigma_{T,s}(t)] \quad (2.31)$$

$$d\Delta T_V/dt = (1/\tau_{T,v})*[-\Delta T_V(t) + \sigma_{T,v}(t)] \quad (2.32)$$

$$T = \Delta T_S + \Delta T_V + T_0 \quad (2.33)$$

where $\sigma_{T,s}$ and $\sigma_{T,v}$ are the output of the static characteristics of sympathetic and parasympathetic activities, respectively, $\tau_{T,s}$, $\tau_{T,v}$, $D_{T,s}$ and $D_{T,v}$ are the time constants and pure latency of the mechanism. $f_{es,min}$ is the minimum sympathetic stimulation, ΔT_S and ΔT_V are the heart period changes induced by sympathetic and parasympathetic stimulations respectively. Finally, T and T_0 denote the heart period and the heart period in the absence of cardiac innervation, respectively.

SA Node: Ursino uses so called "integrate and fire" model to describe the SA node function. The output of the "integrate and fire" variable is utilized in calculation of the ventricle activation function that is modeled as a squared half-sine wave. Integrate and fire model variable (u) is used in this model as a dimensionless variable, ranging between 0 and 1, that represents the fraction of cardiac cycle. The time instance where u = 0 is assumed to be the beginning of

the systole cycle. The value of u increases continuously and as it reaches to 1, its value is reset to 0, which implies a heartbeat and the beginning of a new systole cycle. An expression of u(t) which is modeled as “integrate and fire”, is given in eq. 2.34, as the function of T, instantenaous heart period calculated by eq. 2.33.

$$u(t) = \text{frac}\left[\int_{t_0}^{t} (1/T(\tau))\, d\,\tau + u(t_0)\right] \qquad (2.34)$$

Where the function “fractional part” [frac()] resets the variable u(t) to zero as soon as it reaches the value +1. The instances where u reaches +1 and resets to 0 model the peaks mentioned in Figure 2.8 where heartbeat occurs.

The output of the SA Node and the instantaneous heart period T calculated in eq 2.33 directly affect the left and right ventricle contraction and the systole time of the CV mechanical model, which explains a part of the carotid baroregulation of the Ursino model. It is stated that systole time decreases linearly with the instantaneous heart period. That is, the baroregulation system regulates the blood presure of the mechanical CV system by modulating the systole time and the left and right ventricle contraction, as well as some controlled parameters of resistances, unstressed volumes, and cardiac elastances. The list of the parameter values mentioned in Ursino model is given in Table 2.1.

To summarize the baroregulation control, a decrease in the aortic blood pressure results in decrease in baroreceptors firing rate. A decrease in baroreceptor firing rate, increases the sympathetic firing rate and decreases parasympathetic firing rate. This leads to increase in NE(t) and decrease in Ach(t). This concentration variation will lead to an increase in the heart rate,

stroke volume, and total peripheral resistance. Thus the overall effect of the negative feedback mechanism is an increase in the mean arterial blood pressure, and vice versa.

The mathematical models of feedback mechanisms are modeled as blocks in neurocardiovascular system models. Each block stands for a particular mathematical model, and described by its input, output and its mathematical model. Input of these blocks can be single input as well as multi-input. These inputs can be parameter of a circuit component as well as independent source. The output of a feedback block modulates hemodynamic variables of the cardiovascular system such as the parameter of a circuit component, or the heart rate. Neurocardiovascular system models has peculiar blocks as part of its functioning. Two of the most frequently used peculiar blocks are explained below:

SLV Block: "SLV Block" is the mathematical modeling of the time variation of the left heart stiffness that is the inverse of the left heart compliance. It generates sinusoidal stiffness variation between two threshold values (i.e. minimum and maximum values) during systole and takes constant signal value during diastole. It plays role in determination of the heart-beat contractility.

Integrate and Fire Block: "Integrate and Fire Block" that is also called Integral Pulse Frequency Modulation, is a mathematical modeling of transformation of continuous-time input signal into discrete-time series [9]. It is used to represent impulse generation process of the nerve cells of SA node. The output of this block, which generates heartbeat series, is either 0 or 1 at a time. As mentioned before, the SA node of Ursino model is a special type of integrate and fire block.

Carotid sinus afferent pathway	f_{min} = 2.52 spikes/s, f_{max} = 47.78 spikes/s, τ_z = 6.37 s, τ_p =2.076 s, P_n=92 mmHg, k_a=11.758 mmHg
Sympathetic efferent pathway	$f_{es,\infty}$ = 2,10 spikes/s, $f_{es,0}$ = 16.11 spikes/s, $f_{es,min}$ = 2.66 spikes/s, k_{es} = 0.0675 mmHg
Vagal efferent pathway	$f_{ev,\infty}$.= 6.3 spikes/s, $f_{cs,0}$ = 25 spikes/s, k_{ev} = 7.06 mmHg
Effectors	$G_{T,s}$ = -0.13 s/v, $G_{T,v}$ = 0.09 s/v, $D_{T,s}$ = 2 s, $D_{T,v}$ = 0.2 s, $\tau_{T,s}$ = 2 s, $\tau_{T,v}$ = 1.5 s, T_0 = 0.58 s

Table 2.1. Parameter values of the Ursino model

As a summary, a neurocardiovascular model can be divided into two parts: 1) first part is the mechanical model of the cardiovascular system and 2) second part is feedback control mechanisms model. The first part is composed of circuit components flow resistance, pressure source, flow source, compliance, variable compliance and inertance. Although we have not included the component “inertance” in our previous model, we have realized that some models [12, 1] use this component as a part of the mechanical cardiovascular system. The second part is composed of blocks, interacting with the mechanical system by taking parameters of circuit components as input and modulating parameters of the circuit components by its output. Detailed analysis of these models led us to the conclusion that, in order to be capable of simulating any model, our framework should provide users ability to describe their models using the following components: flow resistance, pressure source, flow source, valve, compliance, variable compliance, compliance with unstressed volume, inertance and blocks. Each component might be used

multiple times. There might be some components that are not commonly known but modeled by researchers. To enhance flexibility of the system, we observed the requirement to add a special type of component that can be defined by the user by supplying its description as well as its implementation.

The complete list of building elements of neurocardiovascular systems with their parameters and properties is given in Table 2. The "Inertance" component that is not explained in the previous section is described below.

Inertance: Inertance is described as the tendency of a flow to continue as the driving force (pressure) is removed [13]. For inertance the eq. 2.35 holds. These are nonlinear components, but can be modeled as having linear characteristic for most practical purposes. Inertance is important for vessels with large diameter, therefore the inertance of arterioles, capillaries and venules is negligible.

$$P = L.dF/dt \qquad (2.35)$$

(a)

Component	**Parameter**
Flow Resistance	Resistance – R
Pressure Source	Pressure – P
Flow Source	Flow – F
Ideal Valve	Threshold - P_{TH}
Compound Valve	Forward and Backward Resistances - R_{ON}, R_{OFF}
Compliance	Compliance – C

Variable Compliance	Compliance – C
Compliance with Unstressed Volume	Compliance – C, Unstressed Volume - V_U
Inertance	Inertance – L
User Component	N/A

(b)

Block
SLV Block
SRV Block
Inverter Block
Ax+b Block
Absolute Delay Block
Linear ARMA Block (Filter Block)
Integrate and Fire Block
Signal Source Blocks (Sin, Squarewave, Step, Impulse, etc)

Table 2.2 Expected Component and Blocks of the framework

(a) List of circuit components of a neurocardiovascular system model.

(b) List of some blocks of the feedback control mechanism

Chapter 3

NeuroCardioVascular Markup Language (NCVML)

3.1 Overview

In recent years, simulation of physiological systems has become a popular method especially for noninvasive medical research and medical education. As mentioned in the previous chapter, particularly in neurocardiovascular system discipline, many mathematical models and approaches for simulation of these models have been postulated. Following this, the concept of simulation of NCVS models with desktop applications have emegered. Currently, the simulation of models are carried out either with a programming language capable of implementing a mathematical model (i.e. Matlab) or with a circuit and system simulation software (i.e. Simulink, Pspice). The common property of all those application softwares is that they are all desktop applications. However, emergence of the world-wide web (WWW) has produced an environment within which many disciplines are being re-evaluated in terms of their inherent approaches, techniques and philosophies. Driven largely by the

phenomenal growth in the WWW and its attendant technologies, it is tempting to view web-based simulation as nothing more than a technology push [14].

In our study, we aimed to develop a special framework that enables web-based simulation of NCV models utilizing WWW. However, for web-based NCV simulation, the starting point of the simulation process is obtaining the simulation scenario from the user and providing the simulation software with this scenario. A simulation scenario contains the description of a NCV model as well as simulation parameters. In this context, the need for the availability of a description language that provides users facility to describe their NCV models, or in general the simulation scenarios are obvious. The design of such a description language should satisfy the following requirements:

i). It should be designed primarily for neurocardiovascular system models, thus, the set of concepts should compose of neurocardiovascular model concepts,
ii). It should be independent of the analysis tool [15],
iii). It should be adaptable to different scenarios [15],
iv). It should support transfer of data via the web,
v). It should be extensible [15].

With the guidance of those requirements given above, we have defined an XML-based Description Language for Neurocardiovascular Models, which we name NeuroCardioVascular Markup Language (NCVML), to let users describe their models. In other words, NCVML is developed as the description language for simulations of neurocardiovascular models. It aims to give potential users the ability to represent their models in an understandable, standard-like way, using component-oriented approach. As its design is based

on actual practical simulation software, it is a practical and functional language.

NCVML is constructed following the analysis of actual neurocardiovascular models and updated iteratively during the development of practical simulation software to reflect emerging requirements. Following this process, the building blocks of NCVML are constructed based on actual neurocardiovascular model elements. Thus, NCVML satisfies the requirement ***i***

Although NCVML is developed primarily as a description language for our simulation package NCVJSim, generic concepts such as *component, connection* and *block* (this concepts are described in detail in Section 3.2) are introduced that enable different model scenarios to be described easily as being independent of the analysis tool. These concepts and the component-oriented methodology defined for describing elements and parameters make NCVML independent of the analysis tool. Thus, NCVML satisfies requirements ***ii*** and ***iii***.

NCVML is based on XML [16] technology. XML is the dominating data exchange standard over the web nowadays. Models encoded with XML are simply text files and can be transferred or exchanged over the web easily. Additionally, XML is extensible in its nature. These lead us to the conclusion that NCVML satisfies the requirements ***iv*** and ***v***. Actually, item ***ii*** is not a strict precondition for a description language but it forms a basis for future integratebility of simulation framework and infrastructure with other frameworks for exchange of models.

Consider that, in our framework, NCVML can be viewed as the carrier of models over the web between users and the simulation package and when

the simulation package receives a model through NCVML, it derives internal data structures and objects from the model for model analysis.

The remainder of this chapter is organized as follows: in Section 3.1, an introduction to XML is given for those who are new to XML technology and its concepts. In Section 3.2, simulation scenario structure and proposed NCVML concepts are described. Section 3.3 explains basic concepts of Unified Modelling Language (UML). Section 3.4 describes the proposed NCVML in general and gives UML models of the proposed model. Following this, Section 3.6 illustrates the mapping from the UML model to XML. In this context, generation of the NCVML Data Type Definition (DTD) will be illustrated. Section 3.6 examines all element of the proposed NCVML model in detail, separately. In this section, XML realization of NCVML elements is also included. Finally, this document concludes with a simple example illustrating XML encoding of a NCV model using NCVML in Section 3.7.

3.2. Extensible Markup Language (XML)

Extensible Markup Language, abbreviated as XML, is a *standard*, *text-based*, *simple*, *self-describing* way of encoding both text and data so that content can be processed with relatively little human intervention and exchanged across diverse hardware, operating systems, and applications [17]. XML is derived from SGML [18], which is a system for defining rules for organizing and tagging elements of a document. However, SGML is a large and complex system, therefore it is difficult to learn and to use in the web environment. XML is a subset of SGML that arised as a new meta language to remove complexity of SGML while inheriting its benefits. XML became a W3C Recommendation on February 10th, 1998 as XML 1.0. This initial

recommendation was revised two times and XML 1.0 (third edition) is available since 04 February 2004.

XML is similar to HTML in its actual format. Both are closely related to the SGML markup definition language, and both are markup languages. So that those familiar with HTML can easily get familiar with basic XML knowledge. However, there are two fundamental differences between HTML and XML:

Separation of form and content: HTML mostly consists of tags defining the appearance of text; in XML the tags generally define the structure and content of the data, with actual appearance specified by a specific application or an associated stylesheet.

XML is extensible: XML provides set of grammar rules for describing document content. Thus, tags can be defined by individuals or organizations for some specific application, whereas the HTML standard tagset is fixed and defined by the World Wide Web Consortium (W3C).

XML documents are composed of *elements, attributes,* and *data.* Elements are building blocks, which are enclosed in a pair of angle brackets (<…>). An element name enclosed in brackets is called *tag*. Elements might have data between the start-tag (<element name>) and end-tag (</element name>). Elements might have attributes, which are located in the start-tag following element name. An element that has no data, and no end-tag is called *empty element*. Consider the following excerpt of an XML document:

<CATALOG>

...

```
...
<BOOK>
        <TITLE> Modeling XML Applications with UML </TITLE>
        <AUTHOR> David Carlson </AUTHOR>
        <PUBLISHER> ADDISON-WESLEY </PUBLISHER>
        <YEAR VALUE = "2001" />
        <PRICE>
                <CURRENCY>  $ </CURRENCY>
                <COST> 28 </COST>
        </PRICE>
</BOOK>
...
</CATALOG>
```

In the above example, CATALOG, BOOK, TITLE, AUTHOR, PUBLISHER, YEAR, PRICE, CURRENCY, and COST are all elements. CATALOG is called root element of the document. Consider that each element name is enclosed in <> brackets. Content of each element is enclosed between start and end-tag of an element. The element PRICE is a container of CURRENCY and COST elements. YEAR element is called empty element, which it does not have any content, or end-tag. VALUE is called the attribute of YEAR element and assigned value just following its name. Note that the syntax of the empty element YEAR is different than other elements, where the start-tag has the structure like <element_name attribute_list />.

As we mentioned previously, XML provides set of grammar rules for describing document content. The formal model that lest users describe their own XML models are called *Data Type Definition (DTD).* In other words, the purpose of a DTD is to define the legal building blocks of an XML document.

It defines the document structure with a list of legal elements, relationship between these elements and element attributes. DTD can be defined inline in a XML document or it can be a separate document. The possible DTD of the catalog XML document given above is given below. We call a possible DTD, because a document may satisfy rules of more than one DTD.

```
<!ELEMENT        CATALOG    (BOOK+)>
<!ELEMENT        BOOK
(TITLE,AUTHOR,PUBLISHER,YEAR,PRICE?)>
<!ELEMENT        PRICE      (CURRENCY,COST)>
<!ELEMENT        TITLE      (#CDATA)>
<!ELEMENT        AUTHOR     (#CDATA)>
<!ELEMENT        PUBLISHER  (#CDATA)>
<!ELEMENT        YEAR       EMPTY>
<!ELEMENT        CURRENCY   (#CDATA)
<!ELEMENT        COST       (#CDATA)>

<!ATTLIST        YEAR       VALUE      CDATA   #REQUIRED>
```

As can be seen, in DTD elements are defined in a tag using !ELEMENT prefix. All elements of an XML model must be defined in its DTD. The root element must be defined at top, and definition of elements must be in logical order. Here “+” represents at least one occurrence, “?” represents zero or one occurrences, and finally “*” represents zero or more occurrences of the element the character succeeds. The textual description of the first line is; a CATALOG element must be followed by at least one BOOK element. The sequential occurrence of elements in another element is represented as (ELEMENT1, ELEMENT2,..,ELEMENTN) using comma separator. Empty elements, which have no content, are defined using EMPTY keyword. In the

seventh line, YEAR element is defined as an empty element using EMPTY keyword. #CDATA keyword means that element content is character data. There are also #PCDATA, which means a parsed character data, ID for unique id data throughout a document and some others. But data types are limited with DTD, which is its main deficiency. Finally, attributes of elements are defined in a DTD using !ATTLIST keyword. In our case, YEAR element has an attribute VALUE, which has character data content. If an attribute value is compulsory, while defining an attribute, #REQUIRED keyword is included at the end of the attribute definition. #IMPLIED and #FIXED are other options, which mean the value is optional and value is fixed and cannot be changed, respectively.

We conclude our XML introduction by explaining well-formed and valid document concepts: an XML document is well-formed, if it is syntactically correct. An XML document is valid if it is well-formed as well as it conforms all rules of its DTD. Therefore, a well-formed document might not be valid document but valid documents are well-formed.

3.3 NCVML Concepts and Neurocardiovascular Model Simulation Scenario

As discussed in the previous chapter (Chapter 2), a neurocardiovascular model consists of a mechanical cardiovascular model and model of the baroregulation feedback mechanism. The mechanical model consists of circuit components and feedback mechanism model consists of a number of blocks. So, a simulation scenario must consist of the full description of a

neurocardiovascular model as well as simulation parameters. Therefore, full description of a neurocardiovascular model consists of the following parts:

- general parameters of the model,
- description of the components of the cardiovascular system as well as parameters of these components,
- mechanisms to describe Baroreflex feedback mechanism of Autonomic Nervous System. (ANS)

A mechanical model is outlined with its general parameters describing general properties of the model such as, number of circuit components, ground node, and number of nodes in the mechanical model. For a model parameter-value pairs of these parameters form the first part of a model description. The following is description of each mechanical model component in detail. Each component is described with its type, location and its parameter values. The first two parts constitute the mechanical cardiovascular model description. The last part of a neurocariovascular model description is the detailed description of baroreflex mechanisms. These mechanisms are modelled as blocks as mentioned in Chapter 2. Its type, its input and its output identify each block. All these concepts will be discussed in detail in Section 3.4.

To describe a model we introduced “Component”, “Connection” and “Block” concepts. The following is the detailed description of those concepts:

Component: Component is the building block of a circuit. A set of linear and non-linear circuit components such as Resistance, Pressure Source, Flow Source, Compliance, Variable Compliance, Compliance with Unstressed Volume, Inductance and Valve are the elements of Component set. These are the smallest subset included in Component set and can be expanded. Beyond this, in our framework, we let users describe their own components, which are

not included in our set, so that component set includes dynamic components whose names and functionalities are determined by the user. Thus, Component set is extendible and not limited with the items listed above. Each component is identified with a unique component number, its location, its type and its parameter values throughout the model.

Connection: Connection is the logical abstraction of the structure between two edges of a component. In other words, connection is container of a component. This concept is introduced upon the need to describe location of components. Each connection is identified by two variables: the first describes the label of the beginning of the connection and the second describes the label of the end point of the connection. Each connection contains exactly one component.

Block: Block is the logical structure to define the feedback mechanism that affects hemodynamic variables of the cardiovascular system. This concept emerged to define the mechanism that modifies a variable of a circuit component with its output, calculated by processing the input according to a mathematical model. In other words, block is a graphical representation of a mathematical model that receives input and produces output. A block is described with its unique number throughout the model, type, name, parameter array, target and inputs. Inputs can be more than one and one input might be either parameter of a component, voltage or current value of a node, another block output or independent source like time.

The following example might be useful for explaining "*Component*", "*Connection*" and "*Block*" concepts. Figure 1 shows a hypothetical model of a part of the blood circulation. Note that it is just for illustration of concepts and has no scientific basis.

In this small hypothetical model, there are 3 nodes labelled with Node1, Node2, Node3, in clockwise direction. The mechanical model consists of flow source, compliance, resistance and compliance components labelled with P, Ca, R, and Cb respectively. There is also a block element labelled with BlockR in the model. So, with our NCVML definition concepts mechanical model components, F, Ca, R, Cb map to *Component*. By definition, each

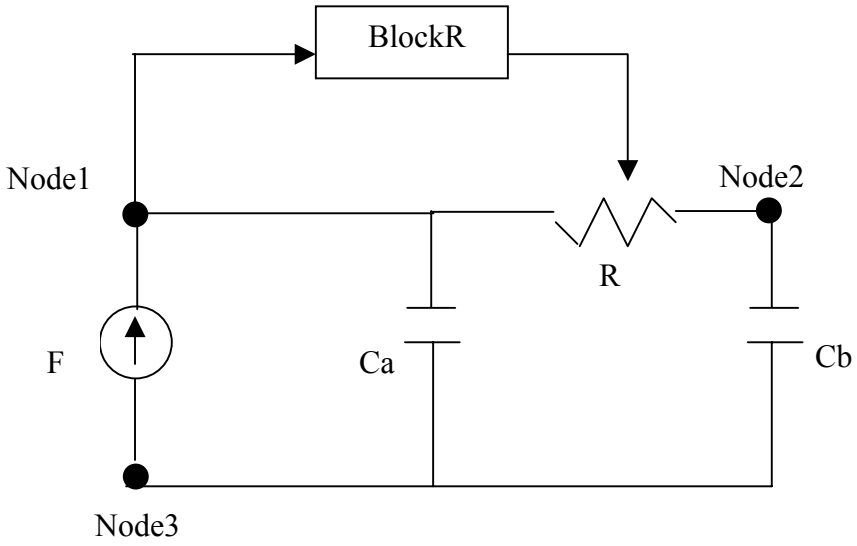


Figure 3.1 A hypothetical model

Component is enclosed in a *Connection;* therefore, in this model there are 4 *Connection*s. First *Connection* is the logical structure between Node3 and Node1 (i.e. connection starts at Node3 and ends at Node1), second is between Node1 and Node2, third is between Node1 and Node3, and the last is between Node2 and Node3. The only feedback in this model is depicted by BlockR, which takes the pressure value of the Node1 as input and modifies the

parameter of resistance *Component* with its output. Therefore, BlockR maps to *Block* in our NCVML definiton.

In our proposed NCVML definition, a NCVML model description simply looks like as follows:

beginning of model definition
    list of model parameters
    list of connections
    list of blocks (optional)
end of model definition

Simulation of an NCVML model is realized through a simulation scenario. An NCVML simulation scenario contains mainly two parts. The first part is the description of the model and the second part is description of simulation parameters such as simulation time, time interval, etc. Thus, simulation scenario looks like as follows:

beginning of simulation scenario

    beginning of model definition
        list of model parameters
        list of connections
        list of blocks (optional)
    end of model definition

    beginning of simulation parameters definition (optional)
        list of simulation parameters
    end of simulation parameters definition

end of simulation scenario

To discriminate between the terms “NCVML Simulation Scenario” and “NCVML Model” the following explanation might be useful: while NCVML model contains full conceptual description of a neurocardiovascular model with NCVML, NCVML simulation scenario is the package contains both a NCVML model and parameters defining the simulation process. Thus, while NCVML model is representation-oriented, NCVML simulation scenario can be defined as process-oriented.

To illustrate the structure and the relationship of NCVML components in a more understandable and standard way, *UML* notation will be utilized throughout this document.

We will make a small introduction of UML in the next section before going deep into the structure of NCVML.

3.4 What is Unified Modeling Language (UML)?

Before dealing with the definition of UML, we have to mention about concepts of *Modeling* and *Visual Modeling*. Models are described in [19] as abstractions that portray the essentials of a complex problem or structure by filtering out nonessential details, thus making the problem easier to understand. The real-world complex systems are abstracted using models. These abstractions might be drawings, geometrical shapes, or any kind of notation. Visual Modeling is a way of thinking about problems using models organized around real-world ideas. Models are useful for understanding problems, communicating with

everyone involved in the project (customers, domain experts, analysts, designers, etc.), modeling enterprises, preparing documentation, and designing programs and databases. Modeling promotes better understanding of requirements, cleaner designs, and more maintainable systems [19].

Unified Modelling Language (UML) is a graphical language for visualising, specifying, constructing, and documenting the artefacts of a software-intensive system [20]. UML has been defined by the Object Management Group www.omg.org and released version 1.4 in November 1997 and later version 2.0 released early in 2003.

Unified: Unification of Object Oriented Analysis & Design (OOAD) methods
Modelling: Analysing requirements and design prior to design of solutions
Language: A graphical notation to express you requirements and your design

UML's array of notations helps system designers to capture their ideas in a way that is expressive yet easy to understand. Thus, UML provides a set of graphical notations for describing new systems in a clear and non-ambiguous way [21]. UML models promote better understanding of requirements, cleaner specifications and designs in a technology and programming language independent way. Having these properties, UML has been accepted as the industry standard for designing new systems.

Class Diagram, Use Case Diagram, Sequence Diagram and Interaction Diagram, State Diagram, Activity Diagram and Implementation Diagram are main graphical notations provided by UML for system modelling. Each type of diagram models a particular aspect of OO design in an easy to understand, visual manner. The UML standard specifies exactly how the diagrams are to be drawn and what each component in the diagram means.

But in this section our interest lies only in class diagram. Class diagrams are used to model the static properties of a system. They represent the main building elements of a system and relationships between them. These building elements are represented either as classes or interfaces. The relationships between two or more classes are modelled as *association*, *aggregation*, *composition* and *generalization*.

Class: A UML class is a container of structural and behavioural features of an entity of a given system. In UML notation, class is represented with three compartments rectangle as shown below in Figure 3.2.a. Two sub-compartments are optional.

Notation of an attribute is as follows: "attribute name [multiplicity]: Type = initial value" Multiplicity is optional and represents the number instances that may take place in a class instance. In other words, multiplicity is the occurrence times of an attribute.

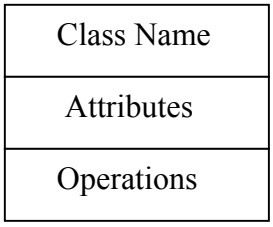
Notation of an operation is as follows: "operation name (argument list): return type"

Association is the case that two classes connected to each other in any way. It indicates the role of a class in the relationship. *Association* notation is given in Figure 3.2.b.

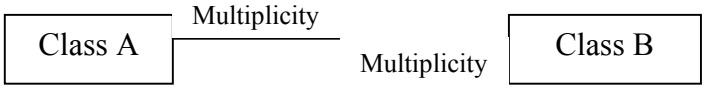
When a class is formed as a collection of other classes, it is called an *aggregation* relationship between these classes. Thus, aggregation models the notion that one object uses another object without owning it. *Aggregation* notation is given in Figure 3.2.c.

A *composition* models the notion of one object owning another. Thus, one object is part of another object. It is a strict form of *aggregation* relation. *Composition* notation is given in Figure 3.2.d.

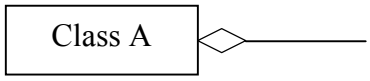
Generalization models the inheritance, which is the relationship between a more general element (the supertype) and a more specific element (the subtype). In generalization relationship, subtype inherits the common functionality defined in the supertype, it might have additional properties as well. *Generalization* notation is given in Figure 3.2.e.



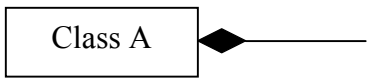
a. UML class notation



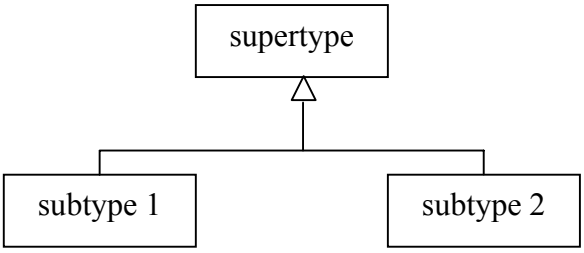
b. Association



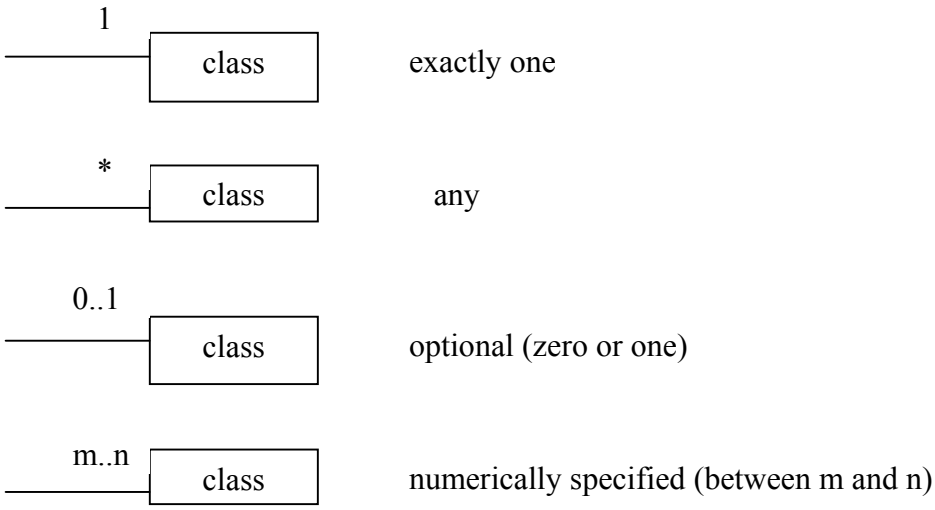
c. Aggregation



d. Composition



e. Generalization



f. Multiplicities

Figure 3. 2 : UML class diagram notations

3.5. Proposed NCVML Model

In this section, our proposed NCVML model is described in detail. In describing the model, top-down approach will be followed. That is, first the general model will be described, and following this each element of the model will be detailed.

As we mentioned earlier, NCVML is designed as an XML-based language for representation of NCV models. Practically speaking, each NCV model is mapped into XML document containing description of the NCV model using NCVML markups.

An XML document could be defined by a set of simple class definitions that specify all of the basic terms required by an application's vocabulary where vocabulary is described as the list of terms used in communication [22]. But, vocabulary is not enough individually to specify a model. Additional rules specifying the relationship between vocabulary term and semantics specifying how each term can or cannot be used are also needed. Following this, we can say that the composition of class definitions of vocabulary items, relationships between these classes, and semantics of vocabulary items form an XML model. UML class diagrams are very suitable for visual representation of XML models, because UML offers a rich set of relationships that can be defined between classes. For this reason, we are going to use UML class diagrams to visualize static class structure of our NCVML model.

We propose NCVML model that includes 25 classes, representing the model container, the model, model parameters, connections, components, blocks, block inputs, and simulation parameters, and enumerated type

parameters. This model is shown in Figure 3.3. Each of these classes will be described in detail in the Section 3.7.

In our NCVML model definition, an NCVML model container (NCVML class) is composed of a model definition (MODEL class) and includes a simulation parameter description container (SIMULATION class). A model (MODEL class) includes the following: exactly one NELEMENTS, one NNODES, one REFERENCENODE classes, at least one or multiple CONNECTION class, and zero or multiple BLOCK classes. The relations between MODEL and CONNECTION, NELEMENT, NNODES, REFERENCENODE are all composition relation. These relations are depicted on the figure with an arrow with a diamond. A model is associated with zero or more BLOCK classes and each BLOCK is associated with one or more INPUT classes. Similarly, each CONNECTION class is composed of either of RESISTANCE, PSOURCE, FSOURCE, VALVE, CVALVE, USERCOMPONENT classes. The occurrence of one of the listed classes is mutually exclusive. That means in one CONNECTION only one of them can take place at a time. A SIMULATION class is composed of exactly one TOTALTIME, one TIMEINTERVAL, zero or one INITIALVALUES classes and zero or more CHART classes. The model also includes six classes BlockType, SourceType, InputType, TargetType, ChartSourceType, ChartSourceParam, and ChartOutputType, which are annotated with <<enumeration>> stereotype. These classes are included to the model to limit the selection of type of a block, type of an input source, type of an block input and type of a block target with their attributes, source of the chart, type of the output of a chart values, and source type of the chart input. Consider that attributes of enumeration type classes do not take attribute type.

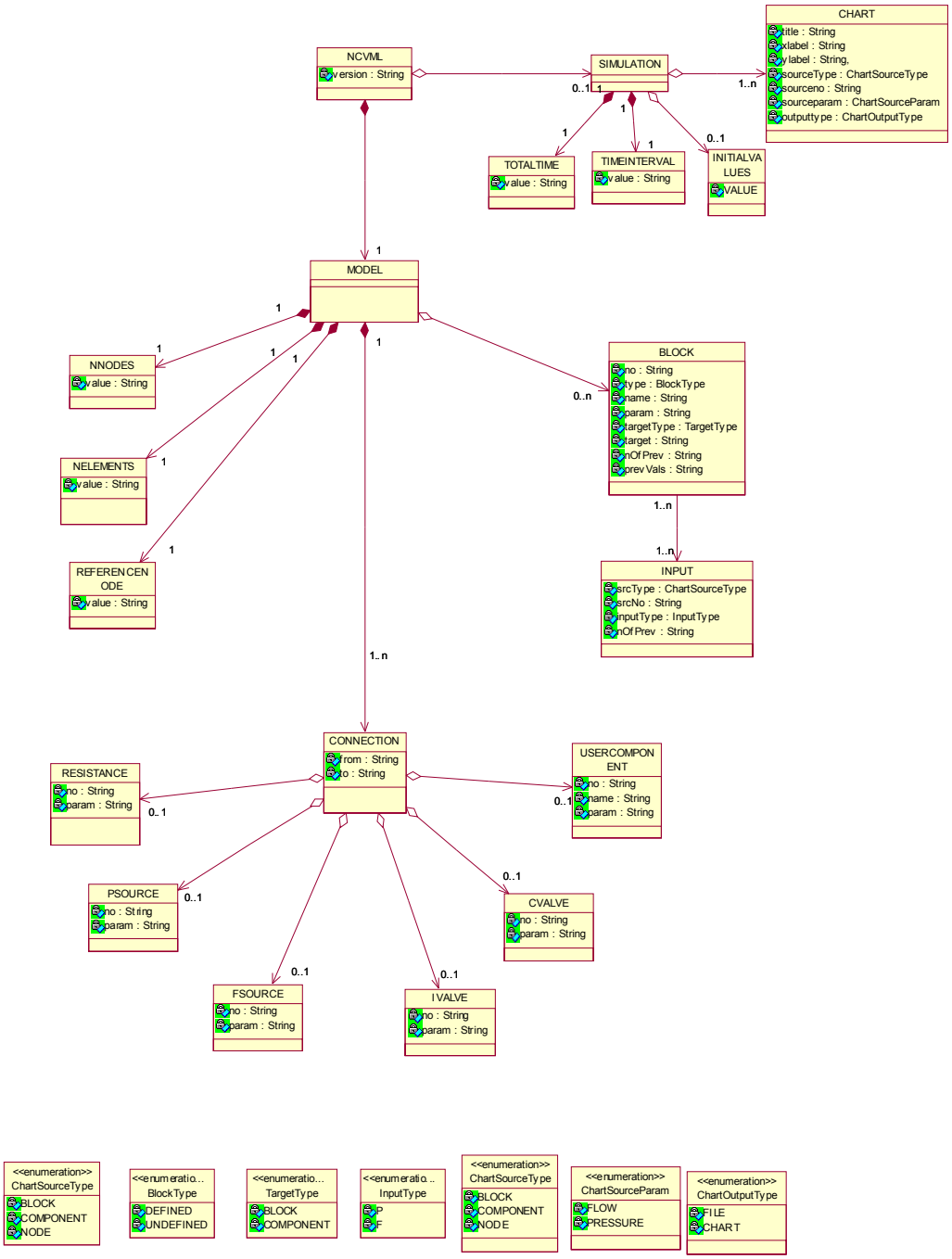


Figure 3.3 The UML Class Diagram of the NCVML model

3.6 Mapping UML Model to XML DTD

A UML model defines all classes, attributes, and relationships between classes required for an application vocabulary. They are used to understand the requirements of an application, its building blocks and the relation between them. However, UML models have no practical usage as a part of an XML application. To make use of the information contained in a UML model of an XML application, the components of the UML model are mapped to form a Data Type Definition (DTD). In another words, UML model is mapped to XML DTD. The purpose of a DTD is to define the legal building blocks of an XML vocabulary with their data types, relationships and usage restrictions.

The XML DTD specification [23] doesn't have an object-oriented structure. Therefore, there is no direct one-to-one method for mapping a UML model into DTD definitions. For this reason we followed the common and widely used mapping rules. While mapping a UML model, each class (except enumeration type) of the model maps to an XML element in the DTD definition. However, class attributes of the UML model might be either separate XML elements or element attributes. There is no specific rule for this decision. The designer must make a trade-off decision according to applications criteria. The composition relationship between classes is mapped to elements related as parent-child in a DTD. Similarly, association between to classes in a UML model is mapped as two elements with one of them is the container of the other. Finally, enumeration type class maps to an XML attribute and element that define this enumerated value list in the DTD. A simple illustration of these approaches, which we utilized for mapping, is depicted in Figure 3.4.

By using the rules explained above, we constructed our proposed DTD, which is given in Listing 3-1. Consider that we have used empty elements for all elements except containers and therefore let data to be carried with attributes. As we mentioned before, there is no strict rule for the decision of using elements with attributes or defining attributes as separated elements. We prefer to define them as attributes because elements with attributes look like more similar to object-oriented structure than attributes as separated elements. Due to the limitations of DTD, we have defined all data types of attributes as CDATA, which stands for string character data. But this is not a limitation in our case, because it decreases the complexity of NCVML document, which reduces the responsibility of users.

The following is the textual description of this DTD. NCVML DTD contains 25 elements. Each element stands for a tag in an XML document. The root element of an NCVML document is NCVML element. In a document, NCVML element must be followed by exactly one MODEL element and zero or one SIMULATION element. A MODEL element must include each of NELEMENTS, NNODES and REFERENCENODE elements. A MODEL element must be followed by at least one CONNECTION element and zero ore more BLOCK elements as well. CONNECTION element might be followed by any one of RESISTANCE, PSOURCE, FSOURCE, VALVE, CVALVE, USERCOMPONENT. But there might be different CONNECTION instances followed by different element instances. BLOCK element must be followed by at least one or more INPUT elements. SIMULATION element must be followed by exactly one SIMULATIONTIME, and exactly one TIMEINTERVAL elements, zero or one INITIALVALUES element and finally zero or more CHART element instances.

The root element NCVML has a compulsory string character data attribute “VERSION”. NNODES, NELEMENTS and REFERENCENODE elements all have an attribute, “VALUE”. These elements are so called empty elements which means they have XML structure like <NNODES VALUE = “” />. CONNECTION element has attributes “FROM” and “TO” which are compulsory. RESISTANCE, PSOURCE, FSOURCE, VALVE, CVALVE, USERCOMPONENT all have common attributes “NO” and “PARAM”. USERCOMPONENT has an additional attribute “NAME” as well. These are also empty elements. BLOCK element has attributes “NO”, “TYPE”, “NAME”, “PARAM”, “TARGETTYPE”, “TARGET”, “NOFPREV” and “PREVVALS”. “TYPE” and “TARGETTYPE” are enumerated type attributes. INPUT element has attributes “SRCTYPE”, “SRCNO”, “INPUTTYPE”, and “NOFPREV”. “SRCTYPE” and “INPUTTYPE” are enumerated type attributes. Finally, TOTALTIME and TIMEINTERVAL, INITIALVALUES elements are empty elements, which have one attribute “VALUE”

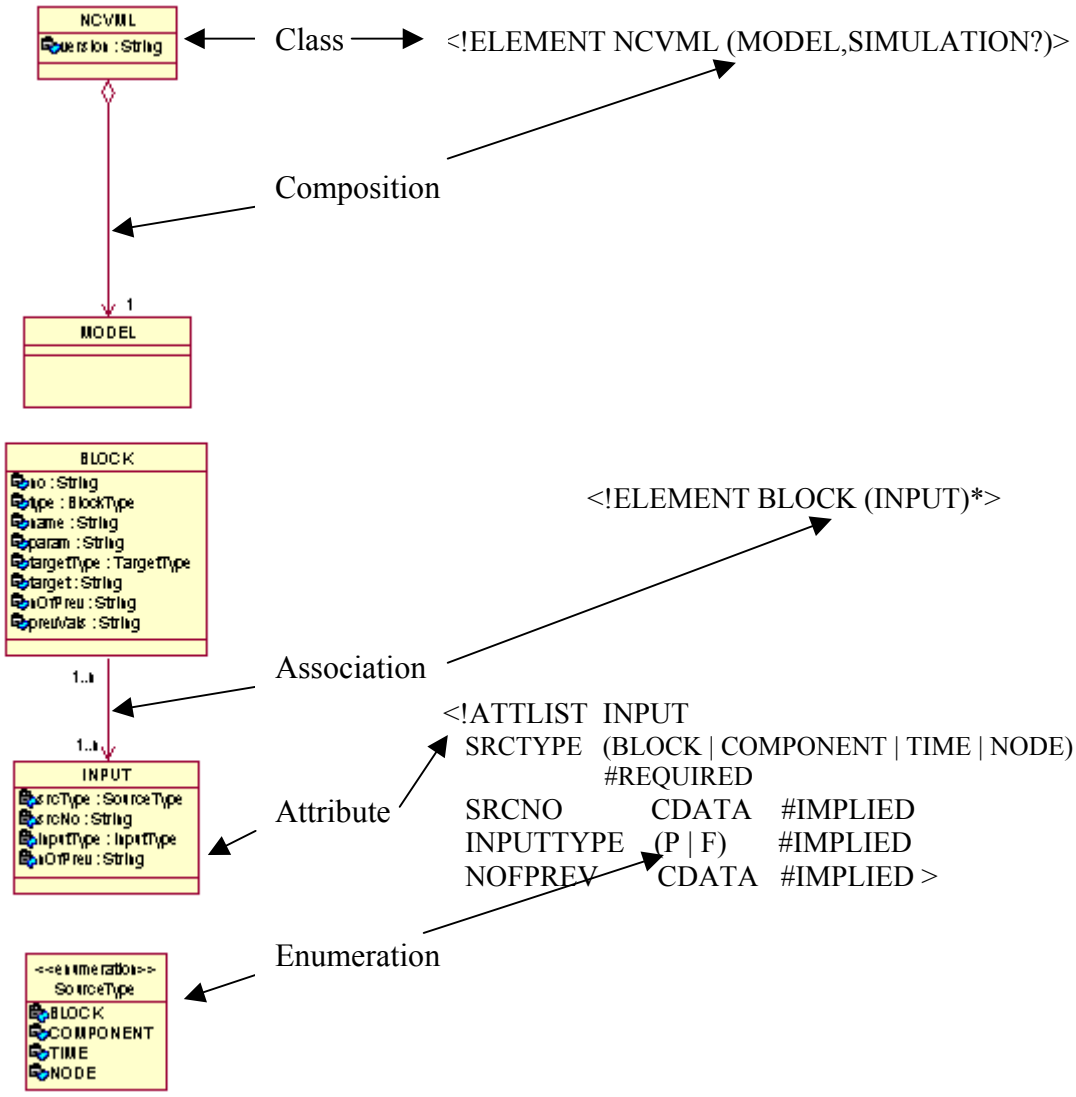


Figure 3.4 UML model to DTD mapping

```
<!ELEMENT  NCVML (MODEL,SIMULATION?)>
<!ELEMENT  MODEL

(NNODES,NELEMENTS,REFERENCENODE,CONNECTION+,BLOCK*)>

<!ELEMENT  NNODES                EMPTY>
```

```
<!ELEMENT  NELEMENTS            EMPTY>
<!ELEMENT  REFERENCENODE      EMPTY>

<!ELEMENT  CONNECTION
           (RESISTANCE|PSOURCE|FSOURCE|VALVE|CVALVE|USERBLOCK)
           >

<!ELEMENT  RESISTANCE        EMPTY>
<!ELEMENT  PSOURCE           EMPTY>
<!ELEMENT  FSOURCE           EMPTY>
<!ELEMENT  VALVE             EMPTY>
<!ELEMENT  CVALVE            EMPTY>
<!ELEMENT  USERBLOCK         EMPTY>
<!ELEMENT  BLOCK (INPUT)+>
<!ELEMENT  INPUT             EMPTY>

<!ELEMENT  SIMULATION (TOTALTIME,TIMEINTERVAL)>
<!ELEMENT  TOTALTIME         EMPTY>
<!ELEMENT  TIMEINTERVAL      EMPTY>
<!ELEMENT  INITIALVALUES     EMPTY>
<!ELEMENT  CHART             EMPTY>

<!ATTLIST  NCVML             VERSION   CDATA   #REQUIRED>
<!ATTLIST  NNODES            VALUE     CDATA   #REQUIRED>
<!ATTLIST  NELEMENTS         VALUE     CDATA   #REQUIRED>
<!ATTLIST  REFERENCENODE     VALUE     CDATA   #REQUIRED>

<!ATTLIST  CONNECTION        FROM      CDATA   #REQUIRED
                             TO        CDATA   #REQUIRED>

<!ATTLIST  RESISTANCE        NO        CDATA   #REQUIRED
                             PARAM     CDATA   #REQUIRED>
<!ATTLIST  PSOURCE           NO        CDATA   #REQUIRED
                             PARAM     CDATA   #REQUIRED>
<!ATTLIST  FSOURCE           NO        CDATA   #REQUIRED
```

```
                                PARAM      CDATA   #REQUIRED>
<!ATTLIST  VALVE                NO         CDATA   #REQUIRED
                                PARAM      CDATA   #REQUIRED>
<!ATTLIST  CVALVE               NO         CDATA   #REQUIRED
                                PARAM      CDATA   #REQUIRED>
<!ATTLIST  USERCOMPONENT        NO         CDATA   #REQUIRED
                                NAME       CDATA   #REQUIRED
                                PARAM      CDATA   #REQUIRED>

<!ATTLIST  BLOCK        NO              CDATA                    #REQUIRED
                        TYPE            (DEFINED|UNDEFINED)      #REQUIRED
                        NAME            CDATA                    #REQUIRED
                        PARAM           CDATA                    #IMPLIED
                        TARGETTYPE      (BLOCK|COMPONENT)        #REQUIRED
                        TARGET          CDATA                    #REQUIRED
                        NOFPREV         CDATA                    #REQUIRED
                        PREVVALS        CDATA                    #IMPLIED>

<!ATTLIST    INPUT    SRCTYPE          (BLOCK|COMPONENT|TIME|NODE)
#REQUIRED
                    SRCNO       CDATA                            #IMPLIED
                    INPUTTYPE   (P|F)                            #IMPLIED
                    NOFPREV     CDATA                            #IMPLIED>

<!ATTLIST  TOTALTIME        VALUE      CDATA          #REQUIRED>
<!ATTLIST  TIMEINTERVAL     VALUE      CDATA          #REQUIRED>
<!ATTLIST  INITIALVALUES    VALUE      CDATA          #REQUIRED>
<!ATTLIST  CHART            TITLE      CDATA          #IMPLIED
                            XLABEL     CDATA          #IMPLIED
                            YLABEL     CDATA          #IMPLIED
        SOURCETYPE    (NODE|COMPONENT|BLOCK)        #REQUIRED
                            SOURCENO CDATA            #REQUIRED
        SOURCEPARAM   (FLOW|PRESSURE)                 #REQUIRED
        OUTPUTTYPE    (CHART|FILE)                    #IMPLIED>
```

Listing 3.1. XML DTD of the NCVML model

3.7 Elements of NCVML

In this section, each of the building elements of NCVML is described in detail. Beside their description, the proposed XML encoding for each element is given as well. The general notation structure of the NCVML design is influenced by SBML encoding structure [24].

3.7.1 NCVML

This element is the root element of NCVML document. That is, it is the highest-level construct of an NCVML document. It represents the beginning of a NCV model simulation scenario. In each NCVML document only one instance of NCVML element is allowed. The UML definition of the NCVML structure is given in Figure 3.5.

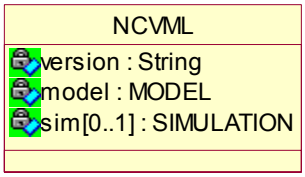


Figure 3.5: The UML definition of NCVML structure

It holds the compulsory attribute “VERSION” that holds which version is used to prepare the document. The current version of NCVML is 1.0, which refers to the first version. An NCVML object must contain exactly one *MODEL* element that is explained in 3.6.2 and may optionally contain exatly one *SIMULATION* component, which is decribed in 3.6.19.

In the transformation of UML model to XML, the NCVML structure turned into NCVML element. The following is an abbreviated example of the outermost content of an NCVML model definition in XML:

<NCVML VERSION = “1.0”>
  ...
</NCVML>

3.7.2 MODEL

MODEL component represents the beginning of a NCV model description. In other words, *MODEL* represents a bounded container in which NCV model is located. In each NCVML document only one instance of *MODEL* element is allowed. The UML definition of the *MODEL* structure is given in Figure 3.6.

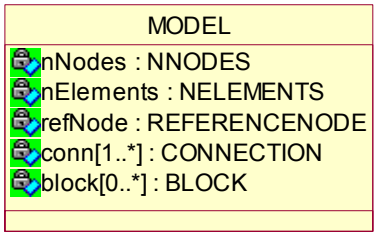


Figure 3.6: The UML definition of **MODEL** structure

A *MODEL* element has exactly one instance of each *NNODES, NELEMENTS, REFERENCENODE*, components, at least one or more instances of *CONNECTION* component and optionally one or more instances of *BLOCK* components.

In the XML encoding of the NCVML, MODEL structure turned into MODEL element. The following example illustrates the MODEL structure in XML:

<MODEL>
...
</MODEL>

3.7.3 NNODES

NNODES component represents the number of nodes in the mechanical cardiovascular model. In each instance of *MODEL* element, exactly one instance of *NNODES* is allowed. The UML definition of the *NNODES* structure is given in Figure 3.7.

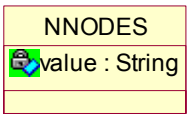


Figure 3.7: The UML definition of **NNODES** structure

NNODES component has an attribute “VALUE” of type String that holds the value of the number of nodes in the CV mechanical model. The value of VALUE attribute must be supplied by the user.

In the XML encoding of the NCVML, NNODES structure turned into empty NNODES XML element with an attribute VALUE. The following example illustrates the NNODES structure in XML:

<NNODES VALUE=”3” />

3.7.4 NELEMENTS

NELEMENTS component represents the total number of components in the CV mechanical model. In each instance of *MODEL* element, exactly one instance of *NELEMENTS* is allowed. The UML definition of the *NELEMENTS* structure is given in Figure 3.8.

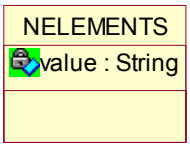


Figure 3.8: The UML definition of **NELEMENTS** structure

NELEMENTS component has an attribute "VALUE" of type String which holds the total number of components in the CV mechanical model. The value of VALUE attribute must be supplied by the user.

In the XML encoding of the NCVML, NELEMENTS structure turned into empty NELEMENTS XML element with an attribute VALUE. The following example illustrates the NELEMENTS structure in XML included in MODEL structure:

< NELEMENTS VALUE="4" />

3.7.5 REFERENCENODE

REFERENCENODE component represents the label of the node, which is to be accepted as reference node. In each instance of *MODEL* element, exactly

one instance of *REFERENCENODE* is allowed. The UML definition of the *REFERENCENODE* structure is given in Figure 3.9.

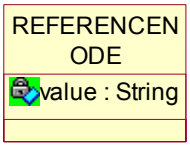


Figure 3.9: The UML definition of **REFERENCENODE** structure

REFERENCENODE component has an attribute "VALUE" of type String which holds the label of the reference node of the CV mechanical model. The value of VALUE attribute must be supplied by the user.

In the XML encoding of the NCVML, REFERENCENODE structure turned into empty REFERENCENODE XML element with an attribute VALUE. The following example illustrates the REFERENCENODE structure in XML included in MODEL structure:

< REFERENCENODE VALUE="2" />

3.7.6 CONNECTION

CONNECTION component represents a bounded container for each component of mechanical CV model. Thus, it is the logical abstraction for the structure between two edges of a component. *MODEL* element may contain at least one or more instance of *CONNECTION* component. Note that *CONNECTION* component is defined for describing the location of a component. The UML definition of the *CONNECTION* structure is given in Figure 3.10.

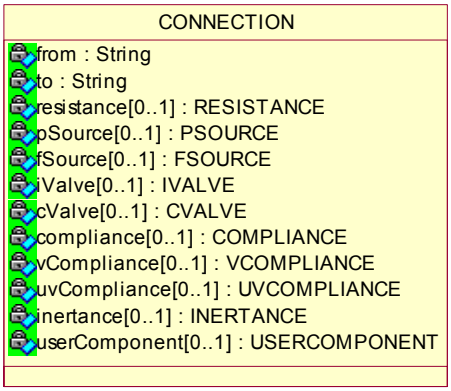


Figure 3.10: The UML definition of **CONNETION** structure

CONNECTION component must have exactly one CV model component object. These are *RESISTANCE, PSOURCE, FSOURCE, IVALVE, CVALVE, COMPLIANCE, VCOMPLIANCE, UVCOMPLIANCE, INERTANCE and USERCOMPONENT*. It has two compulsory attributes "FROM" and "TO" which hold the label of left and right edges of the contained component.

In the mapping UML to XML, CONNECTION structure turned into CONNECTIO XML element with two attributes FROM and TO. The following example illustrates the CONNECTION structure in XML.

< CONNECTION FROM="2" TO="3">

...

</CONNECTION>

3.7.7 RESISTANCE

RESISTANCE component represents the Flow Resistance of the CV mechanical model. In each *CONNECTION* component, only one instance of

RESISTANCE component can occur. However, more than one *RESISTANCE* object might occur throughout a NCVML document instance. The UML definition of *RESISTANCE* is given in Figure 3.11.

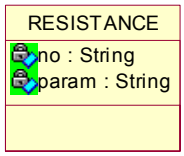


Figure 3.11: The UML definition of **RESISTANCE** structure

Each instance of a CV mechanical model component must be assigned a uniqe id throughout the model. For this reason, *RESISTANCE* component has an obligatory attribute "NO". The second attribute of *RESISTANCE* is PARAM, which holds the parameter values of the *RESISTANCE* object.

All components of the NCVML are envisaged to have the same structure to achieve generic component structure as follows: The number of parameters might be different for each component. One approach to pass parameter values might be defining each parameter staticly as an attribute for each component. Then in the description of the model each attribute of each component might be assigned a parameter value. But, in this approach there are equal to number of components different component descriptions. Thus, each component will have different description and each must be processed in a different way. This is not a convenient situation because of three reasons. First, this requires different behavior for each component. Second, this offers a nongeneric structure, which is a contradiction for our aim to propose a generic component oriented structure. Third, dynamic attribute definition is not possible with XML technology. This constitues a great bottleneck for

processing user defined components (user defined components will be explained in Section 3.7.16).

For parameter passing to components, NCVML model introduces an approach to pass all values of parameters as a string (CDATA in XML DTD notation) assigned to the attribute PARAM. The value of the PARAM string has to start with "[" and must end with "]" characters. The values of the parameters are to be inserted sequencially, leaving a blank character as a separator between parameter values.

In *RESISTANCE* case, as explained in the previous chapter, flow resistance component has one parameter, which is Resistance. Hence, the value of PARAM attribute must be as "[X]" where X is an arbitrary real number. The unit of the value assigned to Resistance parameter is supposed to be mmHg.sec/cc by default. At this time, the system supports only this unit for Resistance parameter.

In the XML encoding of the NCVML, RESISTANCE structure turned into empty RESISTANCE XML element with an attribute NO and PARAM. The following example illustrates the RESISTANCE structure in XML:

< RESISTANCE NO="2" PARAM="[5]"/>

3.7.8 PSOURCE

PSOURCE represents the Pressure Souce component of the CV mechanical model. PSOURCE component might occur more than once throughout a

NCVML document instance but can occur only once in each *CONNECTION* object. The UML definition of *PSOURCE* is given in Figure 3.12.

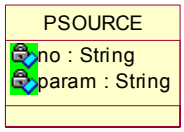


Figure 3.12: The UML definition of **PSOURCE** structure

As all other CV mechanical model components *PSOURCE* has two compulsory attributes “NO” and “PARAM”, that hold the unique component id and parameter values respectively. The component Pressure Source has one parameter, which is Pressure, so the PARAM value will have the structure like “[X]” where X is an arbitrary real number. The unit of the supplied value is expected to be mmHg by default.

In the XML encoding of the NCVML, PSOURCE structure turned into empty PSOURCE XML element with an attribute NO and PARAM. The following example illustrates the PSOURCE structure in XML:

< PSOURCE NO=”2” PARAM=”[98]”/>

3.7.9 FSOURCE

FSOURCE represents the Flow Source component of the CV mechanical model. *FSOURCE* component might occur more than once throughout a NCVML document instance but can occur only once in each *CONNECTION* object. The UML definition of *FSOURCE* is given in Figure 3.13.

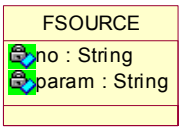


Figure 3.13: The UML definition of **FSOURCE** structure

FSOURCE has two compulsory attributes “NO” and “PARAM”, that hold the unique component id and parameter values respectively. The component Flow Source has one parameter, which is Flow, so the PARAM value will have the structure like “[X]” where X is an arbitrary real number. The default unit of the supplied value is expected to be cc/sec.

In the XML encoding of the NCVML, FSOURCE structure turned into empty FSOURCE XML element with an attribute NO and PARAM. The following example illustrates the FSOURCE structure in XML:

< FSOURCE NO=”1” PARAM=”[5]”/>

3.7.10 VALVE

VALVE represents the Valve component of the CV mechanical model. Two types of valve components are included in the NCVML. This element stands for the Valve, which works as a nonlinear diode as explained in the previous chapter. The second type of valve is explained in Section 3.7.11. *VALVE* component might occur more than once throughout a NCVML document instance but can occur only once in each *CONNECTION* object. The UML definition of *VALVE* is given in Figure 3.14.

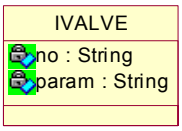


Figure 3.14: The UML definition of **VALVE** structure

VALVE has two compulsory attributes “NO” and “PARAM”, that hold the unique component id and parameter values respectively. The component Valve has two parameters, which are pressure threshold and valve’s saturation flow, respectively, so the PARAM value will have the structure like “[X Y]” where X and Y are arbitrary real numbers. The default unit of the supplied values are expected to be mmHg and cc/sec, respectively.

In the XML encoding of the NCVML, VALVE structure turned into empty VALVE XML element with an attribute NO and PARAM. The following example illustrates the VALVE structure in XML:
< VALVE NO=”2” PARAM=”[98 90]”/>

3.7.11 CVALVE

CVALVE represents the Compound Valve component of the CV mechanical model. This is the second type of valve component included in the NCVML. This element stands for the Compound Valve, which is the combination of a resistance and a diode as explained in the previous chapter. *CVALVE* component might occur more than once throughout a NCVML document instance but can occur only once in each *CONNECTION* object. The UML definition of *CVALVE* is given in Figure 3.15.

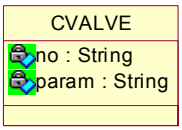


Figure 3.15: The UML definition of **CVALVE** structure

CVALVE has two compulsory attributes “NO” and “PARAM”, that hold the unique component id and parameter values respectively. The component Compound Valve has two parameters which are forward resistance R_{ON} and backward resistance R_{OFF}. Hence, the value of PARAM will have a structure like “[X Y]” where X and Y which are values of R_{ON} and R_{OFF}, respectively are arbitrary real numbers. The default unit of the supplied values are expected to be mmHg.sec/cc.

In the XML encoding of the NCVML, CVALVE structure turned into empty CVALVE XML element with an attribute NO and PARAM. The following example illustrates the CVALVE structure in XML:

< CVALVE NO=”2” PARAM=”[2 100]”/>

3.7.12 USERCOMPONENT

USERCOMPONENT represents the UserComponent component of the CV mechanical model. *USERCOMPONENT* component might occur more than once throughout a NCVML document instance but can occur only once in each *CONNECTION* object. The UML definition of *USERCOMPONENT* is given in Figure 3.16.

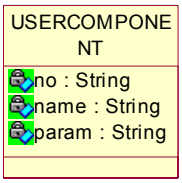


Figure 3.16: The UML definition of **USERCOMPONENT** structure

USERCOMPONENT has three compulsory attributes “NO”, “NAME” and “PARAM”, that hold the unique component id, the name of the user defined component and parameter values, respectively. User Component is designed for components to be described by user at the beginning of the simulation. Therefore, parameters of User Component are not defined until the user gives the definition using *USERCOMPONENT* element. Here, user inserts all values of parameters sequentially into PARAM string, leaving blank between them. With *USERCOMPONENT,* the user is responsible for taking into account the units of supplied parameters.

In the XML encoding of the NCVML, USERCOMPONENT structure turned into empty USERCOMPONENT XML element with an attribute NO and PARAM. The following example illustrates the USERCOMPONENT structure in XML:

<USERCOMPONENT NO=”5” NAME=”UserComponent” PARAM=”[1 16 -0.1]”/>

3.7.13 BLOCK

BLOCK represents the Block components of the baroregulation feedback mechanism. *BLOCK* component might occur more than once throughout an

NCVML document instance. The UML definition of *BLOCK* is given in Figure 3.17.

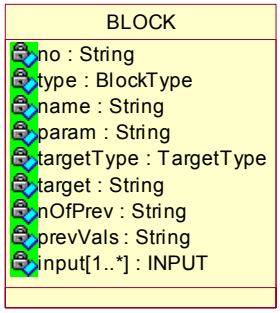


Figure 3.17: The UML definition of **BLOCK** structure

BLOCK element has 7 attributes. These are “NO”, ”TYPE”, “NAME”, “PARAM”, “TRGTYPE”, “TARGET”, “NOFPREV”, “PREVVALS”, respectively. From these attributes, “PARAM” and “PREVVALS” are optional and all others are compulsory attributes. “NO” describes the unique component id throughout the NCVML model. *BLOCK* elements have to be assigned a unique number as well. These ids are important in describing the component. The output of a Block component can be the input of another Block component. Therefore, to describe the input of a block we might have to know the id of the source block. “TYPE” attribute identifies the Block component as being in the system library or to be supplied by the user at the beginning of the simulation. It can take two values; “DEFINED” or “USERDEFINED”, respectively. “DEFINED” states that this bock takes place in the system library, and “USERDEFINED” states that the name and implementation of this block component is to be supplied by the user. “NAME” represents the name of the block instance. If the block is a “DEFINED” block it might take one of the following names given in Table 3.1. These are names of currently available block implementations of the system library. Each of these implementations will be described in the following chapter (Chapter 4). If the block is

"USERDEFINED", this value must be user-supplied string, specifying the name of the block implementation the user supplies.

BSLV	BAbsoluteDelay
BVSLV	BFilter
Bsinus	BAx_Plus_b
BstepFunction	BInverter
BsquareFunction	BPulseFunction
BintegrateAndFire	BSSBaroReflex
Bursino	BcomplianceFeedback
BComplianceUVFeedback	BInertanceFeedback

Table 3.1. List of currently available block components of the system library

"PARAM" represents the list parameters that the block implementation needs. A block might take parameters or might need no parameter. For this reason, this attribute is optional. The structure of the "PARAM" is the same as previous elements. "TRGTYPE" specifies the target of the block. As stated earlier, the output of a block might be either input to another block or it effects the parameter of a mechanical plant component. The set of values it can take contains "BLOCK" and "COMPONENT" that state the output is input to another block and the output modifies the parameter of a mechanical plant component, respectively. "TARGET" specifies the unique id of the component that the output effects. If "TRGTYPE" is "BLOCK", the value of "TARGET" specifies the id of the next block that takes the output of this block as input. Similarly, if "TRGTYPE" is "COMPONENT", the value of "TARGET" specifies the id of the component that the output of this block modifies its parameter. Consider that "TARGET" might take more than one value separated by comma, which means that this block has more than one target of

the same type. “NOFPREV” specifies the number of previous outputs required as input to the block at each step. If “NOFPREV” is more than 0, the initial values must be supplied in “PREVVALS” in the same structure as “PARAM” attribute.

BLOCK element might include one or more *INPUT* element instance in an NCVML document. *BLOCK* is a bounded container of *INPUT* element instances. *INPUT* element is described in detail in 3.7.14.

In the XML encoding of the NCVML, BLOCK structure turned into empty BLOCK XML element with an attributes NO, TYPE, NAME, PARAM, TRGTTYPE, TARGET, NOFPREV, and PREVVALS. The following example illustrates the BLOCK structure in XML:

<BLOCK NO = "3" TYPE = "DEFINED" NAME = "BVSLV" PARAM = "[67/1332 2500/1332 0.3]" TRGTYPE = "BLOCK" TARGET = "4" NOFPREV = "0">
…
</BLOCK>

3.7.14 INPUT

INPUT represents input of Block components of the baroregulation feedback mechanism. In each *BLOCK* element instance more than one *INPUT* element instance might occur in an NCVML document instance. The UML definition of *INPUT* is given in Figure 3.18.

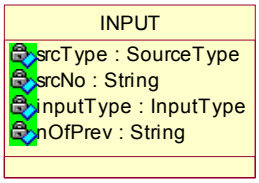


Figure 3.18: The UML definition of INPUT structure

INPUT element has 4 attributes. These are “SRCTYPE”, “SRCNO”, ”INPUTTYPE”, and ”NOFPREV”. SRCTYPE represents the type of the input source. Input of a block might be either a component parameter, a block output, a node, or current time. SRCTYPE might take one of “COMPONENT”, “BLOCK”, “NODE”, and “TIME” strings. As their names imply, they specify the input as a parameter of a component, output of a block, a node, or time, respectively. SRCNO identifies the unique id of the source component or block. INPUTTYPE identifies the type of the input. Input type can be “P” or “V” in case the input source is component or node and can be null in case the input source is block output. NOFPREV holds the previous values of the input if needed. The structure of the NOFPREV is the same as the structure of PARAM attribute.

In the XML encoding of the NCVML, INPUT structure turned into empty INPUT XML element with an attributes SRCTYPE, SRCNO, INPUTTYPE, and NOFPREV. The following example illustrates the INPUT structure in XML:

<INPUT SRCTYPE="BLOCK" SRCNO="3" NOFPREV = "0" />

3.7.15 SIMULATION

SIMULATION is a bounded container of simulation parameters. It is the same level element as *MODEL* element. In a NCVML document only one instance of *SIMULATION* instance is allowed. The UML definition of *SIMULATION* is given in Figure 3.23.

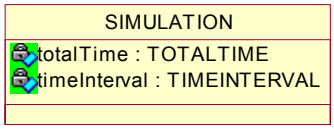


Figure 3.19: The UML definition of **SIMULATION** structure

SIMULATION element might include zero or more *TOTALTIME* and *TIMEINTERVAL* elements instances.

In the XML encoding of the NCVML, SIMULATION structure turned into SIMULATION XML element. The following example illustrates the SIMULATION structure in XML:

<SIMULATION>
…
</SIMULATION>

3.7.16 TOTALTIME

TOTALTIME represents the total time of the simulation. In the *SIMULATION* instance only one instance of *TOTALTIME* instance is allowed. The UML definition of *TOTALTIME* is given in Figure 3.20.

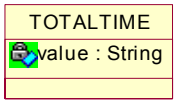


Figure 3.20: The UML definition of **TOTALTIME** structure

TOTALTIME element has a compulsory attribute “VALUE”. The total simulation time should be provided in seconds as value of VALUE attribute.

In the XML encoding of the NCVML, TOTALTIME structure turned into empty TOTALTIME XML element with an attribute VALUE. The following example illustrates the TOTALTIME structure in XML:

<TOTALTIME VALUE=”5” />

3.7.17 TIMEINTERVAL

TIMEINTERVAL represents the time interval between two steps of the simulation. In the *SIMULATION* instance only one instance of *TIMEINTERVAL* instance is allowed. The UML definition of *TOTALTIME* is given in Figure 3.21.

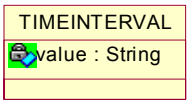


Figure 3.21: The UML definition of **TIMEINTERVAL** structure

TIMEINTERVAL element has a compulsory attribute “VALUE”. The time interval between two steps should be provided in seconds as value of VALUE attribute.

In the XML encoding of the NCVML, TIMEINTERVAL structure turned into empty TIMEINTERVAL XML element with an attribute VALUE. The following example illustrates the TIMEINTERVAL structure in XML:

<TIMEINTERVAL VALUE = “0.01” />

3.7.18 INITIALVALUES

INITIALVALUES represents the numerical values for initial values of the set of non-linear equations variables. In the *SIMULATION* instance only one instance of *INITIALVALUES* instance is allowed. The UML definition of *INITIALVALUES* is given in Figure 3.22.

Figure 3.22: The UML definition of *INITIALVALUES* structure

INITIALVALUES element has a compulsory attribute “VALUE”. Initial values of the variables in the sequence of flow and pressures should be provided as an array as value of VALUE attribute.

In the XML encoding of the NCVML, *INITIALVALUES* structure turned into empty *INITIALVALUES* XML element with an attribute VALUE. The following example illustrates the *INITIALVALUES* structure in XML:

< INITIALVALUES VALUE = “[0.01 1 -3 10 0]” />

3.7.19 CHART

CHART represents the output of the desired parameter of the simulation result. Thus, users can describe the type of the output and the parameters that he/she wants to observe its results. In the *SIMULATION* instance one or more instances of *CHART* instances are allowed. The UML definition of *CHART* is given in Figure 3.23.

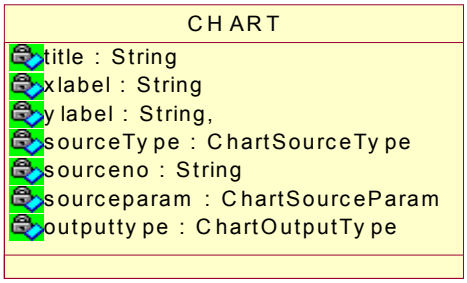


Figure 3.23: The UML definition of *CHART* structure

CHART element has compulsory attributs “SOURCETYPE”, “SOURCENO”, “SOURCEPARAM” and optional attributes “TITLE”, “XLABEL”, “YLABEL”, and “OUTPUTTYPE”.

In the XML encoding of the NCVML, *CHART* structure turned into empty *CHART* XML element with attributes SOURCETYPE, SOURCENO, SOURCEPARAM, TITLE, XLABEL, YLABEL, and OUTPUTTYPE. SOURCETYPE describes the type of the source to be component, node or block. A user might ask to observe pressure value of a node, pressure difference between two nodes (pressure on a component), flow through a component and block output. Thus, SOURCETYPE may take value of either “NODE”, “COMPONENT”, or “BLOCK”. SOURCENO describes the unique number of the node, component or block. SOURCEPARAM might be

“FLOW” if the source is component, “PRESSURE” if the source is component or node, and “OUTPUT” if the source is block. The output can be displayed in two ways: a line chart or as values series in a text file. The default format is displaying chart. However users may request the output values in a text file by providing “FILE” value to the SOURCETYPE attribute. If the output is a chart user can provide title for the chart with TITLE, label the x-axis with XLABEL, and finally label the y-axis with YLABEL attribute.

The following example illustrates the *CHART* structure in XML:

<CHART TITLE=”Pressure vs Time” XLABEL=”Pressure”
YLABEL=”Time” SOURCETYPE=”BLOCK” SOURCENO=”1”
SOURCEPARAM=”OUTPUT”>

3.7.20 Description of other components

3.7.20.1 Compliance Description

As we have mentioned before, compliance is modeled as a pressure source for short time intervals. However, the parameter of the pressure source is updated at the end of each step. To provide this modeling, we proposed the compliance component as a pressure source and a feedback block to the parameter of this pressure source. Thus, the XML description of compliance is the combination of PSOURCE and BLOCK of type “BComplianceFeedback” as follows:

```
<CONNECTION FROM="1" TO="7">
  <PSOURCE NO="7" PARAM ="[99.8]"/>
</CONNECTION>
```

.

.

<BLOCK NO = "1" TYPE = "DEFINED" NAME = "BComplianceFeedback" PARAM = "[4 99.8]" TRGTYPE = "COMPONENT" TARGET = "7" PARAMNO="1" NOFPREV = "1" PREVVALS="[99.8]">

<INPUT SRCTYPE="COMPONENT" SRCNO="7" INPUTTYPE="FLOW" NOFPREV="0" />

</BLOCK>

Here, the output of the block updates the parameter value of the pressure source each iteration. Following this way, the generic structures of the framework and the NCVML are preserved. Alternatively, the functionality of the block could have been included into the source code of the system, which causes a contradiction to the general idea of the system.

3.7.20.2 Variable Compliance Description

XML description of the variable compliance component constitutes a pressure source description and a feedback block to this pressure source. Thus, it constitutes PSOURCE description and BLOCK of type BVariableComplianceFeedback as follows:

<CONNECTION FROM="6" TO="7">

<PSOURCE NO="12" PARAM="[4]"/>

</CONNECTION>

.

.

<BLOCK NO = "7" TYPE = "DEFINED" NAME = "BVariableComplianceFeedback" PARAM = "[1332/67 4]" TRGTYPE = "COMPONENT" TARGET = "12" PARAMNO="1" NOFPREV = "1" PREVVALS="[4]">
<INPUT SRCTYPE="COMPONENT" SRCNO="12" INPUTTYPE="FLOW" NOFPREV="0" />
<INPUT SRCTYPE="BLOCK" SRCNO="6" NOFPREV = "1" PREVVALS="[1332/67]"/>
</BLOCK>

3.7.20.3 Compliance With Unstressed Volume Description

XML description of the compliance with unstressed volume component constitutes a pressure source description and a feedback block to this pressure source. Thus, it constitutes PSOURCE description and BLOCK of type BComplianceUVFeedback as follows:

<CONNECTION FROM="3" TO="14">
<PSOURCE NO="4" PARAM="[7]"/>
</CONNECTION>
.
.
<BLOCK NO = "3" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM = "[2.05 7 274.4]" TRGTYPE = "COMPONENT" TARGET = "4" PARAMNO="1" NOFPREV = "1" PREVVALS="[7]">
<INPUT SRCTYPE="COMPONENT" SRCNO="4" INPUTTYPE="FLOW" NOFPREV="0" />
</BLOCK>

3.7.20.4 Inertance Description

As we have mentioned before, inertance is modeled as a flow source for short time intervals. However, the parameter of the flow source is updated at the end of each step. To provide this modeling, we proposed the description of inertance component as a flow source and a feedback block to the parameter of this flow source. Thus, the XML description of inertance is the combination of FSOURCE and BLOCK of type “BInertanceFeedback” as follows:

```
<CONNECTION FROM="1" TO="2">
    <FSOURCE NO="1" PARAM="[2.2/10000]"/>

</CONNECTION>
.
.
<BLOCK NO = "1" TYPE = "DEFINED" NAME = "BInertanceFeedback" PARAM = "[0.22/1000 0]" TRGTYPE = "COMPONENT" TARGET = "1" PARAMNO="1" NOFPREV = "1" PREVVALS="[0]">
        <INPUT SRCTYPE="NODE" SRCNO="1" INPUTTYPE="PRESSURE" NOFPREV="0" />
        <INPUT SRCTYPE="NODE" SRCNO="2" INPUTTYPE="PRESSURE" NOFPREV="0" />
</BLOCK>
```

3.8 Simple Example

In previous sections, we have described our proposed NVCML model, identified its elements and described their XML implementation. To make

concepts more clear, in this section we will give a simple NCVML encoding example.

Consider the hypothetical model given in Figure 3.1. To be able to start encoding this model, we have to first assign components and blocks unique numbers. Then we have to assign unique numbers to nodes. Following this, we have to define connections of the model. At this time, the framework supports numbering only starting from 1 and continuing sequentially.

Let's start with assigning the flow source the unique number 1 and continue clockwise incrementing one-by-one. Therefore, assigned values of component numbers will be as follows: flow source (F): 1, compliance (Ca): 2, resistance (R): 3, compliance (Cb): 4
Note that, blocks must be assigned separate numbers from components. Thus, conventionally, the sequence of numbers should reset to 1 before assigning numbers to blocks. For this reason we assign the number 1 to BlockR.

The second step is assigning unique number to the nodes of the model. Let's assign 1 to Node1, 2 to Node2, and 3 to Node3. The third step is defining connections of the model. Let's start with the component with number 1 which is flow source. Labels (numbers) of left and right edges of this component are 3 and 1, respectively. Therefore, the connection that contains component 1 starts from node 3 and ends at node 1. Similarly, second connection starts from node 1 and ends at node 3, third connection starts from node 1 and ends at node 2, and finally the last connection starts from node 2 and end at node 3. Note that we define connections only for components. By definition, we don't define connections for blocks.

The last task is determining component parameters, simulation parameters and input and output of the block. As this a hypothetical model is, we don't feel very obliged to assign real parameter values. A comprehensive example with a scientific model and meaningful parameter will be provided in Chapter 5.

The XML encoding of this model using NCVML might be as follows:

```
<NCVML VERSION = "1.0">
<MODEL>

<NNODES VALUE = "3" />
<NELEMENTS VALUE = "4" />
<REFERENCENODE VALUE = "3" />

<CONNECTION FROM = "3" TO = "1">
        <FSOURCE NO = "1" PARAM = "[10]">
</CONNECTION>
<CONNECTION FROM = "1" TO = "3">
        <PSOURCE NO = "2" PARAM = "[7.5]">
</CONNECTION>
<CONNECTION FROM = "1" TO = "2">
        <RESISTANCE NO = "3" PARAM = "[10]">
</CONNECTION>
<CONNECTION FROM = "2" TO = "3">
        <PSOURCE NO = "4" PARAM = "[7.5]">
</CONNECTION>
```

```
<BLOCK NO =”1” TYPE=”DEFINED” NAME = “BAx_Plus_b” PARAM =
“[2 3]” TRGTTYPE=”COMPONENT”  TARGET = “3” NOFPREV = “0”>
        <INPUT SRCTYPE=”NODE” SRCNO=”1” INPUTTYPE = “P” />
</BLOCK>
<BLOCK NO =”2” TYPE=”DEFINED” NAME = “BComplianceFeedback”
PARAM = “[18 7.5]” TRGTTYPE=”COMPONENT” TARGET = “2”
NOFPREV = "1" PREVVALS="[0]" >
        <INPUT SRCTYPE="COMPONENT" SRCNO="2"
INPUTTYPE="FLOW" NOFPREV="0" />
</BLOCK>
<BLOCK NO =”3” TYPE=”DEFINED” NAME = “BComplianceFeedback”
PARAM = “[10 7.5]” TRGTTYPE=”COMPONENT” TARGET = “4”
NOFPREV = "1" PREVVALS="[0]" >
        <INPUT SRCTYPE="COMPONENT" SRCNO="4"
INPUTTYPE="FLOW" NOFPREV="0" />
</BLOCK>
</MODEL>

<SIMULATION>
        <TOTALTIME VALUE = “10” />
        <TIMEINTERVAL VALUE = “0.01” />
          <CHART TITLE=”Pressure vs Time” XLABEL=”Pressure”
      YLABEL=”Time”   SOURCETYPE=”BLOCK” SOURCENO=”1”
                        SOURCEPARAM=”OUTPUT”>
</SIMULATION>
</NCVML>
```

Chapter 4

The Simulator NCVJSim

4.1. Overview

NCVJSim is the actual simulator part of our proposed framework for NCV model simulation. It performs numerical computations to obtain numerical results of model simulations. Thus, when a model description is provided to NCVJSim package, it converts the description into its internal structures, following this it uses numerical methods to simulate the model and returns obtained numerical results at the end of the simulation. Consider that, model descriptions are provided to the NCVJSim by the NCVML parser and renderer, called NCVMLDOMParser, which will be examined in detail in the next chapter.

In its most general term, NCVJSim is a bundle of Java classes. It is completely object-oriented and it constitutes Java classes to perform numerical simulations of NCV models. These classes can be grouped into two sub-packages: 1) the Simulator class, which performs the numerical simulation approach 2) library classes, which are Java class implementations of NCV model component and block functions. The Simulator class, named

NewtonRaphson.class recieves a model description, constructs set of nonlinear equations from them, and applies Newton-Raphson method to iteratively solve this equations. During this process, it creates instances of library classes and invokes required methods of them. On the other hand, the library includes two types of classes. First type of classes are NCV model component implemenations, where the second type is NCV model block implementations. In Figure 4.1, the UML class diagram of the NCVJSim package is depicted.

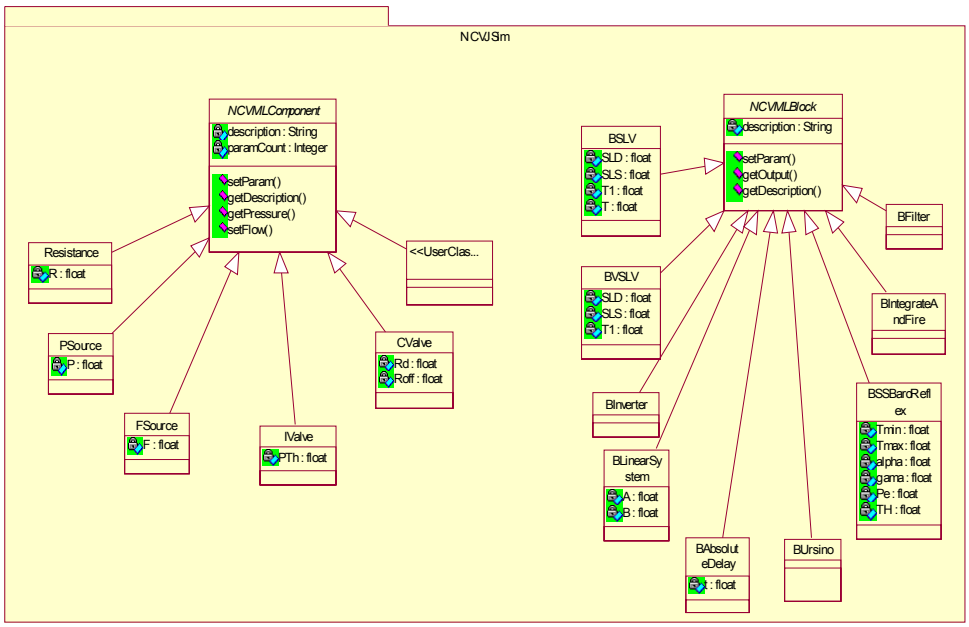


Figure 4.1. UML Class Diagram (inheritance) of the NCVJSim package

As well as static library classes, NCVJSim enables dynamic class loading during run-time. This facility is achieved by application of Java Dynamic Class Loading & Linking [25], and Java Reflection mechanisms [26]. The main feature of these mechanisms is giving the opportunity of incorporating classes at run-time without requiring recompilation. Moreover,

NCVJSim is extensible. More classes can be added to the package to enhance its functionality. But this must be done by an administrator on a PC and requires recompilation of the whole package.

The rest of this chapter is structured as follows: in section 4.2 the simulation approach will be described in detail by explaining underlying techniques and applied methods. In this context, Newton Raphson method is explained in detail in section 4.2.1. Following this, in section 4.3, each element of NCVJSim library is examined with special emphasis on their Java implementations. Section 4.4, discusses the Dynamic Loading and Linking mechanism, which adds a great flexibility to the framework. And finally, section 4.5 describes the implementation of Java Reflection Mechanism in our framework.

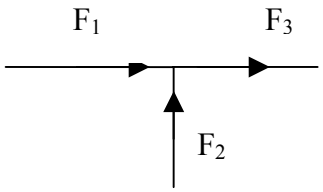
4.2 Simulation technique

Simulation process starts with obtaining a set of equations from the model description. For this purpose, description of mechanical model elements and their interconnections are converted into internal data structures and from the analysis of this data structures a set of equations is derived. The derivation process is simply based on the application of:

i) Node equations,
ii) Element equations.

Node equation states that the sum of the flow into a node must be zero. Alternatively, the sum of the flow out of a node must equal the sum of the flow into the node. We can formulate this statement as:

$\Sigma Fout_i = \Sigma Fin_i$ or $\Sigma Fout_i - \Sigma Fin_i = 0$, for all i = 1, 2, ...n where n is the node number.



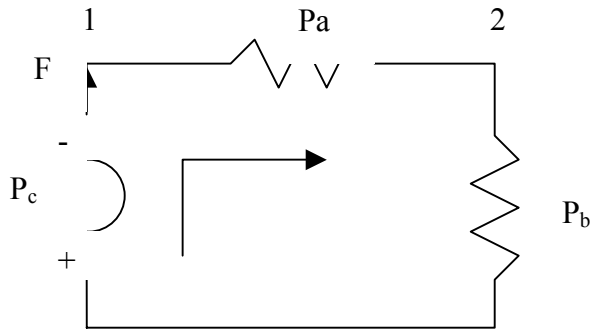
Node equation:

$F_1 + F_2 - F_3 = 0$

Figure 4.2 Node equation example

Furthermore, an element equation states that the sum of the pressures around any closed loop in a circuit is zero. Thus,

$\sum_{i=1}^{n} P_i = 0$, where n is the number of elements in the closed loop.



Element equation:

$P_1 - P_2 = Pa$

Figure 4.3 Element equation example

Utilizing the above rules, the simulator derives *n* element equations and *m* node equations, where *n* is the number of mechanical model elements, and *m* is node number minus one. The aim of deriving a set of equations is to form a linear or nonlinear set of equations of the form Ax = b, thus to transform the problem into a format that is numerically solvable. The set of equations might be either linear or nonlinear depending on the characteristics of circuit elements. But especially in our case, there are nonlinear elements like compliance, inductance, valve, etc. And these are frequently taking place in models. Therefore, as a broader approach, the solution of a model is carried out through constructing and solving a set of nonlinear equations.

Generation of node and element equations is described in the following subsection.

i) node equations: as stated earlier, node equations are based on the rule that the total flow flowing in a node is equal to the flow flowing out of a node. Therefore, a node equation will be in the following form:

$a_1F_1 + a_2F_2 + \ldots. + a_kF_k = 0$, where k = total number of branches connected to the node. Furthermore, a_i =+1 if the branch is an incoming branch and a_i = -1 if the branch is an outgoing branch. Therefore, the corresponding row vector will have value 1 where an element has connection as an incoming branch to the node, -1 where an element has a connection as an outgoing branch to the node, and finally 0 where an element has no connection to this node. Generaly speaking, the pseudo-algorithm that derives node equations is as follows:

for *node*=1 to number of nodes – 1 loop

    for *element* = 1 to number of elements loop

if the *element* is connected to an incoming branch of the *node* then

array(*node*,*element*) = 1

else if the *element* is connected to an outgoing branch of the *node* then

array(*node*,*element*) = -1

else if the element has no connection to the node

array(*node*,*element*) = 0

end loop

end loop

ii) element equations: as stated earlier, element equations are based on the rule that in a closed loop the sum of pressure arround the loop is zero. Thereby, the sum of all element pressures of the cloosed loop is zero. We can utilize this rule to derive pressure equations for each element between its nodes. Consequently the following equation is valid for all circuit elements:
$P_a - P_b - P_e = 0$, where P_a is the pressure on the starting-edge of the element, P_b is the pressure on the ending-edge of the element, and P_e is the pressure on the element.

Here P_e depends on the type of the element and the flow over this element. Besides, the relation to calculate pressure which is in the form of $P_e = f(F_e)$ is different for each element. Table 4.1 gives the list of relations (called branch constitutive relations) for all NCV circuit model elements.

Resistance	P = F*R	F = P/R
Pressure Source	P = Ps	F = N/A
Flow Source	P = N/A	F = Fs
Ideal Valve	P = Pth*ln(F/Fo + 1)	F = Fo*($e^{P/Pth}$ - 1)
Compound Valve	P = Rd*F if F ≥ 0 P = Roff*F if F < 0	F = P/Rd if P≥0 F = P/Roff if P<0
Compliance (*)	P = Ps	F = N/A
Variable Compliance (*)	P = Ps	F = N/A
Compliance with Unstressed Volume (*)	P = Ps	F = N/A
Inertance (**)	P = N/A	F = Fs
User Component	P = N/A (Defined at run-time)	F = N/A (defined at run-time)

Table 4.1 Branch Constitutive Relations.

Actually, the general equations for (*) and (**) are F = C*dP/dt and P = I*dF/dt respectively. But in our framework, we make an assumption that (*) act as a pressure source and (**) act a flow source for very small time intervals. At the end of each step, the pressure value of (*) is updated according to the formula, P(i+1) = P(i) + F(i)*dt/C(i) where i represents the i^{th} step, i+1 represents the $i+1^{th}$ step, and dt represents the step time interval. Similarly, the flow parameter of (**) is updated at the end of each step according to the formula F(i+1) = F(i) + P(i)*dt/L(i).

We have developed an object-oriented architecture that enables obtaining the instant pressure value of any element just by calling generic method getPressure() which is implemented in all element implementations. getPressure() method should also be provided with the flow over current element. This gives us the flexibility to derive element equations at any time of simulation. Following previous discussions, the corresponding array row of equations will have value of +1 where an element has a starting-edge at node a, -1 where an element has an ending-edge at node b. Besides, the equation

also containes a part Pe as we discussed above. But here, instead of locating a static value in the corresponding column, we can imagine as if we left it empty and we obtain P_e when we need it by calling related object's getPressure method and passing the flow as parameter. We can summarize this process with the following algorithm:

for *element* = 1 to number of elements loop
    a = starting-edge of *element*
    b = ending-edge of *element*
    array(number of nodes + element, number of nodes + a) = +1;
    array(number of nodes + element, number of nodes + b) = -1;
end loop

and, during simulation iteratations, P_e is calculated as follows:

for *element* = 1 to number of elements loop
        remainder = P_a – P_b – *element*.getPressure(F_e)
end loop

We will discuss the meaning of the algorithm given above in detail in the following section.

After we write node and element equations, we obtain a set of nonlinear equations of the following form:

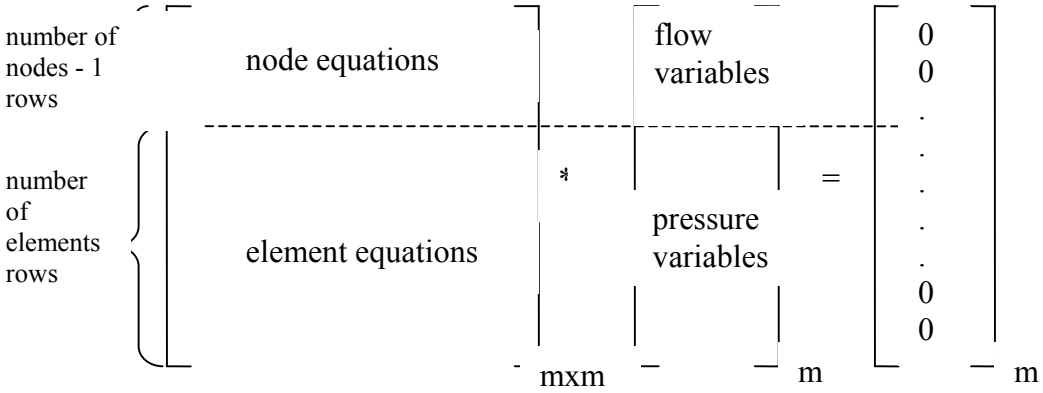
F_1 (F_1, F_2…F_k, P_1, P_2…P_l) = 0
F_2 (F_1, F_2…F_k, P_1, P_2…P_l) = 0
.
.

.

$F_m (F_1, F_2 \ldots F_k, P_1, P_2 \ldots P_l) = 0$

where k = number of nodes-1, l = number of elements, and $m = k + l$

Moreover, solution of this nonlinear set of equations is carried out by numerically solving the following system.



The simulation of a model for a given time interval is carried out first by deriving a set of nonlinear equations of the above form. Following this, nonlinear system is solved for small time intervals with time increments. This processed is repetead until simulation interval is completed.

For solving systems of nonlinear equations with more than one nonlinear equation, unfortunately there are no general and perfect methods [27]. System of nonlinar equations cannot be solved without an initial guess or some specific information. Solution of system of nonlinear equations is also known as “root finding” problem. In our simulation, we used one of the simplest and most powerful method, Newton-Raphson method for

multidimensional root finding. This method provides a very efficient means of converging to a root, if a sufficiently good initial guess is provided. But it harbours the risk that, it can also spectacularly fail to converge, indicating (but doesn't prove) that root does not exist nearby the provided initial guess [27].

4.2.1 Newton-Raphson Method

Let denote variables of nonlinear equations with vector x. A typical problem gives *N* functional relations to be zeroed, involving variables

x_i, $i = 1, 2, \ldots, N$:

$Fi(x_1, x_2, \ldots, x_N) = 0 \qquad i = 1, 2, \ldots, N.$ *(4.1)*

Let **x** denote the entire vector of values *xi* and **F** denote the entire vector of functions *Fi*. In the neighborhood of **x**, each of the functions *Fi* can be expanded in Taylor series

$$F_i(\mathbf{x} + \delta\mathbf{x}) = F_i(\mathbf{x}) + \sum_{j=1}^{N} \frac{\partial F_i}{\partial x_j}\delta x_j + O(\delta\mathbf{x}^2). \qquad (4.2)$$

The matrix of partial derivatives appearing in equation (4.2) is the *Jacobian* matrix **J**:

$$J_{ij} \equiv \frac{\partial F_i}{\partial x_j}. \qquad (4.3)$$

In matrix notation equation (4.2) is

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2). \qquad (4.4)$$

By neglecting terms of order $\delta\mathbf{x}^2$ and higher and by setting $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$, we

obtain a set of linear equations for the corrections $\delta\mathbf{x}$ that move each function closer to zero simultaneously, namely

$$\mathbf{J} \cdot \delta\mathbf{x} = -\mathbf{F}. \qquad (4.5)$$

Matrix equation (4.5) can be solved by *LU or QR decomposition* and the corrections are then added to the solution vector,

$$\mathbf{x}_{new} = \mathbf{x}_{old} + \delta\mathbf{x} \qquad (4.6)$$

and the process is iterated to convergence.

The application of Newton-Raphson method requires the function values to be calculated at 2 points for each iteration. Thus, **F(x)** and **F(x+h)** are calculated where h = δ**x.** For this purpose, we have implemented the following function that calculates the values of node and element equations at given points:

```
public float[] calculateFunctionValues(float[] x0)
```

Here as the definition of the function specifies, the function calculateFunctionValues recieves a floting point array, which holds the variable values and returns function values at given variable values in a float array.

The algorithmic approach of Newton-Raphson method for each step (time interval) is as follows:

---

assign initial guess values into **x** array
assign maximum number of iteration to **max**

assign the total increment tolerance of **x** values to **tolx**

assign the total tolerance function values to **tolf**

for **i** = 1 to **max** loop

calculate function values **fvec** at **x**

if **sum(fvec) < tolf** then exit

calculate Jacobian matrix **fjac**

calculate **x+h**

calculate function values **f** at **x+h**

calculate (**f** – **fvec**)/**h** for each function and each variable

assign **p** = **-fvec**

solve the equation **fjac***δ**x** = **p**, find δ**x**

if **sum**(δ**x**) **< tolx** then

exit

else

x = **x** + δ**x**

end loop

Note that, the algorithm given above is only for one step of the simulation process. This operation is repeated as many times as the number of steps which is equal to:

Total Simulation Time / Time Interval

Thus, the following is the highest-level pseudo-code of the simulation procedure:

step number = total simulation time/time interval
assign initial **x** values
for **i** = 1 to **step number** loop
    implement Newton-Raphson method, get **result** vector
    assign **result** vector to **x** vector as initial values for the next iteration
end loop

As we mentioned before, in Newton-Raphson method, initial values play a critical role in success of finding roots of the nonlinear system of equations. For this reason, at the end of each step, found roots are assigned as initial values for the next step.

4.3 Elements of NCVJSim Library

The UML class diagram of NCVJSim package is given in Figure 4.1. As can be seen from Figure 4.1. the package contains two abstract classes NCMLComponent and NMLBlock. All other classes are derived from either of these two abstract classes. In other words, we can say that each class of NCVJSim package is either inherited from NCVMLComponent or NCVMLBlock. NCVMLComponent is the abstract class of Java class implementations of NCV model circuit components, and similarly NCVMLBlock is the abstract class of Java implementations of NCV model blocks. For standardization, re-usability, and simplicity the implementation of NCV model simulation package is organized in this way. For this purpose, skeleton structures are defined as abstract classes and all other classes are defined as derivation of these classes. The class structures of

NCVMLComponent, NCVMLBlock classes and derivation from these classes are described in detail in the following part.

4.3.1 NCVMLComponent

In object-oriented design approach, abstract classes are used to define the common properties and operations of a group of classes. They contain the definitions of common attributes and common operations that take place in all members of the group of classes. Following this, all classes that derive from this abstract class automatically include the attributes and operations of the abstract class and they have to implement these operations. This structure enables simple and more organized programming structure, code re-usability and an easy approach for addition of new classes.

For implementation of NCV model components, we have defined a standard (generic) class structure and forced all class implementations of components to have this standard structure. There are two main two reasons for following such a way: first is by this standard structure, all class implementations of components will be behaved in the same way throughout Java source codes. This leads to simplicity in programming (reduces complexity) and easy readability of the code. Second reason is; it brings a standard for the class implementations, which are to be uploaded at the beginning of the simulation by users. With this standard, we force users to implement operations that we will need for simulation. This gives us enough knowledge about how to behave and utilize user-implemented classes during simulation. The Java implementation of NCVMLComponent is as follows:

```
public abstract class NCVMLComponent
{
```

```
    protected abstract  int      paramCount;
    protected abstract  String   description;
   public  abstract  void    setParam(float param[]);
   public  abstract  String  getDescription();
   public  abstract  float   getPressure(float F);
   public  abstract  float   getFlow(float P);
}
```

The keyword *abstract* in the definition of the class (1st line) defines this class to be an abstract class. The abstract attributes of the NCVMLComponent are paramCount and description, respectively. Consider that, the access types of these attributes are both *protected*, which means they can be accessed only by the operations of the same class and by operations of the inherited classes. The abstract operations of NCVMLComponent are declared to be setParam, getDescription, getPressure, and getFlow, respectively. Because of including *abstract* keyword in their declaration, all classes that inherit from NCVMLComponent must include these two attributes and implement mentioned four operations. Access type of the operations are *public*, hence they are accessible form any other class instances.

paramCount: this attribute holds the component specific information, the number of parameters. For example, as it takes the value 1 for Resistance component, it takes the value 2 for Compound Valve component. This information must be set by the user during the development phase of the source code.

description: this attribute holds the description of the component in a textual format. As it is declared to be string, any information can be included in this

attribute. This variable must be assigned value by the user during source code development.

setParam: this operation is the generic method for assigning component specific parameters in a proper way. The content of the param array is filled by the program with the parameters of components provided in the NCVML document by users. It assumes the parameter values be provided in the input parameter array *param* in an order starting from zeroth index and follows in a sequential way. Here the order of parameter values is the same as they are supplied in the NCVML document and we expect programmers to be aware of this.

To give an example, if the definition of a Compound Valve component is provided as follows in the NCVML document,

< CVALVE NO = "5" PARAM = "[4 100]">

the setParam implementation of CValve component (CValve component will be detailed in section 4.3.3.5), which inherits from NCVMLComponent will be as follows:

```
public void setParam(float param[])
{
        Rd    = param[0];
        Roff =  param[1];
}
```

where the content of param = {4,100,0,0,…}

getDescription: this method is defined to enable information acquisition about the component. Its implementation should simply return the value of the attribute *description*. Additionally, users may implement their own description procedures in this method. The simplest implementation might be as follows:

```
public String getDescription()
{
        return description;
}
```

getPressure: As we mentioned before, the simulation of NCV model is performed by iteratively solving a set of nonlinear equations, i.e. node equations and element equations. Solution of nonlinear equations requires calculation of function values at some instant times. The calculation of element functions (equations) requires acquiring the instant pressure values of elements. Note that instant pressure means the value of the pressure difference between two nodes of the element for the time interval being in. This method returns the instant pressure value on an element when instant flow passing on this element is provided. The flow passing through nodes of this element is provided as an input parameter F by the *NewtonRaphson* class instance. There are some components like Flow Source, which instant pressure value is not applicable for it. For this type of components, the function returns the value –999, which represents that this operation is not valid. Therefore, implementation of getPressure method is as follows:

```
public float getPressure (float F)
{
        //calculate pressure value for given F
        return P;
        // P is –999 if instant pressure value is not applicable
```

```
}
```

getFlow: similar to getPressure method, getFlow returns the instant flow passing through an element for a given pressure difference between two nodes of the element. The pressure difference between two nodes of an element is provided as an input parameter P by the *NewtonRaphson* class instance. For components, which instant flow is not applicable, the method returns –999 as flow, which means an invalid operation. The implementation of getFlow method is as follows:

```
public float getFlow (float P)
{
        //calculate flow for given P
        return F;
        //F=-999 if this operation is not applicable
}
```

4.3.2 NCVMLBlock

NCVMLBlock is the proposed class for NCV model block elements. The generic structure of an NCV model, block element implementation (NCVMLBlock) is as follows:

```
public abstract class NCVMLBlock
{

            private abstract  String   description;
```

```
        public abstract void setParam(String[] param,float[][] input, float[]
    preOut, int nStep, float fTime);
        public abstract float getOutput();

        public abstract String getDescription();

}
```

As you can see, block class implementations are forced to implement two operations, *setParam* and *getOutput*. The access type for both operations are *public*, hence they can be accessed from any class instance.

description: this attribute holds the textual description of the block.

setParam: this method should implement block specific parameter assignments, and initializations. All parameters that a block might need for calculation of the output is provided as input parameters. These are block specific parameters, current and previous inputs (if needed) to the block, previous outputs of the block (if needed), current step and current simulation time. An example method implementation for SLV block, which will be explained later is as follows:

```
public void setParam(String[] param,float[][] input, float[] preOut, int  nStep,
float fTime)
  {
     this.SLD    = Float.parseFloat(param[0]);
     this.SLS    = Float.parseFloat(param[1]);
     this.T1     = Float.parseFloat(param[2]);
     this.T      = Float.parseFloat(param[3]);
```

```
    this.fTime   = fTime;
  }
```

where SLD, SLS, T1, T and fTime are local variables.

Here param includes the data provided by users in the NCVML documents as parameters of this block. It has the same content with the PARAM attribute of the BLOCK NCVML element.

Input is a two-dimensional array holds input values of the block. The content of this array is generated by the simulator depending on the input specification of the block definition. Each row is reserved for one input data, starting from the 0^{th} index.

If the block definition defines the block to need more than zero previous output values, the simulator locates these outputs in preOut array in one dimension starting from 0^{th} index, but in reverse order. Thus, the last output becomes the 0^{th} index of the preOut array.

nStep is the number of the current step of the simulation process. It is provided by the simulator.

fTime is the current time of the simulation process, and is provided by the simulator.

getOutput: this method makes internal calculations using parameters supplied in setParam method to produce block output and returns the output value. Calculation may require current time, current step, block specific parameter values, previous outputs, current input, and previous inputs. The structure of getOutput method is as follows:

```
public float getOutput ()
{
        //calculate output value
        return output;
}
```

getDescription: this method should be implemented to acquire the value of *description* attribute, thus the textual description of the block. The simplest implementation should be as follows:

```
public String getDescription()
{
        return description;
}
```

4.3.3 Components

Note that, in the rest of this document P will be used as the pressure difference between two nodes of a component, and F as the flow between two nodes of a component. Until section 4.3.4, all classes inherit from *NCVMLComponent* class. Therefore, due to inheritance properties, all implement *setParam, getDescription, getPressure*, and s*etPressure* methods. While describing classes, the methods that a class implements will not be listed and described once more, instead only important points and if any, extra methods implemented by that class will be described when needed. The private attributes of a class indicate the parameters that must be provided in the NCVML document to be able to use this component. For example, if a class has two private attributes and they are assigned their values by the 0^{th} and 1^{st}

index of parameter array in the setParam method, then to be able to use this class (component) in a model the values of these two parameters must be provided beside it's NCVML element tag in the NCVML document. The order in which parameter values must be provided will be described for each component.

4.3.3.1 Resistance

This class is the Java implementation of Resistance NCV model component. It inherits from NCVMLComponent class. It has one private attribute R, which stands for resistance parameter. The parameter R is set to the 0^{th} index of the parameter in setParam method. setParam, getPresure and getFlow methods are as follows:

```
public void setParam(float param[])
{
        R = param[0];
}
public float getPressure(float F)
{
        return (F*R);
}

public float getFlow(float P)
{
        return (P/R);
 }
```

where R is the resistance parameter.

4.3.3.2 PSource

Psource is the Java class implementation of Pressure Source NCV model component. It inherits from NCVMLComponent class. It has one attribute P, which stands for presure parameter. P is set to the 0^{th} index of parameter array. The pressure difference of a pressure source at a given small time interval is equal to the value of P at that time. For this reason, getPressure method returns the value of P at that time independent of the flow F. Besides, the flow between the nodes of a presure source for a given pressure is undefined. Therefore, getFlow method returns -999 representing an invalid operation.

```
public void setParam(float param[])
{
        P = param[0];
}
 public float getPressure(float F)
{
        return P;
}
 public float getFlow(float P)
{
        return (float)-999;
}
```

4.3.3.3 FSource

Fsource is the Java class implementation of the NCV model Flow Source component. It has one private attribute F, which stands for flow parameter. F takes the value of the 0^{th} index of the paramter array in the setParam method. getFlow method returns instant value of F independent of the pressure

difference when invoked. Besides, getPressure method returns -999 as invalid operation flag because the pressure between two nodes of a pressure source for a given flow is undefined.

```
public void setParam(float param[])
{
        F = param[0];
}
public float getPressure(float F)
{
       return (float)-999;
}
public float getFlow(float P)
{
        return F;
}
```

4.3.3.4 Valve

Valve is the Java implementation of the NCV model Valve component. It has two private attributes Pth and Fs, which stand for pressure threshold and valve's saturation flow, respectively. Pth gets the value of the 0^{th} index of the parameter array while Fs gets value of the 1^{st} index in setting parameter values in setParam method. setParam, getPressure, and getFlow methods are as follows:

```
public void setParam(float param[])
{
        Pth = param[0];
```

```
        Fs   = param[1];
}
public float getPressure(float F)
{
        double tmp1 = (F/Fs)+1;
        double tmp2 = Math.log(tmp1);

        return (float) tmp2*Pth;
}
public float getFlow(float P)
{
        double tmp = (P/Pth);
        return (float)(Fs*(Math.exp(tmp) - 1));
 }
```

4.3.3.5 CValve

CValve is the Java class implementation of the NCV model Compound Valve component. It has two private atributes Rd and Roff, which stand for forward resistance and backward resistance parameters, respectively. In setParam method, Rd gets the value of the 0^{th} index and Roff gets 1^{st} index of the parameter array. setParam, getPressure, and getFlow methods are as follows:

```
public void setParam(float param[])
{
        Rd    = param[0];
        Roff  = param[1];
}
public float getPressure(float F)
```

```
{
        if (F >= 0)
                return (float) F*Rd;
        else
                return (float) F*Roff;
}

public float getFlow(float P)
{
        if ( P >= 0 )
                  return (float) (1/Rd)*P;
         else
                return (float) (1/Roff)*P;
}
```

4.3.3.6. UserClass

UserClass is an interface that to be implemented by users and dynamically loaded and linked to the package at run-time. The user-implemented Java classes must habe the structure as follows:

```
class ClassName extends NCMLComponent
{

     //decleration of parameters
     public String description = “write the description of the component here”;

      public void setParam(float param[])
      {
```

```
    //assign parameter values
}

public float getParam( )
{
   // return first parameter value
}

public int getParamCount()
{
   //return the number of parameters
 }

public String getDescription()
{
          return description;
}

public float getPressure(float F)
{
   //compute pressure value for given flow F
   //return computed pressure
 }

public float getFlow(float P)
{
     //compute flow for given pressure and return it
}
```

```
    // add and implement additional methods here
}
```

Consider that the responsibility of function of this class is completely on the user because we don't have the opportunity to debug user-implemented classes. That's why user-implemented classes must be comiled and verified before the simulation starts. Also note that getPressure and getFlow methods may do some calculations to find pressure or flow values, or they may find these values from a lookup table as well.

As user-implemented classes inherit from NCVMLComponent abstract class, to be able to compile a class NCVMLComponent class must be downloaded into a file and located in the classpath by the user.

4.3.4 Blocks

Consider that all block classes inherit from *NCVMLBlock* class. Therefore, due to inheritance properties, all implement *setParam,* and *getOutput* methods. While describing classes, the methods that a class implements will not be listed and described once more, instead only important points and if any, extra methods implemented by that class will be described when needed. The private attributes of a class indicate the parameters that must be provided in the NCVML document to be able to use this block. For example, if a class has two private attributes and they are assigned their values by the 0^{th} and 1^{st} index of parameter array in the setParam method, then to be able to use this class (block) in a model, the values of these two parameters must be provided beside it's NCVML element tag in the NCVML document. The order in which parameter values must be provided will be described for each block.

4.3.4.1. BComplianceFeedback

BComplianceFeedback is the feedback block of the compliance model, which constitutes a pressure source and a feedback block to this pressure source. It takes two parameters, which are compliance (C) and initial pressure (Po), respectively. Moreover, it has one input that the flow passing over the pressure source. Having received these parameters and the input, the block calculates the value of the pressure parameter of the next step using the following method:

```
public double getOutput()
{
        double V = Vo + F*dt;
        return V/C;
}
```

where Vo is the volume at the previous step, F is the flow passing over the pressure source at current step, and dt is the step interval time.

Consider that, this block modulates only the pressure value of the pressure source, and the compliance of this block remains constant throughout the simulation.

4.3.4.2. BVariableComplianceFeedback

BVariableComplianceFeedback is the feedback block of the variable compliance model, which constitutes a pressure source and a feedback block to this pressure source. It takes two parameters, which are initial compliance

value (C) and initial pressure (Po), respectively. Moreover, it has two input that are the flow passing over the pressure source and the current compliance value. Having received these parameters and the input, the block calculates the value of the pressure parameter of the next step using the following method:

```
public double getOutput()
{
    V  = Vo + F*dt;
    P  = V/Cnew;
    return P;
}
```

where Vo is the volume at the previous step, F is the flow passing over the pressure source at current step, Cnew is the current value of the compliance parameter and dt is the step interval time.

Consider that with this block both the pressure value of the pressure source, and the compliance value of this block varies throughout the simulation.

4.3.4.3. BComplianceUVFeedback

BComplianceUVFeedback is the feedback block of the compliance with unstressed volume model, which constitutes a pressure source and a feedback block to this pressure source. It takes three parameters, which are compliance value (C), initial pressure (Po), and unstressed volume (Vu), respectively. Moreover, it has one input that is the flow passing over the pressure source. Having received these parameters and the input, the block calculates the value of the pressure parameter of the next step using the following method:

```
public double getOutput()
{
        double V = Vo + F*dt;
        return (V-Vu)/C;
}
```

where Vo is the volume at the previous step, F is the flow passing over the pressure source at current step, Vu is the unstressed volume parameter and dt is the step interval time.

Consider that with this block only the pressure value of the pressure source varies, and the compliance value and unstressed volume of this block remain constant throughout the simulation.

4.3.4.4. BInertanceFeedback

BInertanceFeedback is the feedback block of the inertance model, which constitutes a flow source and a feedback block to this flow source. It takes two parameters, which are inertance (L), and initial flow (Fo), respectively. Moreover, it has two input that are the pressures at their edge nodes (P1,P2). Having received these parameters and the input, the block calculates the value of the flow parameter of the next step using the following method:

```
public double getOutput()
{
        double F = Fo + (P1-P2)*dt/L;
        return F;
}
```

where, Fo is the previous pressure parameter value, P1 is the pressure at the first node of the flow source, P2 the pressure at the second node of the flow source and dt is the step interval time.

Consider that with this block only the flow value of the flow source varies, and the inertance of this block remain constant throughout the simulation.

4.3.4.5 BSLV

BSLV block is the Java class implementation of the stiffnes function of the left ventricle of the heart. It generates stiffness function with given parameters. It takes only one input, which is simulation time. It has four peciular atributes: SLD, SLS, T1, and T. SLD is the minimum value of stiffness function, SLS is the maximum value of the stiffness function, T1 is the systole time and T is the heart cycle period, respectively. Attributes take the values of the 0^{th}, 1^{st}, 2^{nd} and 3^{rd} index of parameter string in the setParam method. The output of the block is the value of the stiffness function at given time. Note that in this block, the heart cycle doesn't change. Stiffness block of the right ventricle can be obtained by just providing suitable parameters to this block class.

getOutput method is as follows:

```
public float getOutput()
{
   float Slv = 0;

   Slv += this.SLD;

   float t = (float) Math.IEEEremainder(this.fTime,this.T);
```

```
    if ((0 < t ) && (t <= this.T1 ))
    {
       Slv  += this.SLS*(Math.sin(2*(Math.PI)*(1/0.6)*t));
       return Slv;
    }
    else
    {
       return Slv;
    }
 }
```

here the remainder of the current simulation time to the cycle period is calculated, because stiffness function is periodic with heart cycle period.

4.3.4.6 BVSLV

As mentioned above, BSLV has the charasteristics that the heart cycle time T is provided at the beginning of a cycle and it doesn't change. However, in some cases there is a need to change the heart cycle instantly. This is performed with BVSLV block. BVSLV is a derivation of BSLV in that a new cycle begins upon recieving an input value 1. It has one input which sould take values 0 or 1 at a time. Unless an input value 1 reaches the old cycle continues. Here we make an assumption that a new cycle doesn't start during the contraction (systole), thus between 0< t < T1.

getOutput method of BVSLV is as follows:

```
public float getOutput()
```

```
{
    float Slv = 0;

    Slv += this.SLD;

    float t = this.fCycleTime;

    if ((0 < t ) && (t <= this.T1 ))
    {
       Slv  += this.SLS*(Math.sin(2*(Math.PI)*(1/0.6)*t));
       return Slv;
    }
    else
    {
       return Slv;
    }

}
```

4.3.4.7 BInverter

This is the Java implementation of the mathematical inverter function. Thus, given an input f, it outputs 1/f. It recieves the floating number to be inverted in the first index of the input array. It takes no parameters.

```
public void setParam(String[] param,float[][] input, float[] preOut, int nStep,
float fTime)
{
    this.f    = input[0][0];
```

```
}
public float getOutput()
{
    return (1/f);
}
```

where f is the input.

4.3.4.8 BAx_Plus_b

BAx_Plus_b is the Java class implementation of the mathematical function f(x) = Ax+B. It has three private attributes. First two are parameters A and B. A and B must be supplied in the 0^{th} and 1^{st} indexes of the parameter array, respectively. Third is x and x must be provided in the input[0][0].

```
 public void setParam(String[] param, float[][] input, float[] preOut, int nStep,
float fTime)
 {
    this.A      = Float.parseFloat(param[0]);
    this.B      = Float.parseFloat(param[1]);
    this.x      = input[0][0];
 }

 public float getOutput()
 {
    float y=0;

    y += A*x;
    y += B;
```

```
    return y;
}
```

4.3.4.9 BAbsoluteDelay

This block is the Java class implementation of the absolute delay function. Thus, an applied input to the block is delayed k steps (or k*dt sec). It has one input and takes one parameter. It receives the number of steps to be delayed in param[0] and the input to be delayed in input[0][0]. Note that, the number of steps to be delayed is constant throughout the simulation.

4.3.4.10 BFilter

This block is the Java class implementation of filtering function. It implements the following mathemetical function:

$$y_n = \sum_{-\infty}^{\infty} h_k x_{n-k} + \sum_{-\infty}^{\infty} d_k y_{n-k} \qquad x \rightarrow y$$

At least two coefficient arrays must be provided as parameters of this block. These are h, which holds coefficients of input series, and d, which holds coefficients of previous output series. Number of input might be more than one. In such a case coefficient arrays of additional inputs must be supplied as well. Here the following point should be taken into consideration that the coefficient arrays of inputs are separated using ";", therefore separation of coefficients is carried out using ";" as separator.

The current value of the output y_n computed as follows:

```
public double getOutput()
{
    double y = 0;
    for (int i=0; i<rownum; i++)
    {
       for (int k=1; k<=coefficients[i][0]; k++)
           y += coefficients[i][k]*inputs[i][k-1];
    }
    return y;
}
```

4.3.4.11 BIntegrateAndFire

BintegrateAndFire is the java class implementation of the Integrate and Fire block described in Chapter 2. It doesn't have any parameter. It recieves a float input in input[0][0] and produces an output etiher 0 or 1. In general, BIntegrateAndFire and BVSLV are connected sequentially, such that the output of BIntegrateAndFire is the input of the BVSLV. When BIntegrateAndFire produces 1 the new cycle begins in BVSLV and as it produces 0 output the same cycle continues in BVSLV.

```
public void setParam(String[] param,float[][] input, float[] preOut, int nStep,
float fTime)
 {
   T  = input[0][0];
   if (nStep == 1)
   {
     this.dt = 0;
     u = 0;
```

```
    }
    else
      this.dt = fTime/(nStep-1);
  }
  public float getOutput()
  {
    u += dt/T;
    if (u >=1)
    {
      u = 0;
      return 1;
    }
    else
    {
      return 0;
    }
  }
}
```

4.3.4.12 BSSBaroReflex

BSSBaroReflex is the abbreviation for Steady-State Baroreflex Block. It implements the feedback mechanism described below:

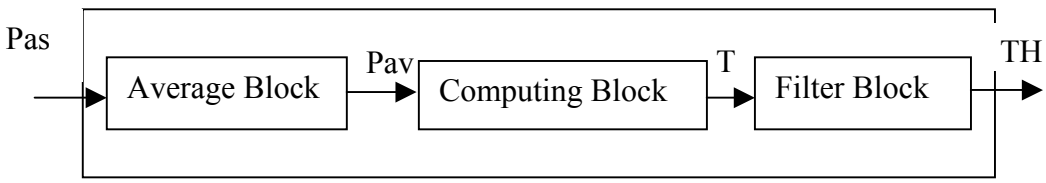


Figure 4.3. Steady-State Block Diagram

Thus, at each step, this block receives the aortic pressure as input and produces the heartbeat cycle period. Here, to calculate the next heartbeat cycle period, the average pressure value (Pav) is utilized instead of instant pressure. For this purpose, the average block takes the input pressure value and adds it to the previous values in the same cycle. At the end of each cycle period (at heartbeat) it averages the sum to the number of steps to calculate the average value of the pressure of this heartbeat interval. The average value of the pressure is applied to the block, which computes a temporary variable. This block implements the following formula:

$T = Tmin + (Tmax - Tmin)/ (1 + \gamma*\exp(-\alpha*Pav/Pe))$ where Tmin, Tmax, γ, α, Pe are constant parameters to be supplied by the user. The output of this block is applied to a filter block. Filter block applies smoothing to the output value of the computing block. The filter block implements the following formula:

$Y(i+1) = Y(i) + dt*(X(i) - Y(i))/RC$ where i represents the i^{th} step, Y is the output of the block, X is the input of the block, and finally RC is constant equals to 2.478. The output of the overall block at the last step of the current heartbeat cycle determines the next heartbeat cycle period. For source code of this block see Appendix 1.

4.3.4.13 BUrsino

BUrsino is the java class implementation of the whole feedback control model proposed by Ursino [12]. Ursino model is explained in detail in Chapter 2. This block receives only the current time as input and outputs the new heart cycle period as output. All parameters are declared in the implementation of the block. The implementation of this block will be sampled in the next chapter

(Chapter 6) in the context of a simple simulation example. For source code of Bursino, see Appendix 1.

4.4 Dynamic Class Loading and Linking

One of the most important flexibility that our framework offers is enabling users to add their own components (component or block) and incorporating this component into the simulation. This flexibility requires the ability of loading and linking valid, user-implemented java classes into the simulation package during simulation process. This is a very complex process and even impossible for most of programming languages. Because all compilers request all source codes to be available and reachable during compile-time. However, Java platform offers a great facility of dynamic class loading and linking via the *ClassLoader* class. In Java platform, types (class or interface) are made available to a running program by the Java Virtual Machine (JVM). Dynamic loading and linking allows types to be incrementally incorporated into a running JVM. This means that an application hosted by the JVM can leverage types that were previously unknown, or perhaps did not even exist, when the application was compiled. Dynamic linking and extension support the notion of *runtime assembly of code*, a notion of critical importance to server-side containers [28].

Usually, applications implement subclasses of *ClassLoader* in order to extend the manner in which the Java virtual machine dynamically loads classes [25]. For this purpose, we have inherited our own class loader *ComponentLoader* from *ClassLoader* class and implemented its methods. *ClassLoader* is an integral part of the JVM and is responsible for loading

trusted classes (e.g. basic Java class library classes). Loading and linking a class is performed in two steps:

i) given the name of the class, produce a binary data stream that constitutes that class,
ii) create an instance of class *java.lang.Class* and parse the stream of binary data produced in step i into it.
iii) resolve the class produced in step ii.

We let the class name be the same as the class uploaded by the user. By overriding the *loadClass* method of *ClassLoader*, we implemented the *loadClass* method for *ComponentLoader* class to perform dynamic loading and linking. In this method, the algorithm with described above with three steps is realized by the following source code:

```
byte data[] = (byte[])o;
c = defineClass(name, data, 0, data.length);
resolveClass(c);
```

Here, o handles the binary data stream, *defineClass* is the method of *ClassLoader* class, which converts given binary data stream into an instance of *java.lang.Class* class with provided name. Finally, *resolveClass* is the method of *ClassLoader* class, which links given class to the JVM.

A source code excerpt that describes the usage of the *ComponentLoader* class to load and link a class that has been acquired in a binary stream *data* is as follows:

```
byte[] data = null;
```

```
GetClassData(data);   //acquires the binary data stream which constitutes class to be uploaded

ComponentLoader cl = new ComponentLoader();
cl.setClassData(className, data);
Class c = new Class();
c = cl.loadClass(className,true);
```

4.5 Java Reflection Mechanism

The flexibility of enabling users to load and link classes during the run-time brings out an important problem. While types, names and executable codes of such classes are provided at java run-time (after loading), it is impossible to write conventional codes like creating objects of these classes or calling methods of such classes at compile-time. Such attempts result in compilation errors unless static source codes are located on compiler's path. But as known, there is no static class implementation for those classes at compile-time. We can access to the reqired information about dynamic classes only after they are loaded and linked into runing program. To surpass this problem we have used *Java Reflection Mechanism*, which offers ways for obtaining information about members of classes/objects and invoking their methods dynamically.

For example, take,

```
myComponent mc = new myComponent();
float P = mc.getPressure();
```

when you compile this code, the compiler will look up for the static or executable code for myComponent. If it exists and it has a valid constructor with no parameters, then the compiler will check if it has a method getPressure returning a float value as well. If it has, the compilation will complete succesfully.

But suppose myComponent is a user-implemented class and we don't have myComponent class in classpath at compile-time. It is going to be loaded and linked at run-time. In this case, compilation will fail.

Instead, Java Reflection Mechanism offers the flowing coding approach;

```
// load and link dynamic class
ComponentLoader cl = new ComponentLoader();
cl.setClassData("myComponent", data);
Class c = new Class();
c = cl.loadClass("myComponent",true);
.
.
Object o =c.newInstance();
Float d = (Float) c.getMethod("getPressure",null).invoke(o, null);
```

which does the same thing by reflection. In tis case, when the compiler looks at that code, it sees only *ComponentLoader,* `Object`, `Class`, `Float`, etc, and a bunch of *Strings* (which happen to contain class and method names, but it doesn't know about that). Therefore, compilation succeds.

Note that, the precise class of the object to create, the name of the method, the number and type of parameters etc, are all contained in runtime variables ("myComponent" is a hardcoded string here, but it could as well be typed in by an user). Therefore, the compiler will compile the code even if there is no myComponent class in the classpath [29]. Bu this time, the responsibility of providing myComponent class, which implements getPressure method is still available because at run-time JVM will try to create an instance of myComponent and to invoke its getPressure method. Absence of any of these will result in run-time error.

Reflection plays an essential role in adding dynamism to Java. However, using the features provided by the reflection mechanism in Java requires careful consideration. Moreover, reflection method calls have substantial performance overhead. Besides, return values from reflection method calls almost always have to be cast to the right type, making the code type-unsafe and less efficient. [30]

Chapter 5

General Architecture of the Framework

5.1. Overview

As we mentioned previously, one of the basic property of our framework is that it offers web-based access. A web-based application is any application that offers access from physically different locations all arround the world by utilizing the internet. Web applications use Hypertext Transfer Protocol (HTTP) as its primary transport protocol. We have implemented the framework as web-based to achieve features such as global availability, easy maintenance, and platform independence [14].

Global availability: Thanks to the wide availability of the internet and WWW, access to a web-based application becomes very easy just by using browsers like Internet Explorer or Netscape. The only requirements are an internet connection and a WWW-browser, which are already widely available.

Furthermore, since the internet is availabile 24 hours under normal circumstances, access to an application becomes time-independent.

Easy maintenance: With WWW-based simulation, the simulation framework is located on a single server in one location. This enables modification of a single workspace in case of adding new features or components to the framework, which contributes the extensibility of the framework. Moreover, in case of error detection, removal of the error in a single location contributes the maintainability of the framework.

Platform independence: In our WWW-based simulation framework, the client-side environment runs simply HTML page in a www-browser. Thus, no platform specific application is performed on client-side. This enables platform-independent access and implementation.

Several system architectures might be constructed for providing web-based simulation. However, system architectures providing web-based web simulation include common central components that can be grouped into two groups: client-side environment and server-side environment. Client-side environments commonly include web browsers implementing HTML codes to interact with the user. Moreover, Java applet is another common component of client-side environment which is embedded into HTML page to interact with users and to execute applications on using client's resources. On the other hand,the essential components of the server-side environment are web server application, intermediate application running on the server and the simulator application. Web server software is the application that runs on a computer and makes Web pages stored on this computer available to Internet users. Thus, web servers perform the communication between client-side and server-side, back and forth. CGI and Java Servlet technologies are two well-known

technologies that operate on the server-side to process client-side requests and produce dynamic-content web pages.

Following the above discussion, in its simplest form, our framework consist of two parts: 1) client-side environment, 2) server-side environment. Client-side environment is simply an HTML page providing user interface to users. Server-side environment consists of web-server, Java Servlet, XML parser and the simulation package. Each of these components will be described in detail in the following sections. The general architecture of our proposed framework is depicted in Figure 5.1.

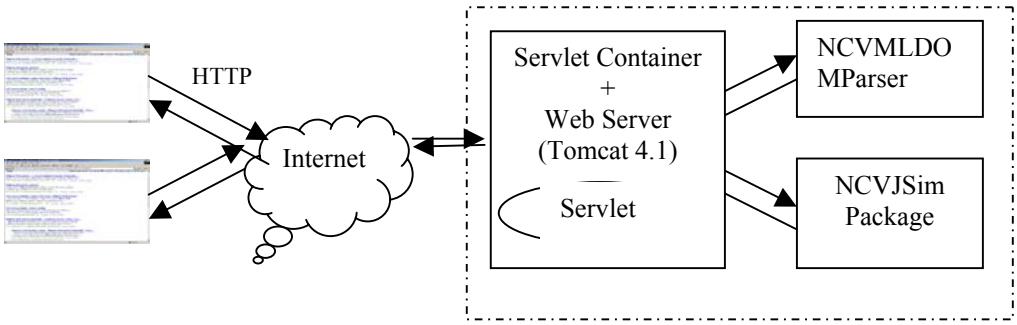


Figure 5.1. The General Architecture of the Framework

5.2 Client-Side Environment

Client-side environment is composed of simple HTML (HyperText Markup Language) pages that can be viewed by WWW browsers. Thus, a WWW browser is the required and the adequate software for accessing the server-side simulation environment. Since many users are familiar with navigating with WWW browsers, client-side environment offers an easy-to-learn content.

The basic idea in our client-side environment implementation is to implement the so called *thin client*. In web programming paradigm, thin-client refers to the approach in which clients serve only for interaction with user purposes and data transfer to the server while the bulk of processing occurs on the server-side. In this approach, client interacts with users to get required input, transfers obtained data to the server-side environment and displays static content as well as dynamic simulation results. Client-side and server-side environments use HTTP protocol to communicate and transfer data.

To access the client-side environment, users must start with typing the URL address (http://139.179.12.157:8080/ncvsimulation/index.html) of the entry page on the browsers navigation address section. This operation results in display of the entry page (index.html). Entry page is an HTML page having a menu and static text content. The snapshot of the entry page is given in Figure 5.2.

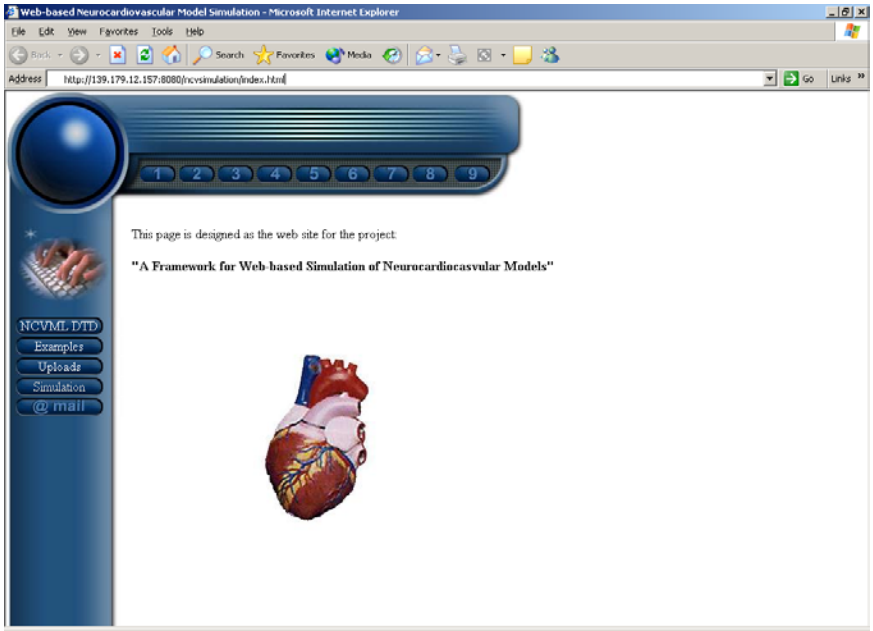


Figure 5.2. Snapshot of the client-side environment entry page

The HTML page offers the following functions:

i) View NCVML DTD

ii) Simulate a model

iii) View NCVJSim components and user's guide

iv) Upload a seperate model, component or block class

The UML use-case diagram of the client-side environment is given in Figure 5.3. Note that since all user-oriented functionalities can be accessed from the client-side environment, this use-case diagram is also the use-case diagram of the overall framework.

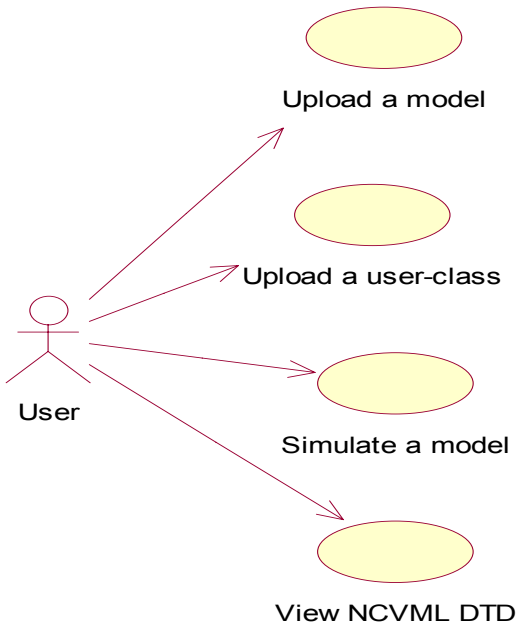


Figure 5.3. UML Use-case Diagram of the Client-side Environment

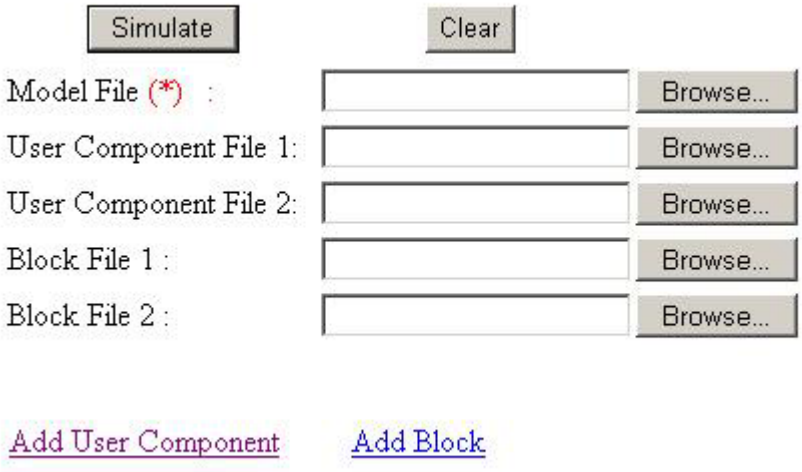
i) Users can access the full specification of the NCVML DTD and XML mappings of NCVML elements.

ii) Users can upload their XML encoded models and initiate simulation process. For this purpose, the XML encoding of the model description must be encoded before the simulation process. Users have to save the description of a model in a text file and save it with .XML extension. HTML offers FORM container tag to enable input acquisition and action handling in web environment. Therefore, to enable file uploading and simulation initiation HTML FORM container is used. INPUT tag is also included to acquire file path and name of the XML file on the client's hard disk. For this purpose, an HTML text input field is included and a predefined type of button Browse is added. Browse button gives the facility of browsing user's hard disk and selection of a file. After selection of the XML file, user can initiate simulation process by just clicking "Simulate" button. This action produces an HTTP request directed to the server-side (web server) addressed by the "SERVERIPADDRESS" address and causes invokation of a servlet instance mapped by "Servlet-Allias". Note that here, client requests are welcomed from 8080 port in server-side listener. HTML source code of simulation interface is as follows:

```
<form action="http://SERVERIPADRESS:8080/Servlet-Allias"
method="POST" enctype="multipart/form-data">>
        <p> NCML File: <input type="text" name="Filename" size="40">
            <input type="button" value="Browse" name="B3">
        </p>
        <p>  <input type="submit" value="Simulate" name="B1">
             <input type="reset" value="Clear" name="B2">
        </p>
</form>
```

Here the “POST” is used as FROM method type, because the server-side Java-API’s responsible for file uploading are related with “POST” method.

The browser view of this HTML source code is as follows:



iii) Users can access class and method definitions of NCVJSim library. Additionally they can recieve information and structure about implementing their own classes.

iv) Beside simulating a model, users can upload a seperate model or user-implemented java classes. Uploaded models are grouped on the server-side in a repository, so that these neurocardiovascular models can be used in the future for Neurocardiovascular Portal Project. Similarly, uploaded classes will be examined by an administrator and incorporated into NCVJSim library, so that the library evolves in time.

5.3 Server Side Environment

5.3.1 Web Server & Servlet Container

Web server is the "sine qua non" part of web applications. Every web application architecture includes a web server. Web server is a software running on a server machine, and delivering web pages upon incoming page access requests. But web servers serve up static content web pages. To add dynamic content to web pages, web servers can operate in connection with enxtension technologies like CGI, PHP or Java Servlets. Java Servlet and Java Server Pages (JSP) technologies offer dynamic HTML pages, which are Java-based technologies. With Java-based technologies to add dynamic structure to web pages, one possibility is to use a web server in connection with a Servlet Container. Apache web server is the most widely known open source web server. Similarly, Tomcat is the most widely known open source servlet container. Therefore, one choice for offering dynamic HTML pages is using Apache web server and Tomcat together. In such an architecture, Apache and Tomcat can communicate with mod_jk apache modul. This brings java application capability to the Apache web server. However, another choice for offering dynamic HTML pages with server-side, Java-based technology is using Tomcat as standalone. Tomcat provides web server capability as well as servlet container function, therefore Tomcat can serve as standalone without another web server.

In our study, we have used Tomcat 4.1 as our web server-servlet container. We prefered Tomcat, because it is free and open source, it offers simple methods of deploying an application, as well. Tomcat is the servlet container that is used in the official reference implementation for the Java Servlet [31] and Java Server Pages [32] technologies. The Java Servlet and

Java Server Pages specifications are developed by Sun under the Java Community Process [33]. It provides a Java Virtual Machine and associated elements to give a complete Java Runtime Environment [34].

5.3.1.1 Application Deployment

Installation of Tomcat 4.1 creates directory structure as follows:

- **/bin** - Startup, shutdown, and other scripts. The *.sh files (for Unix systems) are functional duplicates of the *.bat files (for Windows systems). Since the Win32 command-line lacks certain functionality, there are some additional files in here.
- **/conf** - Configuration files and related DTDs. The most important file in here is server.xml. It is the main configuration file for the container.
- **/logs** - Log files are here by default.
- **/webapps** - This is where user web applications go.

Directory Structure

So, users locate their web applications under **/webapps** directory. This placement is done as follows:

1) A directory is created under the directory /webapps and assigned the name **NCVSimulation,**
2) The file **index.html** is placed under the **NCVSimulation** directory,
3) The following directory structures are created: **/NCVSimulation/WEB-INF/classes,** and **/NCVSimulation/WEB-INF/lib**,
4) Java classes **NCVSimServlet.class**, **NCVMLDOMParser.class**, **SimulationParams.class**, **Block.class**, **BlockInput.class**, **Chart.class**,

Result.class, **UserClassDesc.class** and **FileUpload.class** are placed under **/WEB-INF/classes** directory,

5) An XML file **web.xml** is created with the following content and saved under the **/WEB-INF** directory,

```
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
   "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
   <servlet>
      <servlet-name> NCVSimServlet </servlet-name>
      <servlet-class> NCVSimServlet</servlet-class>
   </servlet>
   <servlet>
      <servlet-name>FileUpload</servlet-name>
      <servlet-class>FileUpload</servlet-class>
    </servlet>

   <servlet-mapping>
      <servlet-name> NCVSimServlet </servlet-name>
      <url-pattern>/ncvsimulation  </url-pattern>
   </servlet-mapping>
   <servlet-mapping>
      <servlet-name>FileUpload</servlet-name>
      <url-pattern>/fileupload </url-pattern>
   </servlet-mapping>
</web-app>
```

6) Jar libraries NCVJSim.jar, OR-Objects.jar, cos.jar are placed under **/WEB-INF/lib** directory,

7) The following directories have created: C:\NCVSimulation\temp, C:\NCVSimulation\models, C:\NCVSimulation\components, C:\NCVSimulation\blocks.

5.3.2 Java Servlet Technology

Servlet is Java technology's alternative to conventional CGI programming. Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems. A servlet can almost be thought of as an applet that runs on the server side--without a face. Java servlets make many Web applications possible [31]. Servlet programs run on a Web server and enables building dynamic web-pages. Java objects, which are based on servlet framework and APIs also extend the functionality of a HTTP server. Following is the list of merits of servlet technology [35]:

- Servlets are more efficient than CGI processes

  CGI (Common Gateway Interface) is a specification for transferring information between a web server and a CGI program. CGI programs run on the server and provide interaction between browser (user) and server. CGI was one of the common ways to provide dynamic content until introduction of servlet technology. One problem with usign CGI is that each time a CGI program is executed, a new process must be initiated. And once the program has executed a request, it disappears. This result in high overhead, therefore slow down on servers. However, servlet is persistent. Thus, once it is started, it remains in memory and therefore can process

multiple requests. This leads to the conclusion that servlets are faster and provides higher performance than CGI.

- Servlets are safe

Servlets run in the same address space as the server. Thus the server is potentially vulnerable to attack by unstable servlets. However, Java has a protection mechanism that prevents such attacks, thereby making it safe to run servlets in the server's address space.

- Servlets are written in Java

Since servlets are java classes they embody all features of Java language, including portability, performance, reusability, and crash protection. Also, servlets have access to the entire family of Java APIs and a library of HTTP-specific calls.

The following is the basic structure of a servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ExampleServlet extends HttpServlet
{
        public void doGet(HttpServletRequest request, HttpServletResponse
response)
                        throws ServletException, IOException
        {
```

```
        // Use "request" to read incoming HTTP headers (e.g. cookies)
        // and HTML form data (e.g. data the user entered and submitted)

        // Use "response" to specify the HTTP response line and headers
    // (e.g. specifying the content type, setting cookies).

        PrintWriter out = response.getWriter();
        // Use "out" to send content to browser
    }
}
```

To be a servlet, a class should extend **HttpServlet** class and override **`doGet`** or **`doPost`** (or both), depending on whether the data are sent by `GET` or by `POST from the client-side`. These methods contain two arguments: an **HttpServletRequest** and an **HttpServletResponse**. The `HttpServletRequest` has methods that let programmers get incoming information such as FORM data, HTTP request headers, and etc. The `HttpServletResponse` has methods that let programmers specify response headers (`Content-Type`, `Set-Cookie`, etc.), and most importantly, let programmers obtain a **PrintWriter** used to send output back to the client. For simple servlet programs, most of the efforts are spent on `println` statements that generate the desired page.

Note that a servlet class has to import classes in `java.io` (for `PrintWriter`, etc.), `javax.servlet` (for `HttpServlet`, etc.), and `javax.servlet.http` (for `HttpServletRequest` and `HttpServletResponse`).

We have implemented a Java servlet named **NCVSimServlet** running on the Tomcat at server side. NCVSimServlet is responsible of the following functionalities:

1. Uploading model description from the client-side,
2. Uploading user-classes, if any,
3. Managing parsing operation of the NCVML-encoded user-provided model (i.e., thus passing the model as a parameter to NCVMLDOMParser object and receiving parsed model),
4. Managing the simulation operation and obtaining results,
5. Formatting simulation results as HTML and sending to the browser,

To upload a file from the client-side, the famous java upload package **com.oreilly.servlet** is utilized. The jar archieve of this package is downloaded from its official web site [36] and incorporated into the servlet project. We import the package with the following code in order to access its **MultipartRequest** class, which we use for uploading one or multiple files at one instance.

```
import com.oreilly.servlet.multipart.*;
import com.oreilly.servlet.MultipartRequest;
```

NCVSimServlet creates an instance of the NCVMLDOMParser (will be described in the next section-section 5.3.3) class and calls its **parse()** method to let the XML file be parsed. To the parse method, the xml document name with its path is passed as a string variable. NCVMLDOMParser converts the model description into internal variable values and structures. Hence, to obtain internal structres derived by the parser NCVSimServlet calls methods of the parser as follows:

```
//create parser object to parse NCML document
NCVMLDOMParser p  =  new  NCVMLDOMParser();
//the path of the input file
String s = new String("C:\\NCVSimulation\\temp\\" + fileName);

//parse the document
p.parse(s);

//obtain parameters
p.getParameters(smlParam);

//declare the internal structures of the model and block descriptions
double[][] circuitDefinition =  new
double[nNodes+nElements][nNodes+nElements];
Vector     BlockVector      =  new Vector();
Vector     UserClassVector  =  new Vector();


//get the circuit definition matrix and block definitions from the parser object
circuitDefinition  =  p.getMatrixParameter();
BlockVector       =  p.getBlockVector();
UserClassVector    =  p.getUserClassVector();
```

In this code, circuitDefinition is the internal structure of the description of the mechanical part of a NCV model. Similarly, BlockVector holds the internal structure of the description of blocks in the NCV model and UserClassVector holds java classes to be uploaded dynamically at the beginnig of the simulation.

NCVSimServlet also initiates and manages the simulation process. For this, basicly an instance of the NewtonRaphson class is created and through iteration simulation continues as follows:

```
//create nonlinear equations solver which utilizes Newton-Raphson algorithm
NewtonRaphson nrp = new NewtonRaphson();

for (int i=1; i<= STEPS ; i++)
{
      //solve equations using Newton-Raphson algorithm and get results
       result = nrp.mnewt(…);

       //perform updates
       .
       .
       .
       //perform block operation
       blockOperations(…);
 }
```

where STEPS is the number of iterations, maxIter is the maximum number of Newton-Raphson iterations. Here only the skeleton of the simulation process source code is given only to point out the implementation.

We have utilized free java package JFreeChart 0.9.11 to draw charts showing results of the simulation visually. We have downloaded the jar archive from [37] and imported to the project as follows:

```
import org.jfree.data.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;
```

This helped us in displaying simulation results in a visual and convenient way. Note that, charts are formed by NCVSimServlet on-fly and placed into the HTML code. Following this, this HTML code is transfered by the browser to the client-side and displayed by the browser. Therefore, the chart is also transfered to the client-side as a part of the HTML code.

5.3.3 XML Parser

As we mentioned earlier, the description of NCV models are represented to the system by users at the client-side. These descriptions uses the notation of our proposed XML-based language, NCVML. Following these, the XML file is uploaded to the server-side. Before numerical simulation, the description of the model should be converted into internal data structures of Java to be able to perform numeric calculations. For this purpose, the XML document should be processed to access its internal structures (i.e., elements, attributes, values, etc.). To process an XML document by accessing internal structures, we need to use APIs (Application Programming Interfaces). The XML 1.0 Recommendation defines the precise behaviour of an XML processor when reading and printing a file, but it says nothing about which API to use. In this study, we have implemented our own NCVML document parser by utilizing DOM (Document Object Model) (Level 2), a tree structure-based API issued as a W3C Recommendation in November 2000 [38].

DOM converts the XML content of a file into a tree structure as it processes it. Listing 5.1 shows a sample NCVML document DOMExample.XML and Figure 5.2 gives its corrsponding DOM structure.

```
<!DOCTYPE NCVML SYSTEM "NCVML.DTD">
<NCVML VERSION = “1.0”>
<MODEL>
<NELEMENTS            VALUE = “3”>
<NNODES               VALUE = “3”
<REFERENCENODE    VALUE = “3”>

<CONNECTION FROM = “3” TO = “1”>
<PSOURCE  NO = “1” PARAM = “[5]”>
</CONNECTION>
<CONNECTION FROM = “1” TO = “2”>
<RESISTOR  NO = “2”  PARAM = “[3]” >
</CONNECTION>
<CONNECTION FROM = “2” TO = “2”>
<RESISTOR  NO = “3”  PARAM = “[1]” >
</CONNECTION>
</MODEL>

<SIMULATION>
<TOTALTIME        VALUE = “10”>
<TIMEINTERVAL  VALUE = “0.01”>
</SIMULATION>
</NCVML>
```

Listing 5.1. Sample NCVML file, DOMExample.XML

The NCVML code given above is the description of a 3 element, 3 node model, which includes one PSOURCE, and two RESISTOR components. The first line declares that this XML uses NCVML DTD notation. NCVML DTD is included in the NCVML.DTD document. The root element of this

NCVML document is NCVML tag. The following is the structure that DOM parser constructs after reading this NCVML document. Rectangular boxes represent elements and attributes of elements are attached at right-hand side of each element following a line.

Java 2.0 offers the package *org.w3c.dom,* which provides the interfaces for the Document Object Model (DOM). It contains required interfaces like Attr, Element, Document, etc to implement all components of an XML document. To process XML documents we have utilized Java 2's *javax.xml.parsers* package, which provides classes allowing the processing of XML documents. The following source code illustrates the process of reading an XML document and constructing the DOM tree:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.w3c.dom.*;

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setValidating(true);
try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse( new File(filename) );
} catch (SAXException sxe) { sxe.printStackTrace(); }
```

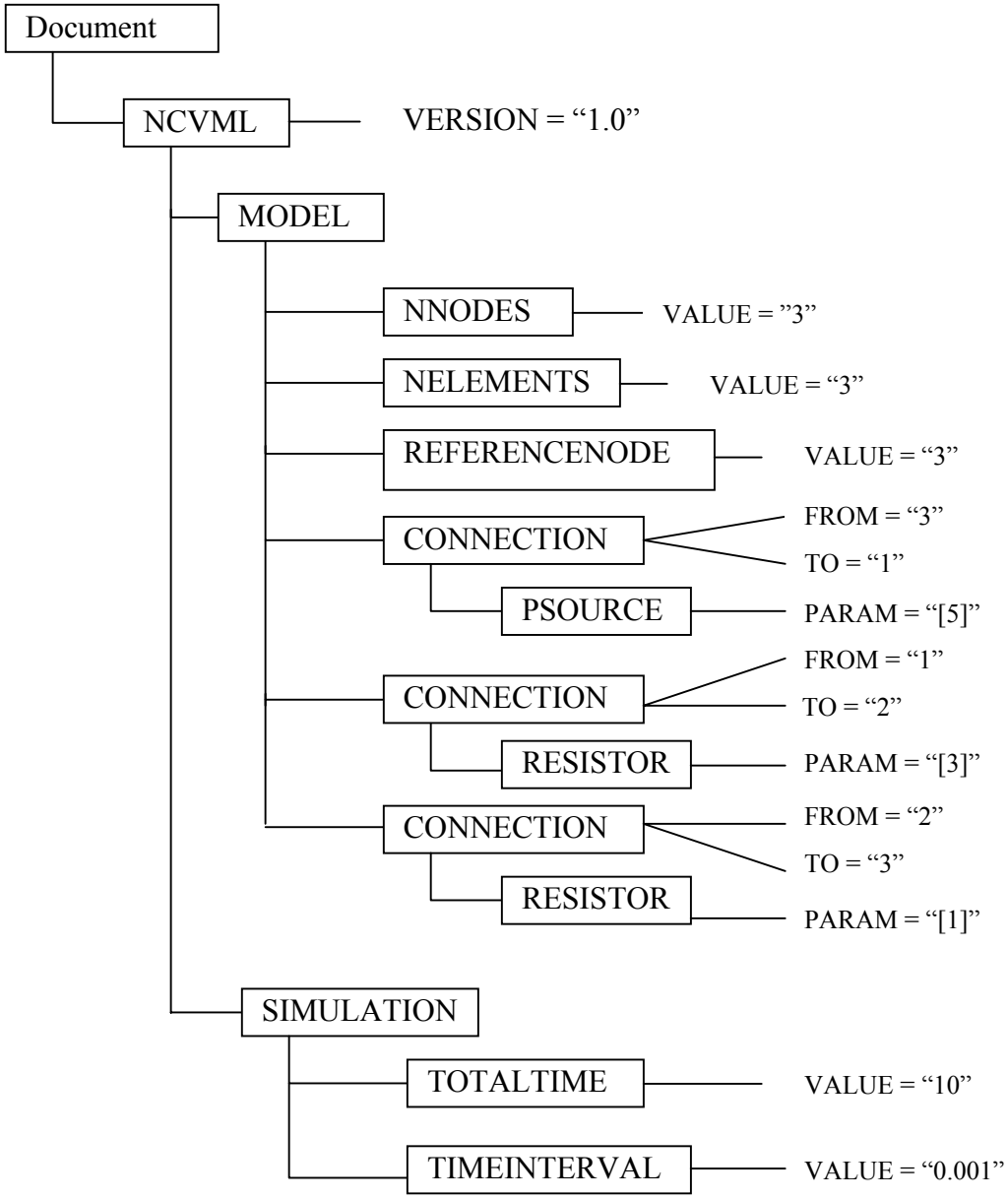


Figure 5.4. DOM structure of DOMExample.XML

where *document* is of type *org.w3c.dom.Document,* and file is a string variable that holds the name of the XML file with its path. To catch parsing exections we utilized SAXException interface that is a member of *org.xml.sax* package. The line *factory.setValidating(true)* enables validation of the XML document

across the NCVML DTD, but we disabled this feature in order to use our own controls.

To process this DOM tree, NCMLDOMParser implements a iterative-recursive method *processDOMRecursively(Node node)*. This method uses the pre-order tree traversal algorithm to traverse and process all nodes. Thus, it starts traversing the tree starting from the root element. Then, recursively traverses left sub-tree childs of a node until leaf node reaches. Following this, it traverses right sub-tree. The traversal of the DOM tree process is performed with the following code:

```
Node node = document.getDocumentElement();
NCVMLDOMParser DOMParse = new NCVMLDOMParser();
DOMParse.processDOMRecursively(node);
```

Here, in the first line *node* is assigned to the root element of the NCVML document, which is NCVML element. In the second line, an instance of the NCVMLDOMParser is cerated ant its *processDOMRecursively* method is called with address of the root element as parameter. Java implementation of the *processDOMRecursively* method is as follows:

```
public void processDOMRecursively(Node node)
{
    if (node instanceof Element)
    {
            //node element is processed here
            process current element
    }
```

```
    for (Node child = node.getFirstChild(); child != null ; child =
child.getNextSibling())
    {
            processDOMRecursively(child);
    }
}
```

As can be seen from the above code, first the current node is processed, then recursively *processDOMRecursively* method is called with the left child of the current element.

5.3.4 NCVJSim Package

NCVJSim is a Java package including library classes of the NCV simulation framework developed as a part of this framework. . It is implemented using JBuilder 4.0 and JDK 1.4.0. We explained NCVJSim in the previous chapter (Chapter 4). *NCVJSim.jar* is included in the implementation of the NCVSimServlet so that we can access library classes throughout the simulation process.

5.4 Information Flow

Information flow througout the simulation process carries out as follows:

1. At client-side, user browses for the model to be simulated, selects the XML file and clicks “Simulate” button to initiate the simulation process. This causes an HTTP POST message to be transmited to the

server-side including the allias of the NCVSimServlet and the XML file path.

2. As HTTP POST message is transmitted to the server-side, Tomcat recieves the HTTP POST message and invokes NCVSimServlet's *doPost()* method.
3. *doPost()* method uploads NCVML file to the server-side.
4. NCVSimServlet creates an instance of NCVMLDOMParser class and calls its parse() method by sending the file name of the NCVML file with its path as a string parameter,
5. NCVSimServlet calls getNodeParameter(), getElementParameter(), getReferenceNode(),getBlockCount(),getMatrixParameter(), and getBlockVector() methods to obtain description of the model,
6. NCVSimServlet creates an instance of NewtonRaphson class of the NCVJSim package.
7. For number of STEPS loop
   7.1. NCVSimServlet calls *mnewt()* of the NewtonRaphson class instance, recieves simulation results in an array,
   7.2. NCVSimServlet calls its *blockOperations()* method,
8. NCVSimServler formats simulation results in graph(s) and sends to the client-side in HTML format as HTTP message.

The UML sequence diagram of the framework is given in Figure 5. 5.

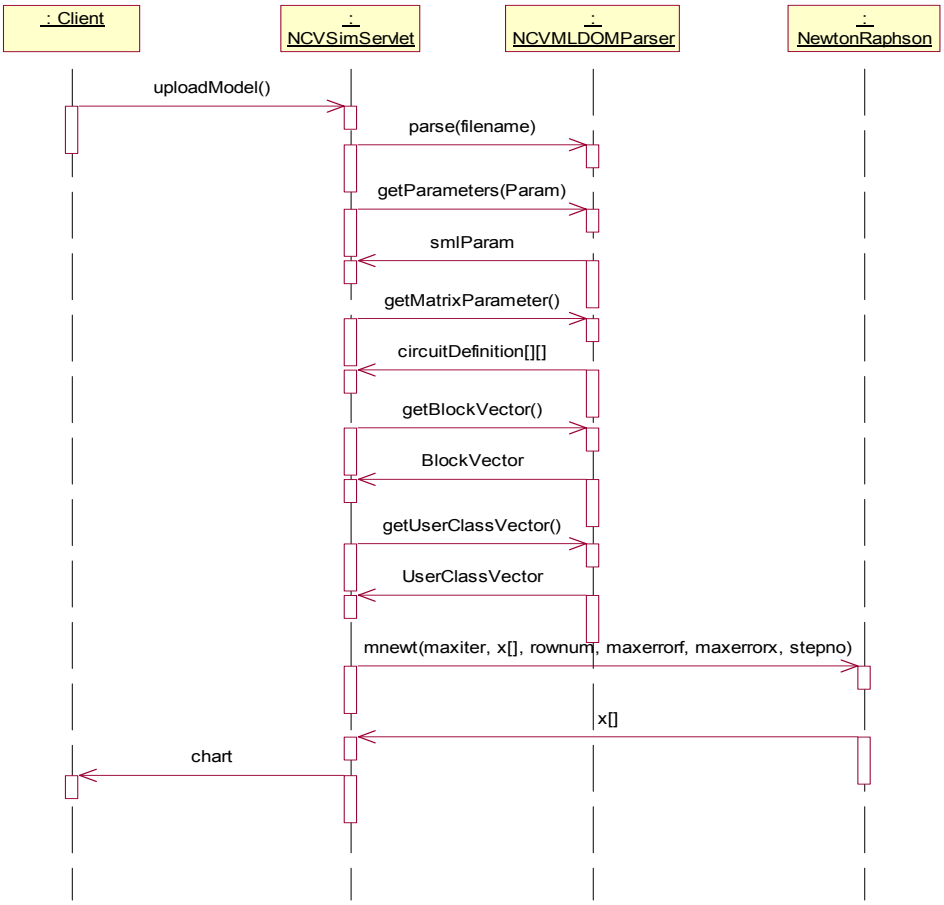


Figure 5.5. The UML sequence diagram of the framework

Chapter 6

Sample Applications

6.1. Introduction

In this chapter, we are going to present two sample applications of a moderate level of complexity as well as a comprehensive model with complex feedback mechanisms. First sample is the application of the cardiovascular system model explained in detail in section 2.1, which is known as cardiovascular mechanic system. Second sample is the comprehensive and complete neurocardiovascular model proposed by Ursino [12] in 1998.

The purposes of this chapter are as follows: we aim to give readers the required information about how to acces to the framework and how to use it, that's why this chapter might serve as a simple user's guide. The second reason it to show that our framework is a practical and running framework, even it can perform comprehensive up-to-date neurocardiovascular models.

Through the following two sections after basic descriptions of each model, all required steps for simulation of a model is explained in detail and finally simulation results are presented.

6.2 Cardiovascular Model Example

In the section 2.1, a simple, moderate level of complexity mechanic cardiovascular system model was discussed in detail. As can be remembered, this model composes left heart, right heart, systemic arterial and venous, pulmonary arterial and venous compartments. Left and right heart are modeled as variable compliances, other arteries and venouses are models as compliances, valves are modeled as a resistance in series with valve component, and finally resistance to the blood flow through veins are modeled with resistance component. Variation of the compliance of variable compliance components is a half-wave sinosoidal SLV and SRV functions. This model incorporates no feedback loop (except for compliance and variable compliance feedbacks). Therefore, to simulate this model using our proposed framework the following steps are followed:

i) Assigning numbers to nodes and elements: In this step, all nodes and all elements of the model should be determined and unique numbers should be assigned to both nodes and elements, seperately. Additionally, the reference node should be determined. Accoding to this, we start to assign numbers from the node, which stands for the systemic arterial pressure and continue increasing the number of both nodes and elements by 1, clock-wise. Therefore the systemic arterial compliance component is assigned the element number 1, Rs assigned 2, etc. Using this unique number assignment procedure, the model incorporates 7 nodes and 12 elements. Consider that valves (combination of a

resistance and a valve) are assigned a single number. The reference node is determined to be the node 7, which concludes the first step.

ii) Building NCVML description: In this step, we are going to construct NCVML description of this model considering our previous provided infomation about NCVML. Under the light of previous notifications we can conclude that the following NCVML elements are used to describe model elements.

Flow Resistance - RESISTANCE
Compliances - PSOURCE + BComplianceFeedback Block
Variable Compliance - PSOURCE + BVComplianceFeedback Block
Resistance + Valve - CVALVE

Following this, utilizing the unique numbers assigned in the first step, our task is to give connection, component and block descriptions of this model with above listed NCVML keywords. Here it should be noted that blocks also should be assined unique numbers. Therefore, we assign number 1 starting from the first compliance feedback block and continue sequentially until we assign numbers to all blocks.

Considering the points explained above, our XML file named EXAMPLE-CV.XML turns out to be as follows:

```
<NCVML VERSION = "1.0">
<MODEL>
  <NNODES          VALUE = "7" />
  <NELEMENTS       VALUE = "12" />
  <REFERENCENODE   VALUE = "7" />

  <CONNECTION FROM="1" TO="2">
    <RESISTANCE NO="1" PARAM="[1]"/>
```

```
  </CONNECTION>
  <CONNECTION FROM="2" TO="3">
    <CVALVE NO="2" PARAM="[5/1332 100]"/>
  </CONNECTION>
  <CONNECTION FROM="3" TO="4">
    <CVALVE NO="3" PARAM="[80/1332 100]"/>
  </CONNECTION>
  <CONNECTION FROM="4" TO="5">
    <RESISTANCE NO="4" PARAM="[0.05]" />
  </CONNECTION>
  <CONNECTION FROM="5" TO="6">
    <CVALVE NO="5" PARAM="[50/1332 100]"/>
  </CONNECTION>
  <CONNECTION FROM="6" TO="1">
    <CVALVE NO="6" PARAM="[80/1332 100]"/>
  </CONNECTION>
  <CONNECTION FROM="1" TO="7">
    <PSOURCE NO="7" PARAM ="[99.8]"/>
  </CONNECTION>
  <CONNECTION FROM="2" TO="7">
    <PSOURCE NO="8" PARAM="[7.19]"/>
  </CONNECTION>
  <CONNECTION FROM="3" TO="7">
    <PSOURCE NO="9" PARAM="[2.01]"/>
  </CONNECTION>
  <CONNECTION FROM="4" TO="7">
    <PSOURCE NO="10" PARAM="[11.9]"/>
  </CONNECTION>
  <CONNECTION FROM="5" TO="7">
    <PSOURCE NO="11" PARAM="[7.5]"/>
  </CONNECTION>
  <CONNECTION FROM="6" TO="7">
    <PSOURCE NO="12" PARAM="[4]"/>
  </CONNECTION>


  <BLOCK NO = "1" TYPE = "DEFINED" NAME = "BComplianceFeedback"
PARAM = "[4 99.8]" TRGTYPE = "COMPONENT" TARGET = "7"
PARAMNO="1" NOFPREV = "1" PREVVALS="[99.8]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="7" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "2" TYPE = "DEFINED" NAME = "BComplianceFeedback"
PARAM = "[400 7.19]" TRGTYPE = "COMPONENT" TARGET = "8"
PARAMNO="1" NOFPREV = "1" PREVVALS="[7.19]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="8" INPUTTYPE="FLOW"
```

```
NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "3" TYPE = "DEFINED" NAME = "BComplianceFeedback"
PARAM = "[5328/67 11.9]" TRGTYPE = "COMPONENT" TARGET = "10"
PARAMNO="1" NOFPREV = "1" PREVVALS="[11.9]">
          <INPUT SRCTYPE="COMPONENT" SRCNO="10"
INPUTTYPE="FLOW" NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "4" TYPE = "DEFINED" NAME = "BComplianceFeedback"
PARAM = "[250 7.5]" TRGTYPE = "COMPONENT" TARGET = "11"
PARAMNO="1" NOFPREV = "1" PREVVALS="[7.5]">
          <INPUT SRCTYPE="COMPONENT" SRCNO="11"
INPUTTYPE="FLOW" NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "5" TYPE = "DEFINED" NAME = "BSLV" PARAM = "[67/1332
2500/1332 0.3 0.8]" TRGTYPE = "BLOCK" TARGET = "6" NOFPREV = "0">
    <INPUT SRCTYPE="TIME" />
  </BLOCK>

  <BLOCK NO = "6" TYPE = "DEFINED" NAME = "BInverter" TRGTYPE =
"BLOCK" TARGET = "7" NOFPREV = "0">
    <INPUT SRCTYPE="BLOCK" SRCNO="5" NOFPREV = "0" />
  </BLOCK>

  <BLOCK NO = "7" TYPE = "DEFINED" NAME =
"BVariableComplianceFeedback" PARAM = "[1332/67 4]" TRGTYPE =
"COMPONENT" TARGET = "12" PARAMNO="1" NOFPREV = "1"
PREVVALS="[4]">
    <INPUT SRCTYPE="COMPONENT" SRCNO="12" INPUTTYPE="FLOW"
NOFPREV="0" />
    <INPUT SRCTYPE="BLOCK" SRCNO="6" NOFPREV = "1"
PREVVALS="[1332/67]"/>
  </BLOCK>

  <BLOCK NO = "8" TYPE = "DEFINED" NAME = "BSLV" PARAM = "[67/1332
2500/1332 0.3 0.8]" TRGTYPE = "BLOCK" TARGET = "9" NOFPREV = "0">
    <INPUT SRCTYPE="TIME" />
  </BLOCK>

  <BLOCK NO = "9" TYPE = "DEFINED" NAME = "BInverter" TRGTYPE =
"BLOCK" TARGET = "10" NOFPREV = "0">
    <INPUT SRCTYPE="BLOCK" SRCNO="8" NOFPREV = "0" />
  </BLOCK>
```

```
  <BLOCK NO = "10" TYPE = "DEFINED" NAME = "BAx_Plus_b" PARAM="[4
0]" TRGTYPE = "BLOCK" TARGET = "11" NOFPREV = "0">
    <INPUT SRCTYPE="BLOCK" SRCNO="9" NOFPREV = "0" />
  </BLOCK>

  <BLOCK NO = "11" TYPE = "DEFINED" NAME =
"BVariableComplianceFeedback" PARAM = "[5 2.01]" TRGTYPE =
"COMPONENT" TARGET = "9" PARAMNO="1" NOFPREV = "1"
PREVVALS="[2.01]">
    <INPUT SRCTYPE="COMPONENT" SRCNO="9" INPUTTYPE="FLOW"
NOFPREV="0" />
    <INPUT SRCTYPE="BLOCK" SRCNO="10" NOFPREV = "1"
PREVVALS="[5]" />
  </BLOCK>

</MODEL>

<SIMULATION>
        <TOTALTIME VALUE = "10" />
        <TIMEINTERVAL VALUE = "0.01" />
        <CHART TITLE = "Aortic Pressure(Psa) vs Time" XLABEL="Time"
YLABEL="Pressure" SOURCETYPE="NODE" SOURCENO="1"
SOURCEPARAM="PRESSURE" />
        <CHART TITLE = "Clv vs Time" XLABEL="Time" YLABEL="Clv"
SOURCETYPE="BLOCK" SOURCENO="6" SOURCEPARAM="OUTPUT"/>
        <CHART TITLE = "Crv vs Time" XLABEL="Time" YLABEL="Crv"
SOURCETYPE="BLOCK" SOURCENO="10" SOURCEPARAM="OUTPUT"/>
        <CHART TITLE = "Plv vs Time" XLABEL="Time" YLABEL="Plv"
SOURCETYPE="BLOCK" SOURCENO="7" SOURCEPARAM="OUTPUT"/>
        <CHART TITLE = "Prv vs Time" XLABEL="Time" YLABEL="Prv"
SOURCETYPE="BLOCK" SOURCENO="11" SOURCEPARAM="OUTPUT"/>
</SIMULATION>

</NCVML>
```

Listing 6.1. EXAMPLE-CV.XML

The first part (<MODEL> .. </MODEL>) of this XML describes model components and blocks and second part (<SIMULATION> .. </SIMULATION>) describes simulation parameters. Model description is composed of three logical parts, which are model parameters description, component descriptions and block descriptions. The first line with <NELEMENTS> defines that the model has 12 components. The 5th line is the

connection description between nodes 1 and 2 and next line is the description of the pressure source (as a part of compliance) component contained by this connection. The keyword PSOURCE defines the component to be pressure source, NO is the unique component number and PARAM is the parameter array to this component as a string. In this model, compliances are modeled as a PSOURCE and a feedback block modulating parameter of the PSOURCE component. The feedback of the left and right ventricles (Blocks 7,11) take output of the blocks 6, 10 respectively in order to calculate parameter values of the PSOURCE components 9 and 12 respectively.

Simulation parameters are described as follows: total simulation time is defined to be 10 sec using the TOTALTIME NCVML keyword. Step duration, in other words the time interval between two sequential steps is defined to be 0.001 sec (1 ms) by using TIMEINTERVAL keyword. As a result of the simulation, the charts of the systemic arterial pressure (Pas) variation and the flow variation passing through aorta are requested to be drawn using CHART keywords.

iii) Initiate the simulation: To be able to initiate the simulation process of this model, first of all user has to acccess client-side html page of the framework. In order to access the html page user must type the following url to the browsers address bar: http://139.179.12.157:8080/ncvsimulation/index.html

Web browser might be Internet Explorer 5.0 or later or Netscape 7.0 or later. Typing the url results in display of the following html page.

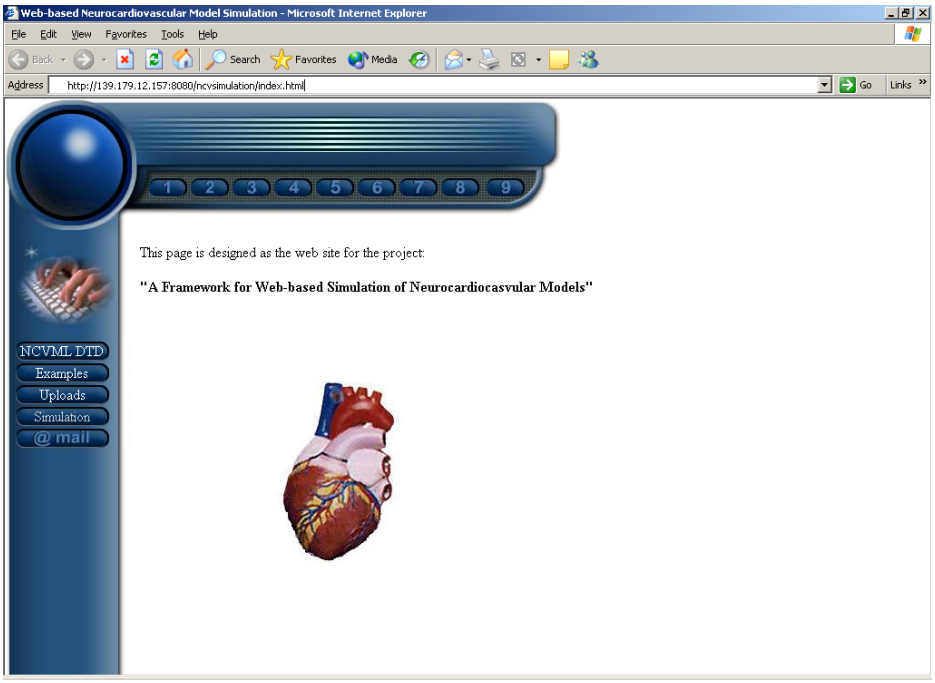


Figure 6.1. NCV Simulation entry page

To access simulation framework, “Simulation” link must be followed on the left menu panel. Following this, user should browse his own disc to locate the model file (XML file), select the file and click “Simulate ” button. This procedure is shown in Figure 6.2.

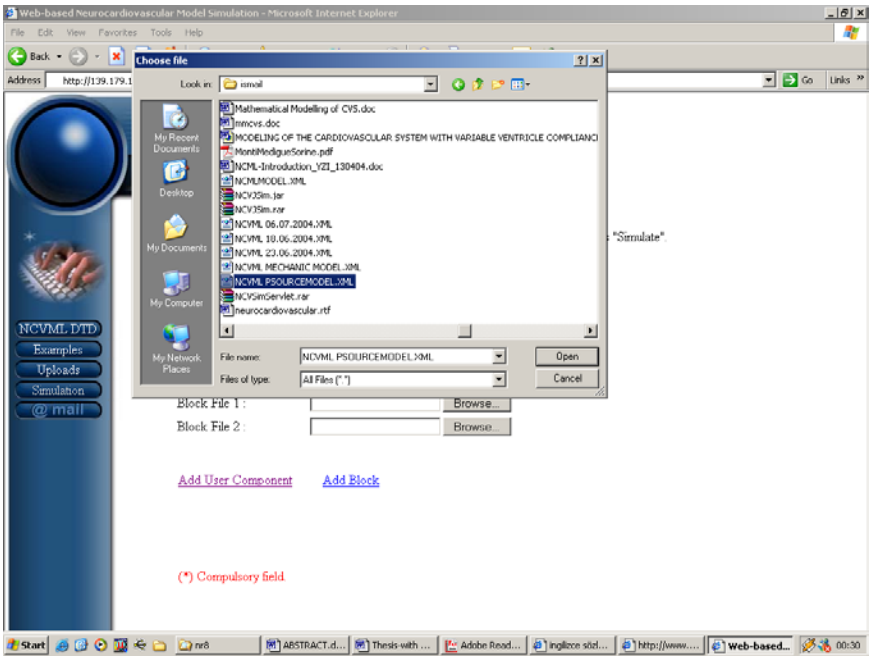
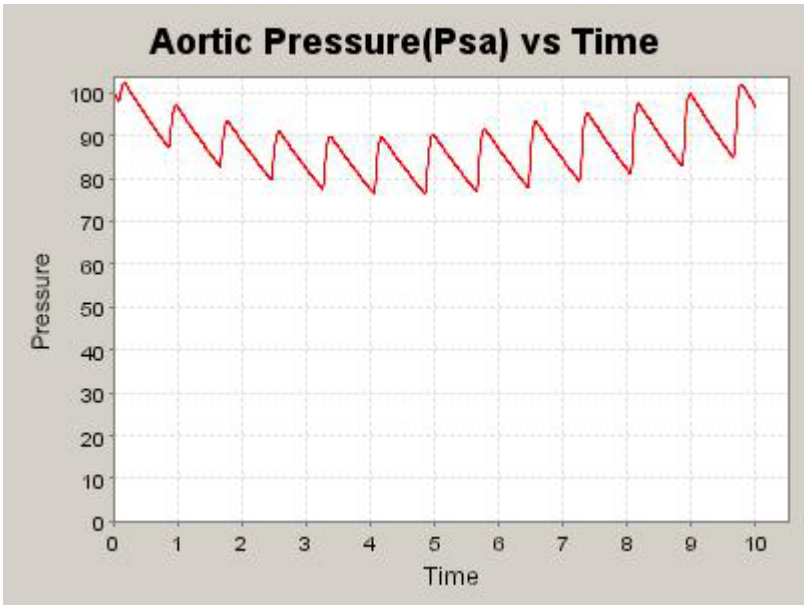


Figure 6.2. Simulation interface

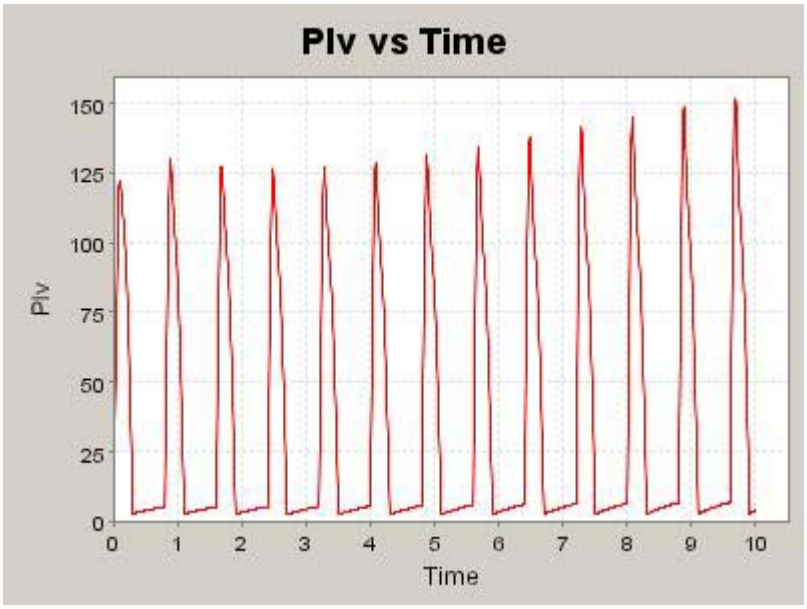
By default, we assumed that model does not incorporates any user defined component or block class. If any, user has to browse and locate his own written and compiled jva classes befor initiating the simulation. By default, there are 2 component and 2 block class browse-and-upload facilities in the html form. If the model containes more than this number of classes users may add additional file browse-and-upload facilities by using “Add Block” and “Add Component” buttons.

iv) Obtaining outputs: At the end of the simulation process requsted charts are displayed simply in chart format on the same html page. In our case, the following charts are displayed as outputs of the simulation.

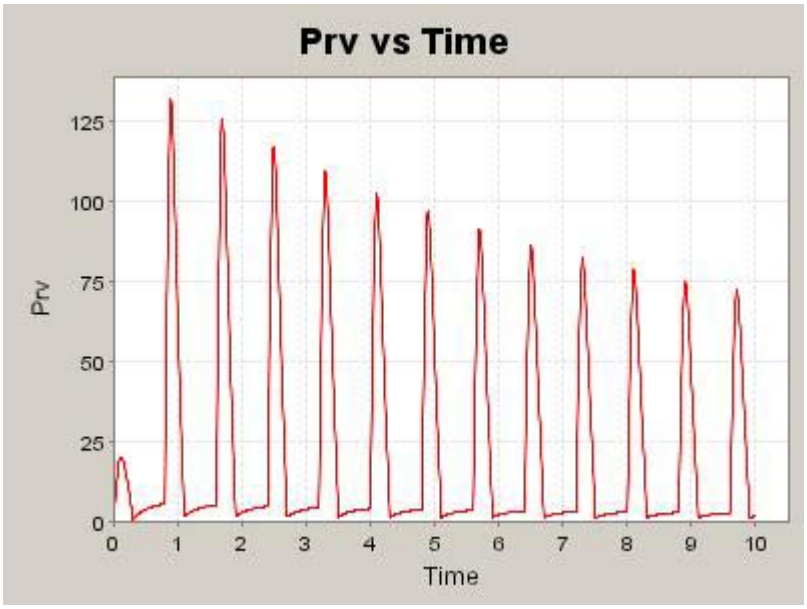
a)


Aortic Pressure(Psa) vs Time
Pressure
Time
0
10
20
30
40
50
60
70
80
90
100
0
1
2
3
4
5
6
7
8
9
10

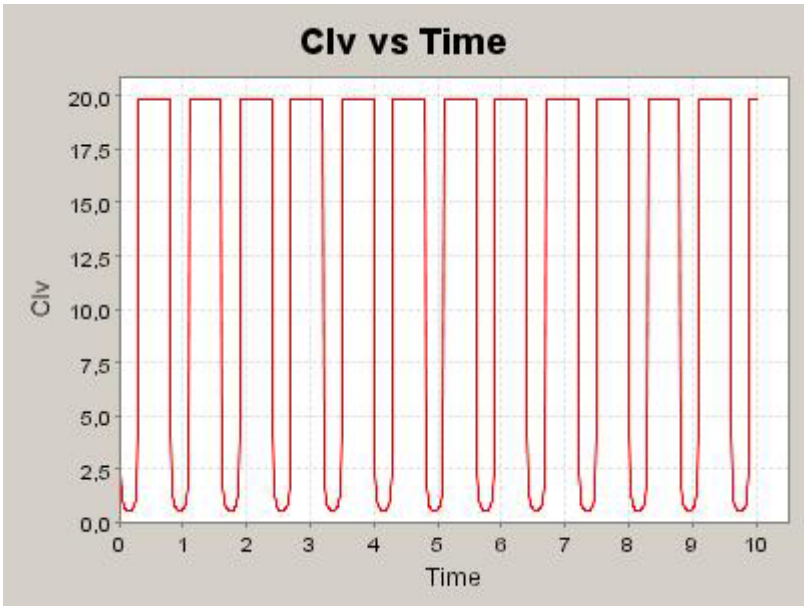
b)


Plv vs Time
Plv
Time
0
25
50
75
100
125
150
0
1
2
3
4
5
6
7
8
9
10

c)


Prv vs Time
Prv
125
100
75
50
25
0
0 1 2 3 4 5 6 7 8 9 10
Time

d)


Clv vs Time
Clv
20,0
17,5
15,0
12,5
10,0
7,5
5,0
2,5
0,0
0 1 2 3 4 5 6 7 8 9 10
Time

e)

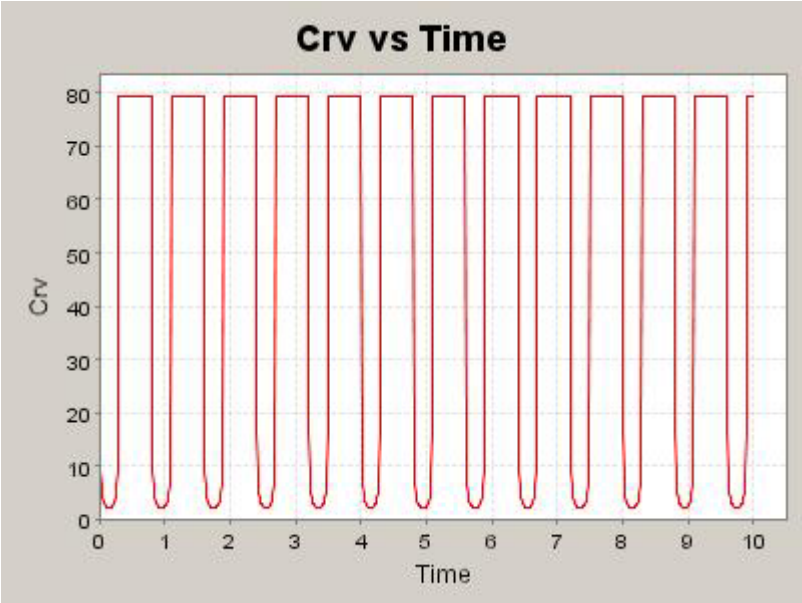


Figure 6.3. Simulation outputs of the cardiovascular model for t = 10 sec

a) Arterial pressure vs time
b) Left ventricle pressure vs time
c) Right ventricle pressure vs time
d) Left ventricle compliance vs time
e) Right ventricle pressure vs time

6.2 Ursino Model

The neurocardiovascular model proposed by Ursino is a comprehensive model with a complex feedback mechanism. We have mentioned this model in Chapter 2, but it seems to be important to give more detailed information about the model. Beside describing the model we are also going to give information

about its NCVML implementation and implementation details. Finally, we are going to give simulation outputs.

The mechanic model of Ursino model differs from our previous model in that in Ursino model the systemic circulation composes 3 compartments. These are systemic arteries (subscript sa), the splanchnic peripheral and venous circulations (subscripts sp and sv, respectively), and the extrasplanchnic peripheral and venous circulations. Ursino also describes left and right heart as pumps. The hydraulic analogue of the mechanical part of the Ursino model is given in Figure 6.4.

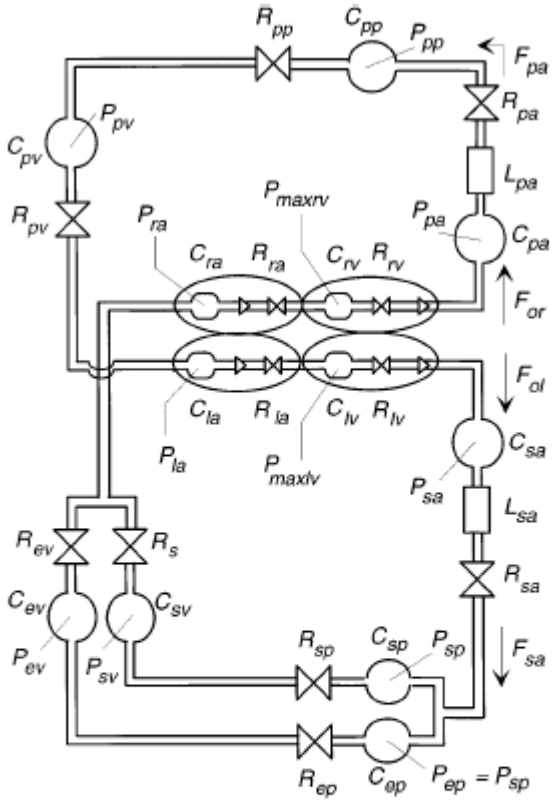


Fig. 1. Hydraulic analog of cardiovascular system. P, pressures; *R*, hydraulic resistances; C, compliances; *L*, inertances; F, flows; sa, systemic arteries; sp, splanchnic peripheral circulation, sv, splanchnic venous circulation; ep, extrasplanchnic peripheral circulation, ev, extrasplanchnic venous circulation; ra, right atrium; rv, right ventricle; pa, pulmonary arteries; pp, pulmonary peripheral circulation; pv, pulmonary veins; la, left atrium; lv, left ventricle; ol and or, output from left and right ventricle, respectively; $P_{max,rv}$ and $P_{max,lv}$, ventricle pressures in isometric conditions.

Figure 6.4. The hydraulic analogue of the Ursino model cardiovascular system (This figure is copied from the online version of Ursino's paper downloaded from http://ajpheart.physiology.org/cgi/content/abstract/275/5/H1733)

Note that Ursino uses hydraulic representation of the model instead of its circuit representation. In this model, all compliances except Csa, and Cpa are compliances with unstressed volume. The following NCVML element-component mapping should be done to describe this mechanic model:

Resistance	- RESISTANCE
Resistance + Valve	- CVALVE
Compliance	- PSOURCE + BComplianceFeedback
Compliance With Unstressed Vol.	- PSOURCE + BComplianceUVFeedback
Comp.with Variable Unstr. Vol.	- PSOURCE + BComplianceVUVFeedback
Inertance	- FSOURCE + BInertanceFeedback

Consider that compliances are modeled as pressure source and a feedback block to this pressure source, which modulates the pressure parameter at the end of each step, inertances are modeled as flow source with a feedback block to this flow source, which modulates the flow variable at each step.

Ursino model proposes feedbacks to T (heart period), Emaxlv (elastance of the left heart), Emaxrv (elastance of the right heart), Rsp, Rep, Vu,sv, Vu,ev parameters. All this parameters are considered to be modulated by sympathetic nerves, and the heart period is proposed to be modulated by the linear combination of both changes caused by sympathetic and parasympahetic (vagal) nerves. The primary input that affects the baroreflex loop is the systemic arterial pressure (Psa). The feedback mechanism is depicted in block diagram in Figure 6.5. For more details about the model see [1].

In our implementation we discarded inertances of the model, because we experienced that the newton-raphson solution of the model with inertance becomes very sensitive to initial values and requires very small step intervals (<0.0005 sec) and very big number of iterations (>20 sec). In this case the system iterates many times to be able to solve nonlinear set of equations which causes very long computation time and the solution even to diverge. Additionally, we decided to perform the heart period and Rsp, Rep feedback

mechanisms and to discard other feedbacks because of the same reasons. Therefore, our NCVML description of the Ursino model turned out to be as given in Listing 6.2

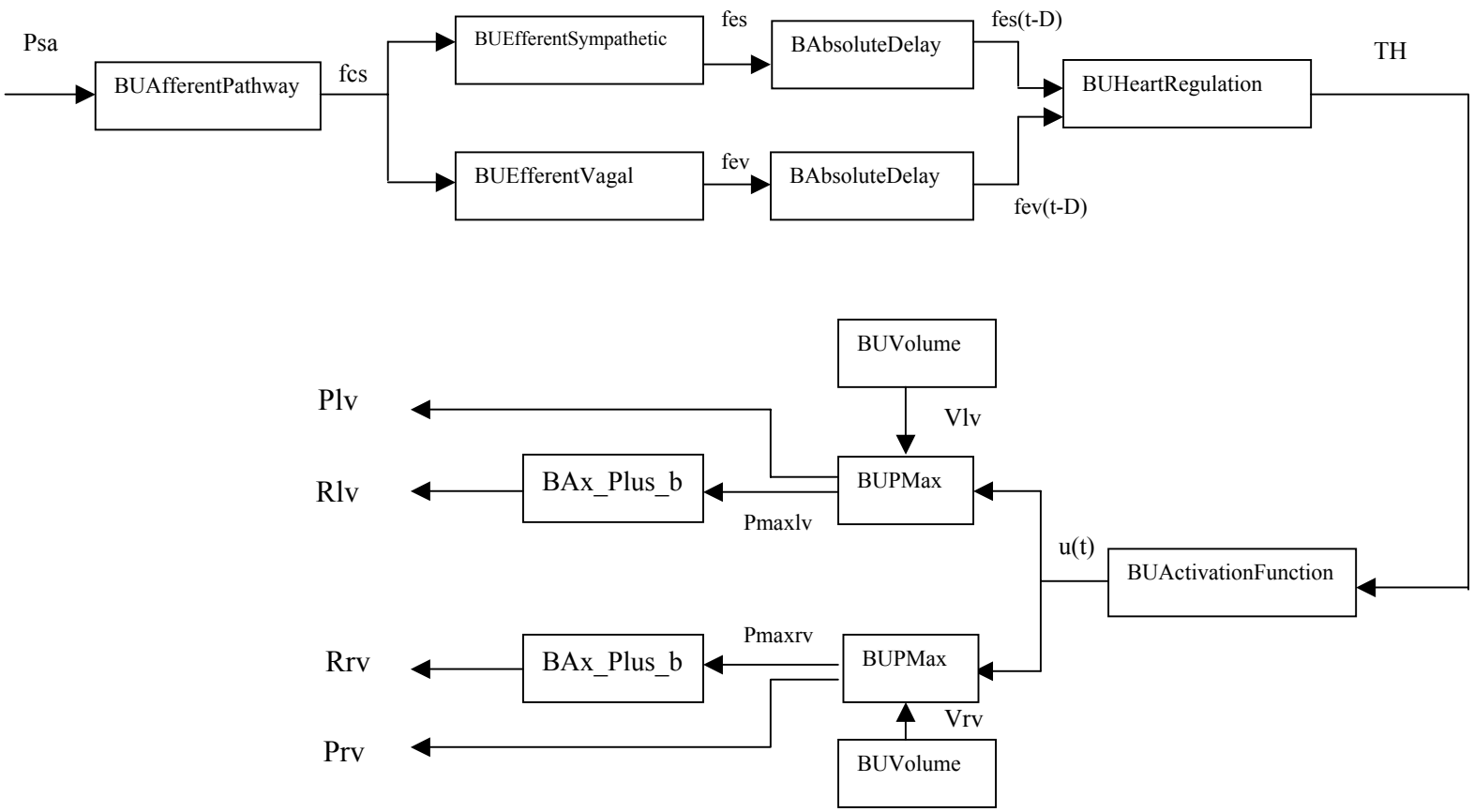


Figure 6.5. Block Diagram of the Ursino Model Feedback Mechanism

```
<NCVML VERSION = "1.0">
<MODEL>
  <NNODES          VALUE = "14" />
  <NELEMENTS       VALUE = "26" />
  <REFERENCENODE    VALUE = "14" />

  <CONNECTION FROM="1" TO="2">
    <RESISTANCE NO="1" PARAM="[0.03]"/>                    <!--LSA-->
  </CONNECTION>
  <CONNECTION FROM="1" TO="14">
    <PSOURCE NO="2" PARAM="[7]"/>                          <!--CSA-->
  </CONNECTION>
  <CONNECTION FROM="2" TO="3">
    <RESISTANCE NO="3" PARAM="[0.03]"/>                    <!--RSA-->
  </CONNECTION>
 <CONNECTION FROM="3" TO="14">
    <PSOURCE NO="4" PARAM="[7]"/>                    <!--CSP-->
  </CONNECTION>
  <CONNECTION FROM="3" TO="4">
    <RESISTANCE NO="5" PARAM="[3.307]"/>                   <!--RSP VAR-->
  </CONNECTION>
  <CONNECTION FROM="4" TO="14">
    <PSOURCE NO="6" PARAM="[7]" />                         <!--CSV  VAR-->
  </CONNECTION>
  <CONNECTION FROM="4" TO="6">
    <RESISTANCE NO="7" PARAM="[0.038]" />                  <!--RSV-->
  </CONNECTION>
  <CONNECTION FROM="3" TO="14">
    <PSOURCE NO="8" PARAM="[7]"/>                          <!--CEP-->
  </CONNECTION>
  <CONNECTION FROM="3" TO="5">
    <RESISTANCE NO="9" PARAM="[1.407]"/>                   <!--REP-->
  </CONNECTION>
  <CONNECTION FROM="5" TO="14">
    <PSOURCE NO="10" PARAM ="[7]"/>                  <!--CEV  VAR-->
  </CONNECTION>
  <CONNECTION FROM="5" TO="6">
    <RESISTANCE NO="11" PARAM="[0.016]"/>            <!--REV  VAR-->
  </CONNECTION>
  <CONNECTION FROM="6" TO="14">
    <PSOURCE NO="12" PARAM="[7]"/>                         <!--CRA-->
  </CONNECTION>
  <CONNECTION FROM="6" TO="7">
    <CVALVE NO="13" PARAM="[2.5/1000 1000]"/>        <!--RRA-->
  </CONNECTION>
  <CONNECTION FROM="7" TO="14">
    <PSOURCE NO="14" PARAM="[1.5]"/>           <!--CRV Right Ventricle-->
  </CONNECTION>
  <CONNECTION FROM="7" TO="8">
    <CVALVE NO="15" PARAM="[1.2/1000 1000]"/>        <!-- RRV Valve -->
  </CONNECTION>
```

```
  <CONNECTION FROM="8" TO="14">
    <PSOURCE NO="16" PARAM="[7]"/>                    <!--CPA-->
  </CONNECTION>
  <CONNECTION FROM="8" TO="9">
    <RESISTANCE NO="17" PARAM="[0.011]"/>             <!--LPA-->
  </CONNECTION>
  <CONNECTION FROM="9" TO="10">
    <RESISTANCE NO="18" PARAM="[0.012]"/>             <!--RPA-->
  </CONNECTION>

  <CONNECTION FROM="10" TO="14">
    <PSOURCE NO="19" PARAM="[7]"/>                    <!--CPP-->
  </CONNECTION>
  <CONNECTION FROM="10" TO="11">
    <RESISTANCE NO="20" PARAM="[0.0894]"/>            <!--RPP-->
  </CONNECTION>
  <CONNECTION FROM="11" TO="14">
    <PSOURCE NO="21" PARAM="[7]"/>                    <!--CPV-->
  </CONNECTION>
  <CONNECTION FROM="11" TO="12">
    <RESISTANCE NO="22" PARAM="[0.0056]"/>            <!--RPV-->
  </CONNECTION>
  <CONNECTION FROM="12" TO="14">
    <PSOURCE NO="23" PARAM="[7]"/>                     <!--CLA-->
  </CONNECTION>
  <CONNECTION FROM="12" TO="13">
    <CVALVE NO="24" PARAM="[2.5/1000 1000]"/>          <!--CVALVE RLA-->
  </CONNECTION>
  <CONNECTION FROM="13" TO="14">
    <PSOURCE NO="25" PARAM="[1.5]"/>                    <!-- CLV Left Ventricle -->
  </CONNECTION>
  <CONNECTION FROM="13" TO="1">
    <CVALVE NO="26" PARAM="[1.485/10000 1000]"/>      <!-- RLV Valve -->
  </CONNECTION>

  <!--LSA feedback-->
  <BLOCK NO = "1" TYPE = "DEFINED" NAME = "BInertanceFeedback" PARAM = "[0.22/1000
0]" TRGTYPE = "BLOCK" TARGET = "1" PARAMNO="1" NOFPREV = "1" PREVVALS="[0]">
     <INPUT SRCTYPE="NODE" SRCNO="1" INPUTTYPE="PRESSURE" NOFPREV="0" />
     <INPUT SRCTYPE="NODE" SRCNO="2" INPUTTYPE="PRESSURE" NOFPREV="0" />
  </BLOCK>

  <!--cSA feedback-->
  <BLOCK NO = "2" TYPE = "DEFINED" NAME = "BCompliance Feedback" PARAM = "[0.28
7]" TRGTYPE = "COMPONENT" TARGET = "2" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="2" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!--CSP feedback-->
  <BLOCK NO = "3" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM =
"[2.05 7 274.4]" TRGTYPE = "COMPONENT" TARGET = "4" PARAMNO="1" NOFPREV = "1"
```

```
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="4" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "4" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM =
"[1.67 7 336.6]" TRGTYPE = "COMPONENT" TARGET = "8" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="8" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!--CRA feedback-->
  <BLOCK NO = "5" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM =
"[31.25 7 25]" TRGTYPE = "COMPONENT" TARGET = "12" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="12" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!--CPA feedback-->
  <BLOCK NO = "6" TYPE = "DEFINED" NAME = "BComplianceFeedback" PARAM = "[0.76
7]" TRGTYPE = "COMPONENT" TARGET = "16" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="16" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!--LPA feedback-->
  <BLOCK NO = "7" TYPE = "DEFINED" NAME = "BInertanceFeedback" PARAM = "[0.18/1000
0]" TRGTYPE = "BLOCK" TARGET = "17" PARAMNO="1" NOFPREV = "1"
PREVVALS="[0]">
     <INPUT SRCTYPE="NODE" SRCNO="8" INPUTTYPE="PRESSURE" NOFPREV="0" />
     <INPUT SRCTYPE="NODE" SRCNO="9" INPUTTYPE="PRESSURE" NOFPREV="0" />
  </BLOCK>

  <!--CPP feedback-->
  <BLOCK NO = "8" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM =
"[5.80 7.5 123]" TRGTYPE = "COMPONENT" TARGET = "19" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="19" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!--CPV feedback-->
  <BLOCK NO = "9" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM =
"[25.37 7.5 120]" TRGTYPE = "COMPONENT" TARGET = "21" PARAMNO="1" NOFPREV =
"1" PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="21" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!--CLA feedback-->
```

```
  <BLOCK NO = "10" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM =
"[19.23 7.5 25]" TRGTYPE = "COMPONENT" TARGET = "23" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="23" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!-- CSV feedback-->
  <BLOCK NO = "11" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM =
"[61.11 7 1121]" TRGTYPE = "COMPONENT" TARGET = "6" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="6" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!-- CEV feedback-->
  <BLOCK NO = "12" TYPE = "DEFINED" NAME = "BComplianceUVFeedback" PARAM = "[50
7 1375]" TRGTYPE = "COMPONENT" TARGET = "10" PARAMNO="1" NOFPREV = "1"
PREVVALS="[7]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="10" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>
  <!-- component feedback end -->

  <BLOCK NO = "13" TYPE = "DEFINED" NAME = "BUAfferentPathway" PARAM = "[2.067
6.37 92 2.52 47.78 11.758]" TRGTYPE = "BLOCK" TARGET = "14" NOFPREV = "0">
        <INPUT SRCTYPE="NODE" SRCNO="1" INPUTTYPE="PRESSURE" NOFPREV="1"
PREVVALS="[0]"/>
  </BLOCK>

  <BLOCK NO = "14" TYPE = "DEFINED" NAME = "BUEfferentSympathetic" PARAM = "[2.10
16.11 0.0675]" TRGTYPE = "BLOCK" TARGET = "15" NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK" SRCNO="13" NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "15" TYPE = "DEFINED" NAME = "BUEfferentVagalPath" PARAM = "[3.2 6.3
25 7.06]" TRGTYPE = "BLOCK" TARGET = "16" NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK" SRCNO="13" NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "16" TYPE = "DEFINED" NAME = "BAbsoluteDelay" PARAM = "[2]"
TRGTYPE = "BLOCK" TARGET = "18" NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK" SRCNO="14" NOFPREV="0" />
  </BLOCK>

  <BLOCK NO = "17" TYPE = "DEFINED" NAME = "BAbsoluteDelayV" PARAM = "[0.2]"
TRGTYPE = "BLOCK" TARGET = "18" NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK" SRCNO="15" NOFPREV="0" />
  </BLOCK>
  <!-- TH regulation -->
  <BLOCK NO = "18" TYPE = "DEFINED" NAME = "BUHeartPeriodRegulation" PARAM = "[-
0.13 2.66 2 0.09 1.5 0.58]" TRGTYPE = "BLOCK" TARGET = "19" NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK" SRCNO="16" NOFPREV="0" />
```

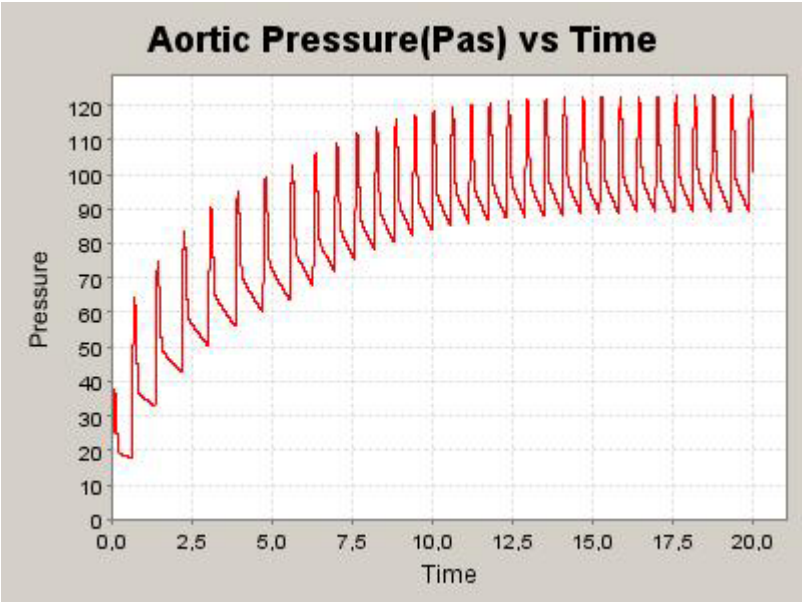
```
        <INPUT SRCTYPE="BLOCK" SRCNO="17" NOFPREV="0" />
  </BLOCK>

  <!-- Activation function - Fi-->
  <BLOCK NO = "19" TYPE = "DEFINED" NAME = "BUActivationFunction" PARAM = "[0.075
0.5]" TRGTYPE = "BLOCK" TARGET = "22"  NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK" SRCNO="18" NOFPREV="0" />
  </BLOCK>


  <!-- Vlv -->
  <BLOCK NO = "20" TYPE = "DEFINED" NAME = "BUVolume" PARAM = "[49 16.77]"
TRGTYPE = "BLOCK" TARGET = "22"  NOFPREV = "1" PREVVALS="[49]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="25" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>

  <!-- Vrv -->
  <BLOCK NO = "21" TYPE = "DEFINED" NAME = "BUVolume" PARAM = "[63 40.8]"
TRGTYPE = "BLOCK" TARGET = "23"  NOFPREV = "1" PREVVALS="[63]">
        <INPUT SRCTYPE="COMPONENT" SRCNO="14" INPUTTYPE="FLOW"
NOFPREV="0" />
  </BLOCK>


  <!-- Pmax,lv -->
  <BLOCK NO = "22" TYPE = "DEFINED" NAME = "BUPMax" PARAM = "[16.77 1.5 0.014
2.95]" TRGTYPE = "COMPONENT" TARGET = "25"  NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK"    SRCNO="19" NOFPREV="0" />
        <INPUT SRCTYPE="BLOCK"    SRCNO="20" NOFPREV="0" />
  </BLOCK>

  <!-- Pmax,rv -->
  <BLOCK NO = "23" TYPE = "DEFINED" NAME = "BUPMax" PARAM = "[40.8 1.5 0.011
1.75]" TRGTYPE = "COMPONENT" TARGET = "14"  NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK"    SRCNO="19"  NOFPREV="0" />
        <INPUT SRCTYPE="BLOCK"    SRCNO="21"  NOFPREV="0" />
  </BLOCK>

  <!-- Rlv feedback-->
  <BLOCK NO = "24" TYPE = "DEFINED" NAME = "BAx_Plus_b" PARAM = "[3.75/10000 0]"
TRGTYPE = "COMPONENT" TARGET = "26" TARGETPARAMNO="1" NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK"  SRCNO="22" NOFPREV="0" />
  </BLOCK>

  <!-- Rrv feedback-->
  <BLOCK NO = "25" TYPE = "DEFINED" NAME = " BAx_Plus_b" PARAM = "[1.4/1000 0]"
TRGTYPE = "COMPONENT" TARGET = "15" TARGETPARAMNO="1" NOFPREV = "0">
        <INPUT SRCTYPE="BLOCK"  SRCNO="23" NOFPREV="0" />
  </BLOCK>


</MODEL>
```

```
<SIMULATION>
        <TOTALTIME VALUE = "10" />
        <TIMEINTERVAL VALUE = "0.001" />
        <CHART TITLE = "Aortic Pressure(Pas) vs Time" XLABEL="Time" YLABEL="Pressure"
SOURCETYPE="NODE" SOURCENO="1" SOURCEPARAM="PRESSURE" />
        <CHART TITLE = "Integrate and Fire vs Time" XLABEL="Time" YLABEL="Ut"
SOURCETYPE="BLOCK" SOURCEPARAM ="OUTPUT" SOURCENO="19" />
        <CHART TITLE = "Plv vs Time" XLABEL="Time" YLABEL="Plv"
SOURCETYPE="BLOCK" SOURCEPARAM ="OUTPUT" SOURCENO="22" />
        <CHART TITLE = "Prv vs Time" XLABEL="Time" YLABEL="Prv"
SOURCETYPE="BLOCK" SOURCEPARAM ="OUTPUT" SOURCENO="23" />
        <CHART TITLE = "Vlv vs Time" XLABEL="Time" YLABEL="Vlv"
SOURCETYPE="BLOCK" SOURCEPARAM ="OUTPUT" SOURCENO="20" />
        <CHART TITLE = "Vrv vs Time" XLABEL="Time" YLABEL="Vrv"
SOURCETYPE="BLOCK" SOURCEPARAM ="OUTPUT" SOURCENO="21" />
</SIMULATION>
</NCVML>
```

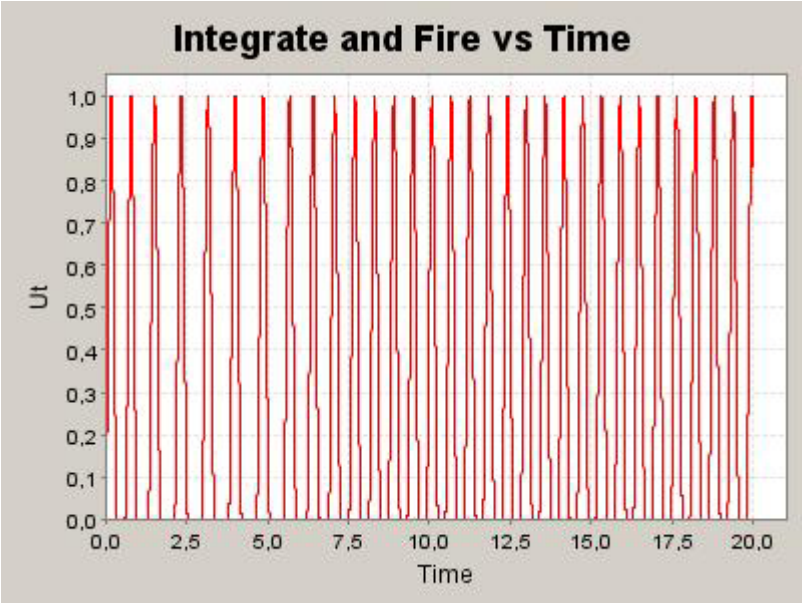
Listing 6.2 URSINOMODEL.XML

Using the same procedure with the previous example, we have initiated a simulation for t=20 sec and dt=0.005 sec and obtained the following charts as the result of the simulation.

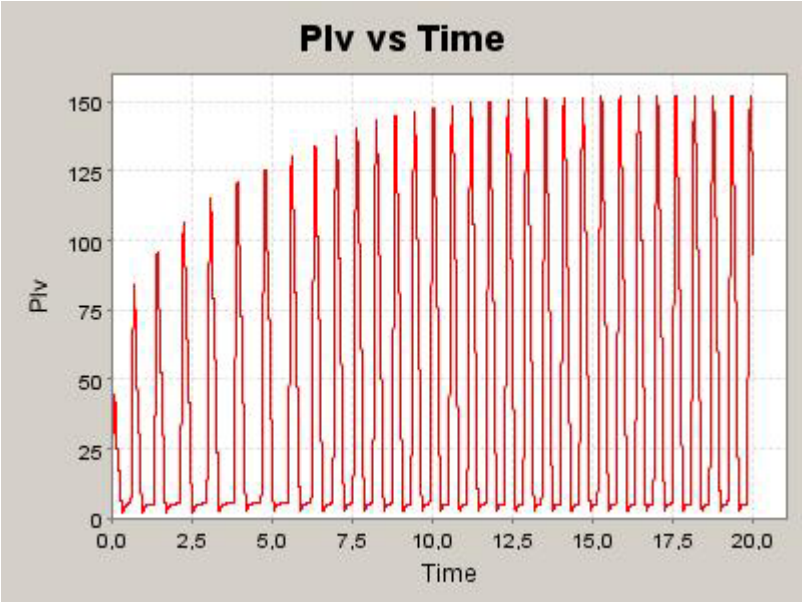
a)



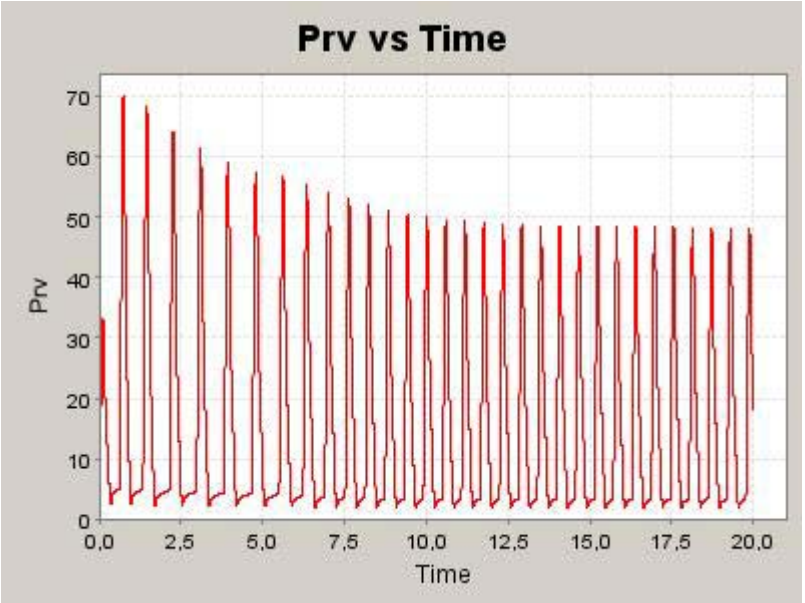
b)


Integrate and Fire vs Time
Ut
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
0,0
2,5
5,0
7,5
10,0
12,5
15,0
17,5
20,0
Time

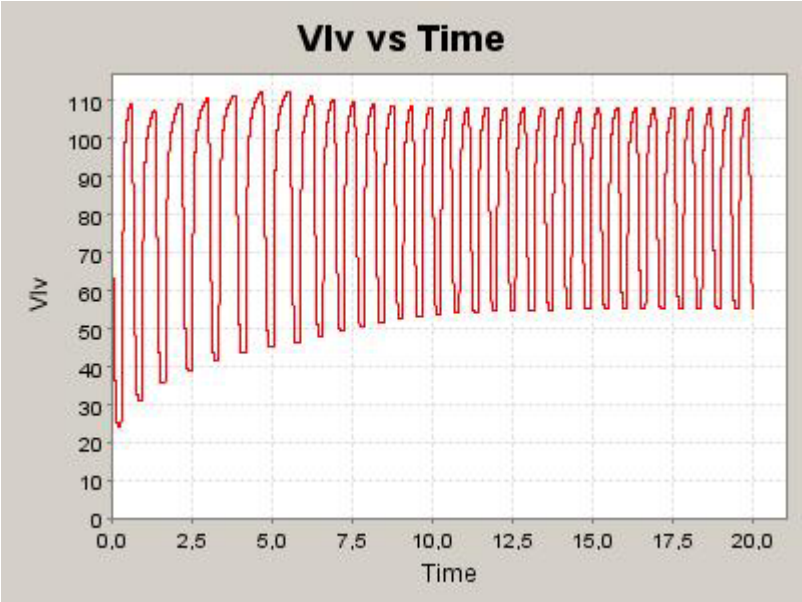
c)


Plv vs Time
Plv
150
125
100
75
50
25
0
0,0
2,5
5,0
7,5
10,0
12,5
15,0
17,5
20,0
Time

d)


Prv vs Time
Prv
70
60
50
40
30
20
10
0
0,0
2,5
5,0
7,5
10,0
12,5
15,0
17,5
20,0
Time

e)


Vlv vs Time
Vlv
110
100
90
80
70
60
50
40
30
20
10
0
0,0
2,5
5,0
7,5
10,0
12,5
15,0
17,5
20,0
Time

f)

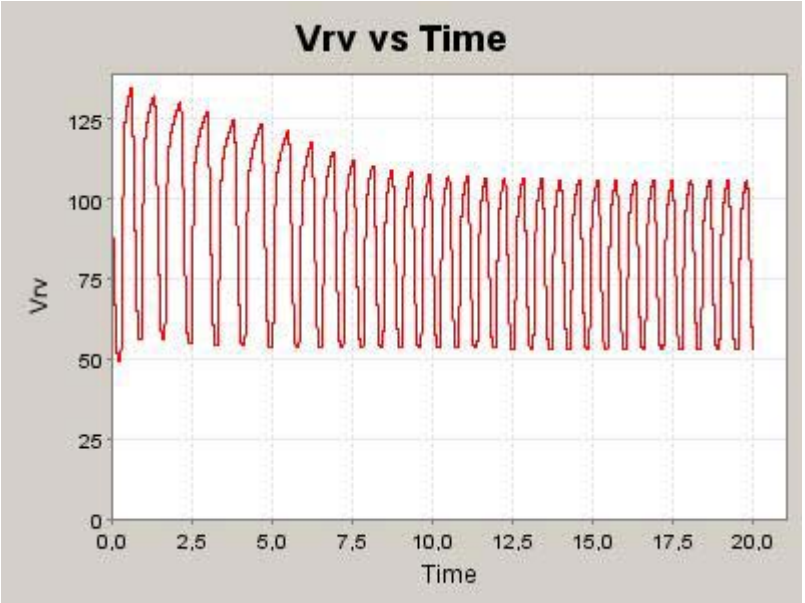


Figure 6.6. Simulation results of the Ursino model for t=20 sec and dt=0.005 sec

a) Aortic Pressure vs time

b) Output of the Integrate and Fire Block (Activation Function)

c) Left ventricle pressure vs time

d) Right ventricle pressure vs time

e) Left ventricle volume vs time

f) Right ventricle pressure vs time

Chapter 7

Conclusions and Future Work

Web-based simulation of neurocardiovascular models offers a great potential in that it provides global access, easy maintenance and platform independence. Moving from desktop to web-based simulation in biomedicine, XML and XML-based technologies play a key role in implementing required architectures. In this thesis, we have designed and implemented a generic, practical and running framework that enables web-based simulation of different types of neurocardiovascular models. We have seen that to implement such a framework the following sub-components are required: a simulation package, a description language and finally supportive software components depending on the architecture. During this thesis, we have included the discussion of the requirements, proposed techniques and implemetation of each above mentioned sub-component in detail. In this context, we have explained NCVJSim java simulator package, NCVSimServlet java servlet, and NCVML XML-based description language concepts. Beside its flexibility, platform independence and simplicity properties, usage of XML brings a very valuable contribution in formation of a standard language for future model exchange infrastructure.

Further research possibilities about this study are as follows:

- Currently, numeric simulation is performed using constant step time intervals. Adoption of adaptive step time intervals is expected to considerably improve the performance of the system. In adaptive step time intervals, the step time interval will be increased where the solution of nonlinear system of equations is slow varying and will be decreased where the solution of the system is fast-varying.

- Newton-Raphson method experiences difficulties in solution of models that contain valves, at the points where the pressure difference across the valve is close to 0, because at these points describing function of the valve has a discontinuous derivative. Introduction and experimenting better mathematical models for valves might improve numeric computation of the system.
- A visual, comprehensive graphical user interface (GUI) to let users design their models visually can be implemented considering web limitations. This might help users design and introduce their models without the knowledge of XML, or NCVML.
- The framework might be extended to include interfaces for web-based model exchange between different applications. This will increase interoperability between different application software of neurocardiovascular simulation and will contribute in construction of a standard, XML-based neurocardiovascular description language.
- The component and block library can be improved such that it evolves in time. Therefore, the framework addresses wider domain of neurocardiovascular models. Besides, simpler methods might be implemented that make user-implemented class utilization during the run-time.

This study leads us to the conclusion that even there are some limitations at this time, particularly in performance, a comprehensive and highly useful framework for web-based simulation of neurocardiovascular models is possible. Furtherwork in this area will pave the way to a more developed framework, which meets additional requirements of the neurocardiovascular simulation.

Bibliography

[1] Vincent C. Rideout, *Mathematical and Comp. Modelling of Physiological Systems*, Prentice Hall Inc, 1991

[2] S. A. Talbot, U. Gessner, *Systems Physiology*, John Wiley and Sons, 1973 (pages 397-426)

[3] Ki H. Chan, Ramakrishna Mukammala, Karin Toska, Thomas J. Mullen, Antonis A. Armoundas, Richard J. Cohen, Linear and Nonlinear System Identification of Autonomic Heart-Rate Modulation, *IEEE Engineering in Medicine and Biology*, September/October (96-105), 1997

[4] R.J. White, D. G. Fitzjerell, R. C. Croston, Fundamentals of Lumped Compartmental Modelling of the Cardiovascular System, *Adv. Cardiovascular Pyhys*. Vol. 5 (Part I), pp 1662-184 (Karger, Basel), 1983

[5] Tina Pro Web-site, http://www.tina.com

[6] Y. Ziya Ider, EE581-Medical Instrumentation Lecture Notes, Available at: http://www.ee.bilkent.edu.tr/~ider [accessed May 3, 2004].

[7] Warner HR, Cox A, A mathematical model of the heart rate control by sympathetic and vagus efferent information. *J. Appl. Physiol*. 17:349-355, 1962

[8] Warner HR, Rusell RO, Effect of combined sympathetic and vagal stimulation on heart rate in the dog. *Circ. Res* 24: 567-573, 1969

[9] Hung-Wen CHIU, Tsair KAO, A Mathematical Model for Autonomic Control of Heart Rate Variation*, IEEE Engineering in Medicine and Biology*, Marc/April 2001, pp 69-76, 2001

[10] Rosenblueth, A., Simeone, F.A. (1934). The interrelations of vagal and accelerator effects on the cardiac rate, *Am. J. Physiol., 110, 42-55*.

[11] R. Mrowka, P.B. Persson, H. Theres, A. Patzak, Blunted Arterial Baroreflex Causes "Pathological" Heart Rate Turbulence, *Am. J. Physiol., Regulatory Integrative Comp. Physiol*. 279; R1171-R1175, 2000

[12] Mauro URSINO, Interaction between carotid baroregulation and the pulsating heart: a mathematical model, *Am. J. Physiol*., Vol. 275 (Heart Circ. Physiol. Vol. 44): H1733-H1747, 1998. Available at : http://ajpheart.physiology.org/cgi/content/abstract/275/5/H1733

[13] Coronary Pressure-Flow Relationships, Avaliable via the web address: http://nsr.bioeng.washington.edu/Courses/BIOEN589_2001/Homeworks/hw3/Pressure-Flow-hw3.htm

[14] Ernest H. Page, Jeffrey M. Opper, Investigating the Application of Web-Based Simulation Principles within the Architecture for a Next-Generation Computer Generated Forces Model, *Elsevier Science*, 1999, Available at: http://www.thesimguy.com/ernie/papers/elsevier/abs.html

[15] Roberto Canonico, Donato Emma, Giorgio Ventre, An XML Based Network Simulation Description Language, Available online at: http://www.grid.unina.it/projects/firb/PubblicazioniURCININA/ 22URCININA.pdf [accessed May 3, 2004]

[16] Bray, Tim, et. al. 2000. Extensible Markup Language (XML) 1.0 (Second Edition). Available online at: http://www.w3.org/TR/2000/REC-xml-20001006
[accessed May 3, 2004].

[17] XML Basics, Available at: http://www.softwareag.com/xml/about/starters.htm

[18] ISO 8879, Information processing -- Text and office systems -- Standard Generalized Markup Language (SGML), 1986, Available at: http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=16387

[19] Terry Quatrani, *Visual Modeling with Rational Rose 2002 and UML, 3rd Edition*, Addison-Wesley (2002) ISBN: 0201729326

[20] Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley (1999).

[21] L.B. Arief and N.A. Speirs, A UML Tool for an Automatic Generation of Simulation Programs, *ACM Proceedings of 2nd International Workshop on Software Performance (WOSP 2000)*,Ottowa, Canada, pp 71-76, September 2000, Available online at: http://homepages.cs.ncl.ac.uk/l.b.arief/home.formal/Papers/wosp2000.pdf [Accessed May 12, 2004]

[22] David Carlson, *Modeling XML Applications with UML*, Addison-Wesley (2001)

[23] World Wide Web Consortium. Extensible Markup Language (XML) 1.0, W3C Recommendation, Avaliable online at: http://www.w3.org/TR/REC-xml/

[24] Michael Hucka, Andrew Finney, Herbert Sauro, Hamid Bolouri, Systems Biology Markup Language (SBML), Available online at: http://www.sbml.org

[25] java.lang Class ClassLoader API Reference, Available at: http://java.sun.com/j2se/1.3/docs/api/java/lang/ClassLoader.htm

[26] Chuck McManis, Take an in-depth look at the Java Reflection API, Available at http://www.javaworld.com/javaworld/jw-09-1997/jw-09-indepth.html

[27] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipies in C: The Art of Scientific Computing (ISBN 0-521-43108-5)*, Cambridge University Press, 1992, Available at: http://www.library.cornell.edu/nr/bookcpdf.html

[28] Brandon E. Taylor, Java Class Loading: The Basics, Available at http://www.developer.com/java/other/article.php/2248831

[29] Java Forum, http://www.jguru.com/forums/view.jsp?EID=1169719

[30] Application Development Forum, http://www.devx.com/tips/Tip/12904

[31] Sun's Servlet homepage, http://java.sun.com/products/servlet/index.jsp

[32] Sun's JSP homepage, http://java.sun.com/products/jsp/

[33] Java Community Process web-site, http://jcp.org/en/home/index

[34] Apache Tomcat official web-site, http://jakarta.apache.org/tomcat/

[35] Hiroshi Marayuma, XML and Java:Developing Web Applications, 1999, Addison-Wesley

[36] com.oreilly.servlet homepage,
http://www.servlets.com/cos/index.html

[37] JFreeChart homepage,
http://www.jfree.org/jfreechart/

[38] Document Object Model (DOM) Level 2 Core Specification, Version 1.0, http://www.w3.org/TR/DOM-Level-2-Core/

Appendix A

Source Codes

In the following section, we provide source codes of the HTML page that enables simulation, Server-side servlet NCVSimServlet, the parser NCVMLDOMParser and NCVJSim package components, respectively.

Simulate.html

```
<html>
<head>
<title>File Upload</title>
<meta http-equiv="Content-Type" content="text/html;">
<SCRIPT LANGUAGE="JavaScript">

   function addUCRow()
   {
          var tbl = document.getElementById('table_id');
          var lastRow = tbl.rows.length;
          // if there's no header row in the table, then iteration = lastRow + 1
          var iteration = lastRow;
          var row = tbl.insertRow(lastRow);

          // left cell
          var cellLeft = row.insertCell(0);
          var  filename = " User Component File :";
          cellLeft.appendChild(document.createTextNode(filename));

          // right cell
          var cellRight = row.insertCell(1);
          var el = document.createElement('input');
          el.setAttribute('type', 'file');
          el.setAttribute('name', 'UCFile' + iteration);
          el.setAttribute('size', '20');
          cellRight.appendChild(el);
   }


   function addBRow()
   {
          var tbl = document.getElementById('table_id');
          var lastRow = tbl.rows.length;
          // if there's no header row in the table, then iteration = lastRow + 1
          var iteration = lastRow;
          var row = tbl.insertRow(lastRow);
```

```
            // left cell
            var cellLeft = row.insertCell(0);
            var  filename = " Block File :";
            cellLeft.appendChild(document.createTextNode(filename));

            // right cell
            var cellRight = row.insertCell(1);
            var el = document.createElement('input');
            el.setAttribute('type', 'file');
            el.setAttribute('name', 'UCFile' + iteration);
            el.setAttribute('size', '20');
            cellRight.appendChild(el);
    }

</SCRIPT>
</head>
<body bgcolor="#FFFFFF" text="#000000"><p><font size="5"color="#FF0000">
<b></b></font></p>
             
To run simulation, select at least your NCVML file using &quot;Browse&quot; button,
then press &quot;Simulate&quot;.<br>
<br>
<br>
<br>

<form name="Form1" enctype="multipart/form-data" method="post"
action="http://139.179.12.157:8080/ncvsimulation/ncvsimservlet">

<table id="table_id" style="border-width: 0">
<tr>
            <td style="border-style: none; border-width:
medium">             &nbs
p;         <input type="submit"value="Simulate"> </td>
    <td style="border-style: none; border-width:
medium">             <input
type="reset" value="Clear"></td>
<tr>
<tr>
            <td style="border-style: none; border-width:
medium">             Model
File<font color="#FF0000">
    (*)    :</font></td>
    <td style="border-style: none; border-width: medium">
    <input type="file" name="File1" size="20" ></td>
</tr>
<tr>
            <td style="border-style: none; border-width:
medium">             User
Component File 1:   </td>
    <td style="border-style: none; border-width: medium">
    <input type="file" name="File4" size="20" ></td>
</tr>
<tr>
            <td style="border-style: none; border-width:
medium">             User
Component File 2: </td>
    <td style="border-style: none; border-width: medium">
```

```
   <input type="file" name="File5" size="20" ></td>
</tr>
<tr>
        <td style="border-style: none; border-width:
medium">             Block File 1
:     </td>
   <td style="border-style: none; border-width: medium">
   <input type="file" name="File2" size="20" ></td>
</tr>
<tr>
        <td style="border-style: none; border-width:
medium">             Block File 2
: </td>
   <td style="border-style: none; border-width: medium">
   <input type="file" name="File3" size="20" ></td>
</tr>

</form>
</table>
<br>
<br>
             <a
href="javascript:addUCRow()">Add User
Component</a>         <a
href="javascript:addBRow()">Add Block </a>
<p> </p>
<p> </p>
<p><font
color="#FF0000">            &nb
sp;
(*) Compulsory field.</font></p>

<p> </p>

</body>
</html>
```

NCVSimServlet.java

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.util.Hashtable;
import java.math.*;
import java.lang.*;
import java.lang.reflect.*;

import org.jfree.data.*;
import org.jfree.chart.*;
import org.jfree.chart.plot.*;

import com.oreilly.servlet.multipart.*;
import com.oreilly.servlet.MultipartRequest;
```

```
/**
 * <p>Title: NCVSimServlet</p>
 * <p>Description: This servlet uploads the NCV model from the client-side,
 * simulates the model and sends formatted simulation results to the client-side</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Bilkent University - EE Department</p>
 * @author Ismail UZUN
 * @version 1.0
 */

public class NCVSimServlet extends HttpServlet {
  private static final String CONTENT_TYPE = "text/html; charset=windows-1254";

 //component definitions
  private final int RESISTANCE    = 1;
  private final int PSOURCE       = 2;
  private final int FSOURCE       = 3;
  private final int VALVE         = 4;
  private final int CVALVE        = 5;
  private final int COMPLIANCE    = 6;
  private final int VCOMPLIANCE   = 7;
  private final int UVCOMPLIANCE  = 8;
  private final int INERTANCE     = 9;
  private final int USERCLASS     = 10;
  private final int BLOCK         = 11;

  XYSeries series = new XYSeries("Average Size");


   //Initialize global variables
  public void init() throws ServletException {
  }




  //Process the HTTP Get request
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>NCVSimServlet</title></head>");
    out.println("<body>");
    out.println("<p>The servlet has received a GET. This is the reply.</p>");
    out.println("</body></html>");
  }

  //Process the HTTP Post request
  public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {

        response.setContentType(CONTENT_TYPE);

            PrintWriter out = null;
            out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>NCVSimServlet</title></head>");
```

```
        out.println("<body>");


/********************** UPLOAD THE XML FILE ****************************/
        String fileName = "";
        String dir = "D:\\ismail\\temp\\";

            try {
              MultipartRequest multi = new MultipartRequest((HttpServletRequest)request, dir, 1024 * 1024);
              fileName = multi.getFilesystemName("File1");

              if (fileName == null)
              {
                 out.println("NCVML File must be uploaded to start simulation process!!");
              }

            } catch (Exception e) {
                        out.println("An error occurred during upload process.");
                        e.printStackTrace(out);
            }

/********************** END XML FILE UPLOAD ***************************/

/********************** PARSE THE MODEL     ***************************/

        //create parser object to parse NCVML document
        NCVMLDOMParser     p        =  new  NCVMLDOMParser();
        SimulationParams   smlParam =  new  SimulationParams();
        Result             rslt     =  new  Result();

         //the path of the input file
        String s = new String(dir + fileName);

        //parse the document
        rslt = p.parse(s);
        if (rslt.bResult == false) //any parsing error occured
        {
            out.println("<p>The following error occured while parsing ");
            out.println(fileName+" :</p>");
            out.println("<p>"+rslt.sResult+" :</p>");
            out.println("</body></html>");
        }

        else    //model parsed successfully!
        {

             //obtain parameters
            p.getParameters(smlParam);

                        double[][] circuitDefinition =  new
                        double[smlParam.nofNodes+smlParam.nofElements][smlParam.nofNodes+smlParam.nofEle
                        ments];
            Vector     BlockVector      =  new Vector();
            Vector     UserClassVector  =  new Vector();

                          //get the circuit definition matrix and block definitions from //the parser object
             circuitDefinition  =  p.getMatrixParameter();
             BlockVector        =  p.getBlockVector();
```

```
        UserClassVector    =  p.getUserClassVector();

        p.freeMemory();

        //Get a class array of number of userclasses size
        Class[] userComponentClass = new Class[UserClassVector.size()];

/********************** CREATE COMPONENT and BLOCK INSTANCES ***********/

                    //create class instances for each circuit element. there might //be more than one from the
                    same element. and element parameters //may change in time. So we need this mechanism.

        ArrayList elementsCollection = new ArrayList();

        for (int i=0; i < smlParam.nofElements ; i++)
        {
          if (circuitDefinition[i][2] == RESISTANCE )
          {
            elementsCollection.add( new Resistance());
          }
          else if (circuitDefinition[i][2] == PSOURCE )
          {
            elementsCollection.add( new PSource());
          }
          else if (circuitDefinition[i][2] == FSOURCE )
          {
            elementsCollection.add( new FSource());
          }
          else if (circuitDefinition[i][2] == COMPLIANCE )
          {
            elementsCollection.add( new Compliance());
          }
          else if (circuitDefinition[i][2] == VCOMPLIANCE )
          {
            elementsCollection.add( new VCompliance());
          }
          else if (circuitDefinition[i][2] == UVCOMPLIANCE )
          {
            elementsCollection.add( new UVCompliance());
          }
          else if (circuitDefinition[i][2] == INERTANCE )
          {
            elementsCollection.add( new Inertance());
          }
          else if (circuitDefinition[i][2] == VALVE )
          {
            elementsCollection.add( new Valve());
          }
          else if (circuitDefinition[i][2] == CVALVE )
          {
            elementsCollection.add( new CValve());
          }
                      //if the code of the component is to be given by the user
          else if (circuitDefinition[i][2] == USERCLASS)
          {
              String componentFileName = "";

              for (int k=0; k < UserClassVector.size(); k++)
```

```
        {
            UserClassDesc bTemp = new UserClassDesc();
            bTemp = (UserClassDesc) UserClassVector.get(k);
            if (bTemp.nIndex == i)
            {
               componentFileName = bTemp.szName;
            }
        }

        File Classfile = new File(dir + componentFileName);
        byte[] data = null;

         try{
               long bytecount = Classfile.length();
               if(bytecount > 0){
                     data = new byte[(int)bytecount];
                                                          FileInputStream in = new
                                                          FileInputStream(Classfile);
                     in.read(data);
                     in.close();
               }
               }catch ( Exception e ){
                     System.out.println("File In-Out Error!!");
         }

        ComponentLoader cl = new ComponentLoader();

        int n = componentFileName.lastIndexOf(".");
        String classname = componentFileName.substring(0,n);

        cl.setClassData(classname,data);

        userComponentClass[i] = cl.loadClass(classname,true);
    }

}

//Get a class array of number of block size
Class[] blockClass = new Class[smlParam.nBlockCount];

for (int i=0; i < BlockVector.size(); i++)
{
   Block bTemp = (Block) BlockVector.get(i);

               //if the block is in the library
   if (bTemp.szType.equals("DEFINED"))
   {
      if (bTemp.szName.equals("BSLV"))
      {
         blockClass[i] = BSLV.class;
      }
      else if (bTemp.szName.equals("BVSLV"))
      {
         blockClass[i] = BVSLV.class;
      }
      else if (bTemp.szName.equals("BInverter"))
      {
         blockClass[i] = BInverter.class;
```

```
}
else if (bTemp.szName.equals("BLinearSystem"))
{
   blockClass[i] = BLinearSystem.class;
}
else if (bTemp.szName.equals("BSSBaroReflex"))
{
   blockClass[i] = BSSBaroReflex.class;
}
else if (bTemp.szName.equals("BUrsino"))
{
   blockClass[i] = BUrsino.class;
}
else if (bTemp.szName.equals("BIntegrateAndFire"))
{
   blockClass[i] = BIntegrateAndFire.class;
}
else if (bTemp.szName.equals("BAbsoluteDelay"))
{
   blockClass[i] = BAbsoluteDelay.class;
}
else if (bTemp.szName.equals("BFilter"))
{
   blockClass[i] = BFilter.class;
}
else if (bTemp.szName.equals("BSimpleIntegral"))
{
   blockClass[i] = BSimpleIntegral.class;
}
else if (bTemp.szName.equals("BComplianceFeedback"))
{
   blockClass[i] = BComplianceFeedback.class;
}
                  else if     (bTemp.szName.equals("BVariableComplianceFeedback"))
{
   blockClass[i] = BVariableComplianceFeedback.class;
}
else if (bTemp.szName.equals("BComplianceUVFeedback"))
{
   blockClass[i] = BComplianceUVFeedback.class;
}
else if (bTemp.szName.equals("BComplianceVUVFeedback"))
{
   blockClass[i] = BComplianceVUVFeedback.class;
}
else if (bTemp.szName.equals("BInertanceFeedback"))
{
   blockClass[i] = BInertanceFeedback.class;
}
else if (bTemp.szName.equals("BUAfferentPathway"))
{
   blockClass[i] = BUAfferentPathway.class;
}
else if (bTemp.szName.equals("BUEfferentSympathetic"))
{
   blockClass[i] = BUEfferentSympathetic.class;
}
else if (bTemp.szName.equals("BUEfferentVagalPath"))
```

```
    {
      blockClass[i] = BUEfferentVagalPath.class;
    }
    else if (bTemp.szName.equals("BUSympatheticRegulation"))
    {
      blockClass[i] = BUSympatheticRegulation.class;
    }
    else if (bTemp.szName.equals("BUHeartPeriodRegulation"))
    {
      blockClass[i] = BUHeartPeriodRegulation.class;
    }
    else if (bTemp.szName.equals("BUActivationFunction"))
    {
      blockClass[i] = BUActivationFunction.class;
    }
    else if (bTemp.szName.equals("BUVolume"))
    {
      blockClass[i] = BUVolume.class;
    }
    else if (bTemp.szName.equals("BUPMax"))
    {
      blockClass[i] = BUPMax.class;
    }
    else if (bTemp.szName.equals("BUVentriclePressure"))
    {
      blockClass[i] = BUVentriclePressure.class;
    }
    else if (bTemp.szName.equals("BAbsoluteDelayV"))
    {
      blockClass[i] = BAbsoluteDelayV.class;
    }
    else if (bTemp.szName.equals("BAbsoluteDelay2"))
    {
      blockClass[i] = BAbsoluteDelay2.class;
    }



}
                    //if the code of the block is to be given by the user
                   else if (bTemp.szType.equals("USERDEFINED"))                    {
     String blockFileName = bTemp.szName;

     File Classfile = new File(dir + blockFileName);
     byte[] data = null;

      try{
            long bytecount = Classfile.length();
            if(bytecount > 0){
                  data = new byte[(int)bytecount];
                                                                                  FileInputStream in = new
                                                          FileInputStream(Classfile);
                  in.read(data);
                  in.close();
            }
            }catch ( Exception e ){
                  System.out.println("File In-Out Error!!");
      }
```

```
            ComponentLoader cl = new ComponentLoader();

            int n = blockFileName.lastIndexOf(".");
            String classname = blockFileName.substring(0,n);

            cl.setClassData(classname,data);

            blockClass[i] = cl.loadClass(classname,true);
        }
    }
/********************** END CREATE COMPONENT and BLOCK INSTANCES ***********/

/********************  SIMULATION  ****************************************/

    //create nonlinear equations solver which utilizes Newton-Raphson algorithm
    NewtonRaphson nrp = new NewtonRaphson();

    //initial x values  -- in Newton-Raphson index starts from 1
            double   x[] = {0,0,0/*2*/,0,0,0,0,0,0,0,0,0,0
            /*13*/,0,0/*15*/,0/*16*/,0,0,0,0,0,0,0/*23*/,0/*24*/,0,0/*26*/,0,0,0,0,0,0,0/*7*/,0,0,0,0,0,0/
            *13*/,0};
    double[] y;


    Vector SeriesVec = new Vector(smlParam.ChartVector.size());
    for (int k = 0; k < smlParam.ChartVector.size(); k++)
    {
       XYSeries tmpSeries = new XYSeries("XX");
       SeriesVec.add(k,tmpSeries);
    }


    //set the parameters and circuit definition to solver
            nrp.setParameters( smlParam.nofNodes, smlParam.nofElements, smlParam.nReferenceNode,
            circuitDefinition, elementsCollection, userComponentClass);

    int STEPS = (int)(smlParam.dTotalTime/smlParam.dt) + 1;

              double[]    result          = new double[smlParam.nofNodes+smlParam.nofElements];
              double[][]  resultBackup    = new
              double[STEPS+1][smlParam.nofNodes+smlParam.nofElements+1];
              double[][]  parameterBackup  = new double[STEPS+1][smlParam.nofElements];
              double[][]  blockOutputBackup = new double[STEPS+1][smlParam.nBlockCount+1];
    double[]   Q          = new double[smlParam.nofElements];

    double fTime = 0;
    BigDecimal bdTime = BigDecimal.valueOf(0);
    BigDecimal bdDt   = new BigDecimal(smlParam.dt);

    for (int i=1; i<= STEPS ; i++)
    {
                            //solve equations using Newton- Raphson algorithm and //get results
        result = nrp.mnewt(100,x,(smlParam.nofNodes+smlParam.nofElements-1),0.0001,0.0001,i);
                            for(int n = 1; n <= smlParam.nofNodes + smlParam.nofElements; n++)
        {
            resultBackup[i][n] = result[n];
        }
```

```
for(int j=0; j < smlParam.nofElements; j++)
{
                    parameterBackup[i][j] = (double) circuitDefinition[j][4];
}

//block operations will be performed using current results. The feedbacks will be applied
blockOperations(blockClass, BlockVector, circuitDefinition, smlParam.nofElements,
resultBackup, blockOutputBackup, parameterBackup, i, smlParam.dt, bdTime.doubleValue());

//charts operations
for (int k = 0; k < smlParam.ChartVector.size(); k++)
{
   XYSeries tmpSeries = (XYSeries)SeriesVec.get(k);
   Chart crt = (Chart) smlParam.ChartVector.get(k);
   if (crt.sSourceType.equals("COMPONENT"))
   {
      if (crt.sSourceParam.equals("FLOW"))
                              tmpSeries.add(bdTime.doubleValue(),
                              result[crt.nSourceNo]);
                        else if (crt.sSourceParam.equals("PRESSURE"))
                              tmpSeries.add(bdTime.doubleValue(),
                  result[smlParam.nofElements+crt.nSourceNo]);
                        else if (crt.sSourceParam.equals("PARAMETER"))
                              tmpSeries.add(bdTime.doubleValue(),
                              parameterBackup[i][crt.nSourceNo-1]);
    }
    else if (crt.sSourceType.equals("NODE"))
    {
      if (crt.sSourceParam.equals("FLOW"))
      {
         double f = 0;
          for (int j=0; j < smlParam.nofElements; j++)
         {
            if (circuitDefinition[j][1] == crt.nSourceNo)
            {
               f += result[(int)circuitDefinition[j][3]];
            }
         }
         tmpSeries.add(bdTime.doubleValue(),f);
      }
      else if (crt.sSourceParam.equals("PRESSURE"))
      {
        //special case in our models..each component has one edge to the reference node
        tmpSeries.add(bdTime.doubleValue(), result[smlParam.nofElements+crt.nSourceNo]);
      }
   }
   else if (crt.sSourceType.equals("BLOCK"))
   {
         tmpSeries.add(bdTime.doubleValue(), blockOutputBackup[i][crt.nSourceNo]);
         //System.out.println(blockOutputBackup[i][crt.nSourceNo]);
   }

   SeriesVec.set(k,tmpSeries);
}

bdTime = bdTime.add(bdDt);
bdTime = bdTime.setScale(6, BigDecimal.ROUND_HALF_UP);
```

```
            }

        //draw charts
        for (int k = 0; k < smlParam.ChartVector.size(); k++)
        {
            Chart crt = (Chart) smlParam.ChartVector.get(k);
            //if the output is requested as file
            if (crt.sOutputType.equals("FILE"))
            {
                String outfileName = "File"+k+".txt";
                String outfileDirName = "C:\\Program Files\\Apache Group\\Tomcat
4.1\\webapps\\ROOT\\files\\";
                FileWriter fout = new FileWriter(outfileDirName+outfileName);

                XYSeries xy = (XYSeries)SeriesVec.get(k);

                for (int m=1; m <= xy.getItemCount(); m++)
                {
                   fout.write((xy.getXValue(m)).toString());
                   fout.write(";");
                   fout.write((xy.getYValue(m)).toString());
                   if (m!= xy.getItemCount())
                     fout.write("&");
                }

                fout.close();
                out.println("<a href = \"../files/"+outfileName+"\">"+outfileName+"</a>");
            }
            //if the output is requested as chart
            else
            {
                XYDataset xyDataset = new XYSeriesCollection((XYSeries)SeriesVec.get(k));

                JFreeChart chart = ChartFactory.createXYLineChart
                          (crt.sTitle,  // Title
                           crt.xLabel,           // X-Axis label
                           crt.yLabel,           // Y-Axis label
                           xyDataset,          // Dataset
                           PlotOrientation.VERTICAL,               // Show legend
                           false,
                           false,false
                          );

                 String chartName = "Chart"+k+".jpeg";
                 String chartDirName = "C:\\Program Files\\Apache Group\\Tomcat
4.1\\webapps\\ROOT\\graphs\\";
                 File fout = new File(chartDirName+chartName);

                 ChartUtilities.saveChartAsJPEG(fout,chart,400,300);

                 out.println("<img src = \"../graphs/"+chartName+"\">");
             }
        }

         out.println("</body></html>");

        }
```

```
 }

  public void  blockOperations(Class[] c, Vector BlockVector, double[][] circuitDefinition, int nElements,
double[][] resultBackup, double[][] blockOutputBackup, double[][]  parameterBackup, int i, double dt, double
fTime)
 {

      Class params[] = {String[].class, double[][].class, double[].class, int.class, double.class};

      Object[]  prmObj =  new Object[5];
      Integer    nStep =  new Integer(i);
      Double         F =  new Double(fTime);

      for (int j = 0; j < BlockVector.size(); j++)
      {

          Block b = new Block();
          b = (Block)BlockVector.get(j);

          //the previous outputs array that will be input is to be formed
          double[] preOut = new double[b.nPreviousOutputs];

          if (i == 1)         //if the first step the initial values are obtained from the XML file(user)
          {
             if (b.nPreviousOutputs > 0)
             {
                for(int h=0; h < b.nPreviousOutputs; h++)
                {
                   preOut[h] = Double.parseDouble(b.prevVal[h]);
                }
            }
          }
          else
          {
             if (b.nPreviousOutputs > 0)
             {
                for(int h=0; h < b.nPreviousOutputs; h++)
                {
                   preOut[h] = blockOutputBackup[i-h-1][b.nNo];
                }
             }
          }
          //

          //the multidimensional-inputs-array is to be formed
          int MAX_HISTORY = 10;
          double[][] inputs = new double[b.getInputSize()][MAX_HISTORY];
          for (int k = 0; k < b.getInputSize(); k++)
          {
              BlockInput bI = new BlockInput();
              bI = (BlockInput)b.getInput(k);

              if (bI.szSrcType.equals("COMPONENT"))
              {
                 if (bI.szInputType.equals("PRESSURE"))   //Pressure
                 {
                    int Node1 = (int)circuitDefinition[bI.nSrc][1];
                    int Node2 = (int)circuitDefinition[bI.nSrc][2];
```

```
                    for(int h=0; h <= bI.nOfPrevValues; h++)
                    {
                        inputs[k][h] = resultBackup[(i-h)][(Node1+nElements-1)] - resultBackup[(i-
h)][(Node2+nElements-1)];
                    }
                }
                else if (bI.szInputType.equals("FLOW"))  //Flow
                {
                    /*for(int h=0; h <= bI.nOfPrevValues; h++)
                    {
                        //inputs[k][h] = resultBackup[bI.nSrc][(i-h)]
                         //07.07.2004 -- block input might be a flow passing through a component
                    }*/
                     inputs[k][0] = resultBackup[i][bI.nSrc];
                }
                else if (bI.szInputType.equals("PARAM"))   //Parameter
                {
                    /*for(int h=0; h <= bI.nOfPrevValues; h++)
                    {
                       inputs[k][h] = parameterBackup[bI.nSrc][i-h];
                    }*/
                    //07.07.2004
                    inputs[k][0] = circuitDefinition[bI.nSrc-1][3+bI.nParamNo];
                }
            }
            else if (bI.szSrcType.equals("NODE"))
            {
                if (bI.szInputType.equals("PRESSURE"))
                {
                     inputs[k][0] = resultBackup[i][bI.nSrc+nElements];
                     int h=1;
                     for (h = 1; ((h <= (i - 1))&&(h <= bI.nOfPrevValues)); h++)
                     {
                         inputs[k][h] = resultBackup[i-h][bI.nSrc+nElements];
                     }

                     int x = 0;
                     for (int m=h; m<= bI.nOfPrevValues; m++)
                     {
                         inputs[k][m] = Double.parseDouble(bI.prevVal[x]);
                         x++;
                     }

                }
            }
            else if (bI.szSrcType.equals("BLOCK"))   //If input is the output of a block
            {
                if (i == 1)
                {
                    inputs[k][0] = blockOutputBackup[i][bI.nSrc];
                    for(int h=1; h <= bI.nOfPrevValues; h++)
                    {
                        inputs[k][h] = Double.parseDouble(bI.prevVal[h-1]);
                    }
                }
                else
                {
```

```
                    for(int h=0; h <= bI.nOfPrevValues; h++)
                    {
                        inputs[k][h] = blockOutputBackup[i-h][bI.nSrc];
                    }
                  }
                }
                if (bI.szSrcType.equals("TIME"))
                {
                  //we dont have to form input because we are already passing the current simulation time to
blocks..
                }
            }

            prmObj[0]  =  b.param; //parameters of the block if exists
            prmObj[1]  =  inputs;  //inputs to that block
            prmObj[2]  =  preOut;  //previous outputs of that block
            prmObj[3]  =  nStep;      //the number of current step
            prmObj[4]  =  F;          //simulation time
            //

            try
            {
                Object o =c[j].newInstance();
                //call setParam metod
                c[j].getMethod("setParam",params).invoke(o, prmObj);

                //call getOutput metod
                            Double d = (Double) c[j].getMethod("getOutput",null).invoke(o,   null);

                //update corresponding component parameter with the output of the block.
                if (b.szTargetType.equals("COMPONENT") == true)
                            circuitDefinition[b.nTarget-1][3+b.nTargetParamNo] = d.doubleValue();

                //save the output of the block
                blockOutputBackup[i][j+1] = d.doubleValue();

            }
            catch (Exception e)
            {
                e.printStackTrace();
            }

        }


    }

    //Clean up resources
    public void destroy() {
    }
}
```

NCVMLDOMParser.java

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
```

```
import javax.xml.parsers.FactoryConfigurationError;
import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

import java.io.File;
import java.io.IOException;
import java.util.*;

import org.w3c.dom.*;

/**
* <p>Title: </p>
* <p>Description: </p>
* <p>Copyright: Copyright (c) 2003</p>
* <p>Company: </p>
* @author unascribed
* @version 1.0
*/

public class NCVMLDOMParser {

   public NCVMLDOMParser()
   {
   }

   private final int RESISTANCE    = 1;
   private final int PSOURCE       = 2;
   private final int FSOURCE       = 3;
   private final int VALVE         = 4;
   private final int CVALVE        = 5;
   private final int COMPLIANCE    = 6;
   private final int VCOMPLIANCE   = 7;
   private final int UVCOMPLIANCE  = 8;
   private final int INERTANCE     = 9;
   private final int USERCLASS     = 10;
   private final int BLOCK         = 11;


  //Global variable definitions
  //Global value so it can be ref'd by the tree-adapter
   static         Document  document;
   private        HashMap   VM          = new HashMap();

   static   private int nofNodes;
   static   private int nofElements;
   static   private int nReferenceNode;
   static   private int nBlockCount;
   static   private double dt;
   static   private double dTotalTime;
   static   private double[] fInitialValues;

   static   double[][]  equationsArray = new double[50][10];
   static   private     Vector    BlockVector  = new Vector();
   static   private     Vector    UserClassVec = new Vector();
   static   private     Vector    ChartVector  = new Vector();
```

```
int             nIdx;
int             nBlockIndx;
int             nUserClassIndx;
static boolean    bComp_or_Inertance = false;
static int        nComp_or_Inertance;


public Result processDOMRecursively(Node node, Result rslt)
{

  //if the current node is an XML element
  if (node instanceof Element)
  {

    if (node.getNodeName().equals("NCVML"))
    {
        System.out.println("NCVML");
        NamedNodeMap nnm = node.getAttributes();
        Attr value = (Attr)nnm.getNamedItem("VERSION");
    }
    else if (node.getNodeName().equals("MODEL"))
    {
        System.out.println("MODEL");
    }
    else if (node.getNodeName().equals("SIMULATION"))
    {
        System.out.println("SIMULATION");
    }
    else if (node.getNodeName().equals("NNODES"))
    {
        NamedNodeMap nnm = node.getAttributes();
        Attr value = (Attr)nnm.getNamedItem("VALUE");

        if (value != null)
           nofNodes = Integer.parseInt(value.getValue());
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid parameter format to NNODES: VALUE must be entered!!";
           System.out.println("Invalid parameter format to NNODES: VALUE must be entered!!");
           return rslt;
        }

        System.out.println("NNODES value: "+value.getValue());
    }
    else if (node.getNodeName().equals("NELEMENTS"))
    {
        NamedNodeMap nnm = node.getAttributes();
        Attr value = (Attr)nnm.getNamedItem("VALUE");

        if (value != null)
           nofElements = Integer.parseInt(value.getValue());
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid parameter format to NELEMENTS: VALUE must be entered!!";
           System.out.println("Invalid parameter format to NELEMENTS: VALUE must be entered!!");
           return rslt;
```

```
    }

    System.out.println("NELEMENTS value: "+value.getValue());
}
else if (node.getNodeName().equals("REFERENCENODE"))
{
    NamedNodeMap nnm = node.getAttributes();
    Attr value = (Attr)nnm.getNamedItem("VALUE");

    if (value != null)
       nReferenceNode = Integer.parseInt(value.getValue());
    else
    {
       rslt.bResult = false;
       rslt.sResult = "Invalid parameter format to REFERENCENODE: VALUE must be entered!!";
       System.out.println("Invalid parameter format to REFERENCENODE: VALUE must be entered!!");
       return rslt;
    }

    System.out.println("REFERENCENODE value: "+value.getValue());
}
else if (node.getNodeName().equals("CONNECTION"))
{
    NamedNodeMap nnm = node.getAttributes();
    Attr from = (Attr)nnm.getNamedItem("FROM");
    Attr to = (Attr)nnm.getNamedItem("TO");

    if ((from != null)&&(to != null))
    {
       equationsArray[nIdx][0] =  Integer.parseInt(from.getValue());
       equationsArray[nIdx][1] =  Integer.parseInt(to.getValue());
    }
    else
    {
       rslt.bResult = false;
       rslt.sResult = "Invalid values for CONNECTION: FROM and TO must be entered!!";
       System.out.println("Invalid values for CONNECTION: FROM and TO must be entered!!");
       return rslt;
    }

    System.out.println("CONNECTION from:"+from.getValue()+" to:"+to.getValue());
}
else if (node.getNodeName().equals("RESISTANCE"))
{
    equationsArray[nIdx][2] = RESISTANCE;

    NamedNodeMap nnm = node.getAttributes();
    Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the block
    Attr params     = (Attr)nnm.getNamedItem("PARAM");    //number of the block

    if (no != null)
       equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
    else
    {
       rslt.bResult = false;
       rslt.sResult = "Invalid attribute to Resistance : NO must be entered!!";
       System.out.println("Invalid attribute to Resistance : NO must be entered!!");
       return rslt;
```

```
        }

        String param = params.getValue();
        param = param.trim();
        String[]  tmpPrmArr;
        if ((param.startsWith("[")) && (param.endsWith("]")))
        {
           int nBegin = param.indexOf("[");
           int nEnd   = param.indexOf("]");
           param      = param.substring(nBegin+1,nEnd);
           tmpPrmArr  = param.split(" ");

           for (int i=0; i < tmpPrmArr.length; i++ )
           {
              if (tmpPrmArr[i].equals("") == false)
              {
                 if (tmpPrmArr[i].indexOf("/") != -1)
                 {
                   String[] nom_denom = tmpPrmArr[i].split("/");
                   tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                 }
                 equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
              }
           }
        }
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid parameter format to RESISTANCE!!";
           System.out.println("Invalid parameter format to RESISTANCE");
           return rslt;
        }

        nIdx++;

        System.out.println("RESISTANCE value: "+no.getValue());
     }
     else if (node.getNodeName().equals("PSOURCE"))
     {
        equationsArray[nIdx][2] = PSOURCE;

        NamedNodeMap nnm = node.getAttributes();
        Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the block
        Attr params     = (Attr)nnm.getNamedItem("PARAM");    //number of the block

        if (no != null)
           equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid attribute to PSOURCE : NO must be entered!!";
           System.out.println("Invalid attribute to PSOURCE : NO must be entered!!");
           return rslt;
        }

        String param = params.getValue();
        param = param.trim();
```

```
            String[]  tmpPrmArr;
            if ((param.startsWith("[")) && (param.endsWith("]")))
            {
               int nBegin = param.indexOf("[");
               int nEnd   = param.indexOf("]");
               param      = param.substring(nBegin+1,nEnd);
               tmpPrmArr  = param.split(" ");

               for (int i=0; i < tmpPrmArr.length; i++ )
               {
                  if (tmpPrmArr[i].equals("") == false)
                  {
                     if (tmpPrmArr[i].indexOf("/") != -1)
                     {
                        String[] nom_denom = tmpPrmArr[i].split("/");
                        tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                     }
                     equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
                  }
               }
            }
            else
            {
               rslt.bResult = false;
               rslt.sResult = "Invalid parameter format to PSOURCE!!";
               System.out.println("Invalid parameter format to PSOURCE");
               return rslt;
            }

            nIdx++;

            System.out.println("PSOURCE NO: "+no.getValue());
         }
         else if (node.getNodeName().equals("FSOURCE"))
         {
            equationsArray[nIdx][2] = FSOURCE;

            NamedNodeMap nnm = node.getAttributes();
            Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the block
            Attr params     = (Attr)nnm.getNamedItem("PARAM");    //number of the block

            if (no != null)
               equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
            else
            {
               rslt.bResult = false;
               rslt.sResult = "Invalid attribute to FSOURCE : NO must be entered!!";
               System.out.println("Invalid attribute to FSOURCE : NO must be entered!!");
               return rslt;
            }

            String param = params.getValue();
            param = param.trim();
            String[]  tmpPrmArr;
            if ((param.startsWith("[")) && (param.endsWith("]")))
            {
               int nBegin = param.indexOf("[");
```

```
            int nEnd   = param.indexOf("]");
            param      = param.substring(nBegin+1,nEnd);
            tmpPrmArr  = param.split(" ");

            for (int i=0; i < tmpPrmArr.length; i++ )
            {
               if (tmpPrmArr[i].equals("") == false)
               {
                  if (tmpPrmArr[i].indexOf("/") != -1)
                  {
                    String[] nom_denom = tmpPrmArr[i].split("/");
                    tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                  }
                  equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
               }
            }
         }
         else
         {
            rslt.bResult = false;
            rslt.sResult = "Invalid parameter format to FSOURCE!!";
            System.out.println("Invalid parameter format to FSOURCE");
            return rslt;
         }

         nIdx++;

         System.out.println("FSOURCE NO: "+no.getValue());
      }
      else if (node.getNodeName().equals("COMPLIANCE"))
      {
         equationsArray[nIdx][2] = COMPLIANCE;

         NamedNodeMap nnm = node.getAttributes();
         Attr no        = (Attr)nnm.getNamedItem("NO");       //number of the block
         Attr params    = (Attr)nnm.getNamedItem("PARAM");    //number of the block

         if (no != null)
            equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
         else
         {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to COMPLIANCE : NO must be entered!!";
            System.out.println("Invalid attribute to COMPLIANCE : NO must be entered!!");
            return rslt;
         }

         String param = params.getValue();
         param = param.trim();
         String[]  tmpPrmArr;
         if ((param.startsWith("[")) && (param.endsWith("")))
         {
            int nBegin = param.indexOf("[");
            int nEnd   = param.indexOf("]");
            param      = param.substring(nBegin+1,nEnd);
            tmpPrmArr  = param.split(" ");
```

```
            for (int i=0; i < tmpPrmArr.length; i++ )
            {
               if (tmpPrmArr[i].equals("") == false)
               {
                  if (tmpPrmArr[i].indexOf("/") != -1)
                  {
                    String[] nom_denom = tmpPrmArr[i].split("/");
                    tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                  }
                  equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
               }
            }
         }
         else
         {
            rslt.bResult = false;
            rslt.sResult = "Invalid parameter format to COMPLIANCE!!";
            System.out.println("Invalid parameter format to COMPLIANCE");
            return rslt;
         }
         bComp_or_Inertance = true;
         nComp_or_Inertance = nIdx;
         nIdx++;

         System.out.println("COMPLAINCE NO: "+no.getValue());
      }
      else if (node.getNodeName().equals("VCOMPLIANCE"))
      {
         equationsArray[nIdx][2] = VCOMPLIANCE;

         NamedNodeMap nnm = node.getAttributes();
         Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the block
         Attr params     = (Attr)nnm.getNamedItem("PARAM");    //number of the block

         if (no != null)
            equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
         else
         {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to VCOMPLIANCE : NO must be entered!!";
            System.out.println("Invalid attribute to VCOMPLIANCE : NO must be entered!!");
            return rslt;
         }

         String param = params.getValue();
         param = param.trim();
         String[]  tmpPrmArr;
         if ((param.startsWith("[")) && (param.endsWith("]")))
         {
            int nBegin = param.indexOf("[");
            int nEnd   = param.indexOf("]");
            param      = param.substring(nBegin+1,nEnd);
            tmpPrmArr  = param.split(" ");

            for (int i=0; i < tmpPrmArr.length; i++ )
            {
               if (tmpPrmArr[i].equals("") == false)
```

```
                {
                    if (tmpPrmArr[i].indexOf("/") != -1)
                    {
                        String[] nom_denom = tmpPrmArr[i].split("/");
                        tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                    }
                    equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
                }
            }
        }
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid parameter format to VCOMPLIANCE!!";
            System.out.println("Invalid parameter format to VCOMPLIANCE");
            return rslt;
        }

        bComp_or_Inertance = true;
        nComp_or_Inertance = nIdx;
        nIdx++;

        System.out.println("VCOMPLIANCE no: "+no.getValue());
    }
    else if (node.getNodeName().equals("UVCOMPLIANCE"))
    {
        equationsArray[nIdx][2] = UVCOMPLIANCE;

        NamedNodeMap nnm = node.getAttributes();
        Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the block
        Attr params     = (Attr)nnm.getNamedItem("PARAM");    //number of the block

        if (no != null)
            equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to UVCOMPLIANCE : NO must be entered!!";
            System.out.println("Invalid attribute to UVCOMPLIANCE : NO must be entered!!");
            return rslt;
        }

        String param = params.getValue();
        param = param.trim();
        String[]  tmpPrmArr;
        if ((param.startsWith("[")) && (param.endsWith("")))
        {
            int nBegin = param.indexOf("[");
            int nEnd   = param.indexOf("]");
            param      = param.substring(nBegin+1,nEnd);
            tmpPrmArr  = param.split(" ");

            for (int i=0; i < tmpPrmArr.length; i++ )
            {
                if (tmpPrmArr[i].equals("") == false)
                {
                    if (tmpPrmArr[i].indexOf("/") != -1)
```

```
                {
                  String[] nom_denom = tmpPrmArr[i].split("/");
                  tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                }
                equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
              }
            }
          }
          else
          {
            rslt.bResult = false;
            rslt.sResult = "Invalid parameter format to UVCOMPLIANCE!!";
            System.out.println("Invalid parameter format to UVCOMPLIANCE");
            return rslt;
          }

          bComp_or_Inertance = true;
          nComp_or_Inertance = nIdx;
          nIdx++;

          System.out.println("VCOMPLIANCE no: "+no.getValue());
        }
        else if (node.getNodeName().equals("INERTANCE"))
        {
          equationsArray[nIdx][2] = INERTANCE;

          NamedNodeMap nnm = node.getAttributes();
          Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the INERTANCE
          Attr params     = (Attr)nnm.getNamedItem("PARAM");    //parameters of the INERTANCE

          if (no != null)
            equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
          else
          {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to INERTANCE : NO must be entered!!";
            System.out.println("Invalid attribute to INERTANCE : NO must be entered!!");
            return rslt;
          }

          String param = params.getValue();
          param = param.trim();
          String[]  tmpPrmArr;
          if ((param.startsWith("[")) && (param.endsWith("")))
          {
            int nBegin = param.indexOf("[");
            int nEnd   = param.indexOf("]");
            param      = param.substring(nBegin+1,nEnd);
            tmpPrmArr  = param.split(" ");

            for (int i=0; i < tmpPrmArr.length; i++ )
            {
              if (tmpPrmArr[i].equals("") == false)
              {
                if (tmpPrmArr[i].indexOf("/") != -1)
                {
                  String[] nom_denom = tmpPrmArr[i].split("/");
```

```
                    tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                  }
                  equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
               }
            }
         }
         else
         {
            rslt.bResult = false;
            rslt.sResult = "Invalid parameter format to INERTANCE!!";
            System.out.println("Invalid parameter format to INERTANCE");
            return rslt;
         }
         bComp_or_Inertance = true;
         nComp_or_Inertance = nIdx;
         nIdx++;

         System.out.println("INERTANCE NO: "+no.getValue());
      }
      else if (node.getNodeName().equals("VALVE"))
      {
         equationsArray[nIdx][2] = VALVE;

         NamedNodeMap nnm = node.getAttributes();
         Attr no        = (Attr)nnm.getNamedItem("NO");       //number of the INERTANCE
         Attr params    = (Attr)nnm.getNamedItem("PARAM");    //parameters of the INERTANCE

         if (no != null)
            equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
         else
         {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to VALVE : NO must be entered!!";
            System.out.println("Invalid attribute to VALVE : NO must be entered!!");
            return rslt;
         }

         String param = params.getValue();
         param = param.trim();
         String[]  tmpPrmArr;
         if ((param.startsWith("[")) && (param.endsWith("]")))
         {
            int nBegin = param.indexOf("[");
            int nEnd   = param.indexOf("]");
            param      = param.substring(nBegin+1,nEnd);
            tmpPrmArr  = param.split(" ");

            for (int i=0; i < tmpPrmArr.length; i++ )
            {
               if (tmpPrmArr[i].equals("") == false)
               {
                  if (tmpPrmArr[i].indexOf("/") != -1)
                  {
                     String[] nom_denom = tmpPrmArr[i].split("/");
                     tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                  }
```

```
                    equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
                }
            }
        }
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid parameter format to VALVE!!";
            System.out.println("Invalid parameter format to VALVE");
            return rslt;
        }
        nIdx++;

        System.out.println("VALVE NO: "+no.getValue());
    }
    else if (node.getNodeName().equals("CVALVE"))
    {
        equationsArray[nIdx][2] = CVALVE;

        NamedNodeMap nnm = node.getAttributes();
        Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the INERTANCE
        Attr params     = (Attr)nnm.getNamedItem("PARAM");    //parameters of the INERTANCE

        if (no != null)
            equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to CVALVE : NO must be entered!!";
            System.out.println("Invalid attribute to CVALVE : NO must be entered!!");
            return rslt;
        }

        String param = params.getValue();
        param = param.trim();
        String[]  tmpPrmArr;
        if ((param.startsWith("[")) && (param.endsWith("]")))
        {
            int nBegin = param.indexOf("[");
            int nEnd   = param.indexOf("]");
            param      = param.substring(nBegin+1,nEnd);
            tmpPrmArr  = param.split(" ");

            for (int i=0; i < tmpPrmArr.length; i++ )
            {
               if (tmpPrmArr[i].equals("") == false)
               {
                  if (tmpPrmArr[i].indexOf("/") != -1)
                  {
                     String[] nom_denom = tmpPrmArr[i].split("/");
                     tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                  }
                  equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
               }
            }
        }
        else
```

```
            {
               rslt.bResult = false;
               rslt.sResult = "Invalid parameter format to CVALVE!!";
               System.out.println("Invalid parameter format to CVALVE");
               return rslt;
            }

            nIdx++;

            System.out.println("CVALVE NO: "+no.getValue());
         }
         else if (node.getNodeName().equals("USERCLASS"))
         {
            equationsArray[nIdx][2] = USERCLASS;

            UserClassDesc  bUSCD = new UserClassDesc();

            NamedNodeMap nnm = node.getAttributes();
            Attr no         = (Attr)nnm.getNamedItem("NO");       //number of the INERTANCE
            Attr params     = (Attr)nnm.getNamedItem("PARAM");    //parameters of the INERTANCE
            Attr name       = (Attr)nnm.getNamedItem("NAME");

            if (no != null)
               equationsArray[nIdx][3] = Double.parseDouble(no.getValue());
            else
            {
               rslt.bResult = false;
               rslt.sResult = "Invalid attribute to USERCLASS : NO must be entered!!";
               System.out.println("Invalid attribute to USERCLASS : NO must be entered!!");
               return rslt;
            }

            String param = params.getValue();
            param = param.trim();
            String[]  tmpPrmArr;
            if ((param.startsWith("[")) && (param.endsWith("]")))
            {
               int nBegin = param.indexOf("[");
               int nEnd   = param.indexOf("]");
               param      = param.substring(nBegin+1,nEnd);
               tmpPrmArr  = param.split(" ");

               for (int i=0; i < tmpPrmArr.length; i++ )
               {
                  if (tmpPrmArr[i].equals("") == false)
                  {
                     if (tmpPrmArr[i].indexOf("/") != -1)
                     {
                        String[] nom_denom = tmpPrmArr[i].split("/");
                        tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                     }
                     equationsArray[nIdx][i+4] = Double.parseDouble(tmpPrmArr[i]);
                  }
               }
            }
            else
            {
```

```
            rslt.bResult = false;
            rslt.sResult = "Invalid parameter format to USERCLASS!!";
            System.out.println("Invalid parameter format to USERCLASS");
            return rslt;
        }

        bUSCD.nNo    = Integer.parseInt(no.getValue());
        bUSCD.nIndex = nIdx;
        bUSCD.szName = name.getValue();

        UserClassVec.add(nUserClassIndx,bUSCD);

        nUserClassIndx++;
        nIdx++;

        System.out.println("USERCLASS NO" + no.getValue());
    }

    else if (node.getNodeName().equals("BLOCK"))
    {
        NamedNodeMap nnm   = node.getAttributes();
        Attr no            = (Attr)nnm.getNamedItem("NO");       //number of the block
        Attr type          = (Attr)nnm.getNamedItem("TYPE");     //DEFINED ya da USERDEFINED olabilir
        Attr name          = (Attr)nnm.getNamedItem("NAME");
        Attr prm           = (Attr)nnm.getNamedItem("PARAM");    //parameters of that block
        Attr trgtype       = (Attr)nnm.getNamedItem("TRGTYPE");  //type of the target ELEMENT/BLOCK
        Attr targetno      = (Attr)nnm.getNamedItem("TARGET");   //number of the target
        Attr targetParamNo = (Attr)nnm.getNamedItem("TARGETPARAMNO");   //number of the target
        Attr nOfPrev       = (Attr)nnm.getNamedItem("NOFPREV");  //number of previous outputs required

        Block b = new Block();

        if (no != null)
            b.nNo              = Integer.parseInt(no.getValue());
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to Block : NO must be entered!!";
            System.out.println("Invalid attribute to Block : NO must be entered!!");
            return rslt;
        }

        if (type != null)
            b.szType           = type.getValue();
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to Block : TYPE must be entered!!";
            System.out.println("Invalid attribute to Block : TYPE must be entered!!");
            return rslt;
        }

        if (name != null)
            b.szName           = name.getValue();
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to Block : NAME must be entered!!";
```

```
            System.out.println("Invalid attribute to Block : NAME must be entered!!");
            return rslt;
        }

        if (prm != null)
        {
            String sParams      = prm.getValue();
            sParams             = sParams.trim();

            String[]  tmpPrmArr;

            if ((sParams.startsWith("[")) && (sParams.endsWith("]")))
            {
               int nBegin = sParams.indexOf("[");
               int nEnd   = sParams.indexOf("]");
               sParams    = sParams.substring(nBegin+1,nEnd);
               tmpPrmArr  = sParams.split(" ");

               Vector    tmp = new Vector();

               for (int i=0; i < tmpPrmArr.length; i++ )
               {
                  if (tmpPrmArr[i].equals("") == false)
                  {
                     if (tmpPrmArr[i].indexOf("/") != -1)
                     {
                        String[] nom_denom = tmpPrmArr[i].split("/");
                        tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                     }
                     tmp.add(tmpPrmArr[i]);
                  }
               }

               String[]  paramArr = new String[tmp.size()];

               for (int j=0; j<tmp.size(); j++)
                  paramArr[j] = tmp.get(j).toString();

               b.param = paramArr;     //assign to block class param


            }
            else
            {
               rslt.bResult = false;
               rslt.sResult = "Invalid format for BLOCK parameters : PARAM";
               System.out.println("Invalid format for BLOCK parameters : PARAM");
               return rslt;
            }
        }

        if (trgtype != null)
           b.szTargetType     = trgtype.getValue();
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid attribute to Block : TRGTTYPE must be entered!!";
```

```
            System.out.println("Invalid attribute to Block : TRGTTYPE must be entered!!");
            return rslt;
        }

        if (targetno != null)
            b.nTarget           = Integer.parseInt(targetno.getValue());
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to Block : TARGET must be entered!!";
            System.out.println("Invalid attribute to Block : TARGET must be entered!!");
            return rslt;
        }

        if (targetParamNo != null)
            b.nTargetParamNo    = Integer.parseInt(targetParamNo.getValue());
        else
            b.nTargetParamNo    = 1;

        System.out.println("Block " + no.getValue());

        if (nOfPrev != null)
            b.nPreviousOutputs  = Integer.parseInt(nOfPrev.getValue());
        else
        {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to Block : NOFPREV must be entered!!";
            System.out.println("Invalid attribute to Block : NOFPREV must be entered!!");
            return rslt;
        }


        if ((Integer.parseInt(nOfPrev.getValue()) != 0)&&(nOfPrev != null))
        {
            Attr prevVals       = (Attr)nnm.getNamedItem("PREVVALS");
            String sPreVals     = prevVals.getValue();
            sPreVals            = sPreVals.trim();

            String[]  tmpPrmArr;
            Vector    tmp = new Vector();

            if ((sPreVals.startsWith("[")) && (sPreVals.endsWith("]")))
            {
                int nBegin = sPreVals.indexOf("[");
                int nEnd   = sPreVals.indexOf("]");
                sPreVals   = sPreVals.substring(nBegin+1,nEnd);
                tmpPrmArr  = sPreVals.split(" ");

                for (int i=0; i < tmpPrmArr.length; i++ )
                {
                    if (tmpPrmArr[i].equals("") == false)
                    {

                        if (tmpPrmArr[i].indexOf("/") != -1)
                        {
                            String[] nom_denom = tmpPrmArr[i].split("/");
                            tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
```

```
                    }
                    tmp.add(tmpPrmArr[i]);
                  }
                }

                String[]  preValArr = new String[tmp.size()];

                for (int j=0; j<tmp.size(); j++)
                  preValArr[j] = tmp.get(j).toString();


                b.prevVal = preValArr;

              }
              else
              {
                rslt.bResult = false;
                rslt.sResult = "Invalid format for initial values";
                System.out.println("Invalid format for initial values");
                return rslt;
              }
            }

            int i = 0;
            for (Node child = node.getFirstChild(); child != null ; child = child.getNextSibling())
            {
              if (child.getNodeName().equals("INPUT"))
              {
                NamedNodeMap nm = child.getAttributes();

                Attr srcType      = (Attr)nm.getNamedItem("SRCTYPE");
//ELEMENT/NODE/BLOCK/TIME olabilir
                Attr srcNo        = (Attr)nm.getNamedItem("SRCNO");         // source no
                Attr srcInputType  = (Attr)nm.getNamedItem("INPUTTYPE");   // type of the input type :
V(voltage)/I(current)/B(block output)/P(parameter) olabilir
                Attr srcParamNo    = (Attr)nm.getNamedItem("PARAMNO");   //
                Attr nofprev      = (Attr)nm.getNamedItem("NOFPREV");     // number of previous values
required

                BlockInput bI     = new BlockInput();

                if (srcType != null)
                    bI.szSrcType      = srcType.getValue();
                else
                {
                    rslt.bResult = false;
                    rslt.sResult = "Invalid attribute to Block Input : SRCTYPE must be entered!!";
                    System.out.println("Invalid attribute to Block Input : SRCTYPE must be entered!!");
                    return rslt;
                }

                if (srcNo != null)
                    bI.nSrc           = Integer.parseInt(srcNo.getValue());

                if (srcInputType != null)
                  bI.szInputType    = srcInputType.getValue();

                if ((srcParamNo != null)&&(srcInputType.getValue().equals("PARAM")))
```

```
                    bI.nParamNo       = Integer.parseInt(srcParamNo.getValue());
                  else
                     bI.nParamNo      = 1;

                  if (nofprev != null)
                    bI.nOfPrevValues  = Integer.parseInt(nofprev.getValue());

                  if ((nofprev != null)&&(Integer.parseInt(nofprev.getValue()) != 0))
                  {
                     Attr prevVals     = (Attr)nm.getNamedItem("PREVVALS");
                     String sPreVals   = prevVals.getValue();
                     sPreVals  = sPreVals.trim();

                      String[]  tmpPrmArr;
                      Vector    tmp = new Vector();

                      if ((sPreVals.startsWith("[")) && (sPreVals.endsWith("]")))
                      {
                         int nBegin = sPreVals.indexOf("[");
                         int nEnd   = sPreVals.indexOf("]");
                         sPreVals   = sPreVals.substring(nBegin+1,nEnd);
                         tmpPrmArr  = sPreVals.split(" ");

                         for (int k=0; k < tmpPrmArr.length; k++ )
                         {
                            if (tmpPrmArr[k].equals("") == false)
                            {
                               if (tmpPrmArr[k].indexOf("/") != -1)
                               {
                                  String[] nom_denom = tmpPrmArr[k].split("/");
                                  tmpPrmArr[k] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                               }
                               tmp.add(tmpPrmArr[k]);
                            }
                         }

                         String[]  preValArr = new String[tmp.size()];

                         for (int j=0; j<tmp.size(); j++)
                            preValArr[j] = tmp.get(j).toString();

                         bI.prevVal = preValArr;

                      }
                      else
                      {
                         rslt.bResult = false;
                         rslt.sResult = "Invalid format for initial values";
                         System.out.println("Invalid format for initial values");
                         return rslt;
                      }
                  }

                  b.setInput(i,bI);   //set the input class to the block vector

                  i++;
               }
```

```
    }

    BlockVector.add(nBlockIndx,b);

    nBlockIndx++;
    nBlockCount++;
}
else if (node.getNodeName().equals("TOTALTIME"))
{
    NamedNodeMap nnm = node.getAttributes();
    Attr value = (Attr)nnm.getNamedItem("VALUE");

    System.out.println("TOTALTIME value: "+value.getValue());

    if (value != null)
       dTotalTime = Double.parseDouble(value.getValue());
    else
    {
       rslt.bResult = false;
       rslt.sResult = "Invalid attribute to TOTALTIME : VALUE must be entered";
       System.out.println("Invalid attribute to TOTALTIME : VALUE must be entered");
       return rslt;
    }

}
else if (node.getNodeName().equals("TIMEINTERVAL"))
{
    NamedNodeMap nnm = node.getAttributes();
    Attr value = (Attr)nnm.getNamedItem("VALUE");

    System.out.println("TIMEINTERVAL value: "+value.getValue());

    if (value != null)
        dt = Double.parseDouble(value.getValue());
    else
    {
       rslt.bResult = false;
       rslt.sResult = "Invalid attribute to TIMEINTERVAL : VALUE must be entered";
       System.out.println("Invalid attribute to TIMEINTERVAL : VALUE must be entered");
       return rslt;
    }
}
else if (node.getNodeName().equals("INITIALVALUES"))
{
    NamedNodeMap nnm = node.getAttributes();
    Attr value = (Attr)nnm.getNamedItem("VALUE");

    System.out.println("INITIALVALUES value: "+value.getValue());

    if (value != null)
    {
        String sPreVals     = value.getValue();
        sPreVals            = sPreVals.trim();
        String[]  tmpPrmArr;


        if ((sPreVals.startsWith("[")) && (sPreVals.endsWith("")))
```

```
            {
               int nBegin = sPreVals.indexOf("[");
               int nEnd   = sPreVals.indexOf("]");
               sPreVals   = sPreVals.substring(nBegin+1,nEnd);
               tmpPrmArr  = sPreVals.split(" ");

               Vector    tmp = new Vector();

               for (int i=0; i < tmpPrmArr.length; i++ )
               {
                  if (tmpPrmArr[i].equals("") == false)
                  {

                    if (tmpPrmArr[i].indexOf("/") != -1)
                     {
                       String[] nom_denom = tmpPrmArr[i].split("/");
                       tmpPrmArr[i] =
String.valueOf(Double.parseDouble(nom_denom[0])/Double.parseDouble(nom_denom[1]));
                     }
                     tmp.add(tmpPrmArr[i]);
                  }
               }

               fInitialValues = new double[tmp.size()];

               for (int j=0; j<tmp.size(); j++)                    fInitialValues[j] =
Double.parseDouble(tmp.get(j).toString());
            }
         }
         else
         {
            rslt.bResult = false;
            rslt.sResult = "Invalid attribute to INITIALVALUES : VALUE must be entered";
            System.out.println("Invalid attribute to INITIALVALUES : VALUE must be entered");
            return rslt;
         }
      }
      else if (node.getNodeName().equals("CHART"))
      {
         NamedNodeMap nnm = node.getAttributes();
         Attr title      = (Attr)nnm.getNamedItem("TITLE");
         Attr xlabel     = (Attr)nnm.getNamedItem("XLABEL");
         Attr ylabel     = (Attr)nnm.getNamedItem("YLABEL");
         Attr sourcetype  = (Attr)nnm.getNamedItem("SOURCETYPE");
         Attr sourceno    = (Attr)nnm.getNamedItem("SOURCENO");
         Attr sourceparam = (Attr)nnm.getNamedItem("SOURCEPARAM");
         Attr outputtype  = (Attr)nnm.getNamedItem("OUTPUTTYPE");

         Chart ch = new Chart();

         System.out.println("CHART value: "+title.getValue());

         if (title != null)
            ch.sTitle = title.getValue();

         if (xlabel != null)
            ch.xLabel = xlabel.getValue();
```

```
        if (ylabel != null)
            ch.yLabel = ylabel.getValue();

        if (sourcetype != null)
            ch.sSourceType = sourcetype.getValue();
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid attribute to CHART : SOURCETYPE must be entered";
           System.out.println("Invalid attribute to CHART : SOURCETYPE must be entered");
           return rslt;
        }

        if (sourceno != null)
            ch.nSourceNo = Integer.parseInt(sourceno.getValue());
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid attribute to CHART : SOURCENO must be entered";
           System.out.println("Invalid attribute to CHART : SOURCENO must be entered");
           return rslt;
        }

        if (sourceparam != null)
            ch.sSourceParam = sourceparam.getValue();
        else
        {
           rslt.bResult = false;
           rslt.sResult = "Invalid attribute to CHART : SOURCEPARAM must be entered";
           System.out.println("Invalid attribute to CHART : SOURCEPARAM must be entered");
           return rslt;
        }

        if (outputtype != null)
            ch.sOutputType = outputtype.getValue();

        ChartVector.add(ch);
    }

    else if (node.getNodeName().equals("INPUT"))
    {
    }
    else
    {
        rslt.bResult = false;
        rslt.sResult = "Error in the input file";
        System.out.println("Error in the input file");
        return rslt;
    }
}


for (Node child = node.getFirstChild(); child != null ; child = child.getNextSibling())
{
   rslt = processDOMRecursively(child,rslt);
   if (rslt.bResult == false)
     return rslt;
}
```

```
  rslt.bResult = true;
  rslt.sResult = "File parsed sucsessfully!!";
  return rslt;

}


public Result parse(String filename)
{
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   //factory.setValidating(true);
   //factory.setNamespaceAware(true);
   try {
      DocumentBuilder builder = factory.newDocumentBuilder();
      document = builder.parse( new File(filename) );

   } catch (SAXException sxe) {
      // Error generated during parsing)
      Exception  x = sxe;
      if (sxe.getException() != null)
          x = sxe.getException();
      x.printStackTrace();

   } catch (ParserConfigurationException pce) {
       // Parser with specified options can't be built
       pce.printStackTrace();

   } catch (IOException ioe) {
      // I/O error
      ioe.printStackTrace();
   }

   System.out.println("File is validated and parsed sucessfully!!");

   Node node = document.getDocumentElement();

   NCVMLDOMParser DOMParse = new NCVMLDOMParser();
   nIdx          = 0;
   nBlockIndx    = 0;
   nBlockCount   = 0;
   nUserClassIndx = 0;

   Result rslt = new Result();
   //call processDOMRecursively to process sub-elements
   rslt = DOMParse.processDOMRecursively(node,rslt);

   return rslt;

} // parse

public void getParameters(int nOfNodes, int nOfElements,int nGround)
{
     nOfNodes = nofNodes;
     nOfElements = nofElements;
     nGround = nReferenceNode;
}
```

```
//get all simulation parameters
public void getParameters(SimulationParams smlParam)
{
    smlParam.nofNodes           = nofNodes;
    smlParam.nofElements        = nofElements;
    smlParam.nReferenceNode     = nReferenceNode;
    smlParam.dt                 = dt;
    smlParam.dTotalTime         = dTotalTime;
    smlParam.nBlockCount        = nBlockCount;
    smlParam.fInitialValues     = fInitialValues;
    smlParam.bComp_or_Inertance = bComp_or_Inertance;
    smlParam.ChartVector        = ChartVector;
}

public double[][] getMatrixParameter()
{
    return equationsArray;
}

public Vector getBlockVector()
{
    return BlockVector;
}

public Vector getUserClassVector()
{
    return UserClassVec;
}

public int getBlockCount()
{
    return nBlockCount;
}

public int getNodeParameter()
{
    return nofNodes;
}

public int getElementParameter()
{
    return nofElements;
}

public int getReferenceNodeParameter()
{
    return nReferenceNode;
}

public boolean isThereComplianceorInertance()
{
    return bComp_or_Inertance;
}

public int getCorIPosition()
{
    return nComp_or_Inertance;
}
```

```
   //clear data structures for the next simulation
   public void freeMemory()
   {
      BlockVector  = null;
      UserClassVec = null;
      ChartVector  = null;
      BlockVector  = new Vector();
      UserClassVec = new Vector();
      ChartVector  = new Vector();
      equationsArray = null;
      equationsArray = new double[50][10];
   }
}
```

ComponentLoader.java

```
import java.util.*;
import java.io.*;
import java.lang.*;

public class ComponentLoader extends ClassLoader{

   Hashtable       local       = new Hashtable();
   Hashtable       localClasses = new Hashtable();


   public ComponentLoader()
   {
   }

   public void setClassData(String name,byte Agent_code[])
   {
         localClasses.put(name,Agent_code);
   }

   public synchronized Class defineClass(String name,byte data[])
   {
      Class c = defineClass(name,data,0,data.length);
         return c;
   }

   public Class loadClass(String name,boolean resolve)
   {
      Object o = null;
      Class c = null;

         try {
                  if ((o = localClasses.get(name)) == null)
                           return findSystemClass(name);
         } catch (Exception e)
         {
                  e.printStackTrace();
         }


         byte data[] = (byte[])o;
```

```
            c = defineClass(name, data, 0, data.length);
            resolveClass(c);

            return c;
    }

}
```

Result.java

```
public class Result
{

   //default constructor
   public Result()
   {
     bResult = true;
     sResult = "";
   }

   public boolean bResult;
   public String  sResult;
}
```

SimulationParams.java

```
import java.util.*;

public class SimulationParams
{
        public SimulationParams()
        {
           nofNodes          = 0;
           nofElements       = 0;
           nReferenceNode    = 0;
           nBlockCount       = 0;
           dt                = 0.01;
           dTotalTime        = 5;
           bComp_or_Inertance = false;
        }

        public int    nofNodes;
        public int    nofElements;
        public int    nReferenceNode;
        public int    nBlockCount;
        public double   dt;
        public double   dTotalTime;
        public double[] fInitialValues;
        public boolean bComp_or_Inertance;
        public Vector  ChartVector;
}
```

UserClassDesc.java

```
public class UserClassDesc
{

  public int nNo;
  public int nIndex;
  public String szName;

}
```

Block.java

```
import BlockInput;
import java.util.*;

public class Block {

  public int nNo;
  public int nTarget;
  public int nPreviousOutputs;
  public int nOfInputs;
  public String szType;
  public String szTargetType;
  public int nTargetParamNo = 1;
  public String szName;


  public String[] prevVal;
  public String[] param;

  Vector inputVector = new Vector();

  public void setInput(int i,BlockInput bI)
  {
    inputVector.add(i,bI);
  }

  public BlockInput getInput(int i)
  {
    return (BlockInput) inputVector.get(i);
  }

  public int getInputSize()
  {
    return inputVector.size();
  }
}
```

BlockInput.java

```
public class BlockInput
{

  public  String   szSrcType;;
  public  int      nSrc;
  public  String   szInputType;;
  public  int      nParamNo;
```

```
  public  int      nOfPrevValues;
  public  String[] prevVal;
}
```

Chart.java

```
public class Chart
{
   public String sTitle      = "";
   public String xLabel      = "";
   public String yLabel      = "";
   public String sSourceType  = "";
   public int    nSourceNo    = 0;
   public String sSourceParam = "";
   public String sOutputType  = "";


}
```

NCVJSim

PSource.java

```
class PSource extends NCVMLComponent
{

        private double P;
      private int paramCount = 1;
        String description = "This component is Voltage Source. It has only one parameter. The voltage-
current equation is V = I*R";

      public void setParam(double param[])
        {
                P = param[0];
        }

      public double getParam( )
        {
                return P;
        }

      public int getParamCount()
        {
            return paramCount;
        }

      public String getDescription()
        {
            return description;
        }

      public double getPressure(double F)
        {
                return P;
        }
```

```
	public double getFlow(double P)
		{
			return (double)-999;
		}
}
```

FSource.java

```
class FSource  extends NCVMLComponent
{

		private double I;
		private int paramCount = 1;
		String description = "This component is Current Source. It has only one parameter.";

	public void setParam(double param[])
		{
			I = param[0];
		}

	public double getParam( )
		{
			return I;
		}

	public int getParamCount()
		{
			return paramCount;
		}

	public String getDescription()
		{
			return description;
		}

	public double getPressure(double F)
	{
		 return (double)-999;
	}

	public double getFlow(double P)
		{
			return I;
		}

}
```

Resistance.java

```
class Resistance extends NCVMLComponent
{

		private double R;
	private int paramCount = 1;
		String description = "This component is Resistance. It has only one parameter. The pressure flow
equation is P = F*R";
```

```
	public void setParam(double param[])
	    {
		R = param[0];
	    }

	public double getParam( )
	    {
		return R;
	    }

	public int getParamCount()
	    {
	        return paramCount;
	    }

	public String getDescription()
	    {
	        return description;
	    }

	public double getPressure(double F)
	    {
		return (F*R);
	    }

	public double getFlow(double P)
	    {
		return (P/R);
	    }

}
```

Valve.java

```
class Valve extends NCVMLComponent
{

	private double Pth;
	private double Fo;
	private int paramCount = 2;
	String description = "This component is Valve. It has two parameters. The pressure-flow equation is F =
Fo (e^P/Pth - 1)";

	public void setParam(double param[])
	    {
		Pth = param[0];
		Fo  = param[1];
	    }

	public double getParam( )
	    {
		return Pth;
	    }

	public int getParamCount()
	    {
```

```
            return paramCount;
        }

    public String getDescription()
        {
            return description;
        }

    public double getPressure(double F)
        {
              double tmp1 = (F/Fo)+1;
              double tmp2 = Math.log(tmp1);

               return tmp2*Pth;
        }

    public double getFlow(double P)
        {
                double tmp = (P/Pth);
                 return (Fo*(Math.exp(tmp) - 1));
    }

}
```

CValve.java

```
class CValve extends NCVMLComponent
{

        private double Rd;
    private double Roff;
    private int paramCount = 2;
        String description = "This component is a Compound Valve. It is the combination of a Resistance and
a Valve. It has two parameters.";

    public void setParam(double param[])
        {
                Rd    = param[0];
                Roff  = param[1];
        }

    public double getParam( )
        {
                return Roff;
        }

    public int getParamCount()
        {
                 return paramCount;
        }

    public String getDescription()
        {
            return description;
        }

    public double getPressure(double F)
```

```
        {
         if (F >= 0)
           return  F*Rd;
         else
           return  F*Roff;
        }

    public double getFlow(double P)
        {
         if ( P >= 0 )
           return  (1/Rd)*P;
         else
           return  (1/Roff)*P;
    }
}
```

NCVMLBlock.java

```
public abstract class NCVMLBlock
{

   protected String   description;

   public  abstract  String  getDescription();
   public  abstract  void    setParam(String[] param,double[][] input, double[] preOut, int nStep, double fTime);
   public  abstract  double   getOutput();

}
```

NCVMLComponent.java

```
public abstract class NCVMLComponent
{

      protected  int       paramCount;
      protected  String   description;

      public  abstract  void    setParam(double param[]);
      public  abstract  String  getDescription();
      public  abstract  double   getPressure(double F);
      public  abstract  double   getFlow(double P);
}
```

BSLV.java

```
public class BSLV extends NCVMLBlock
{

  private double SLD;
  private double SLS;
  private double T1;
  private double T;
  private double fTime;
  String description = "";
```

```
  public String getDescription()
  {
     return description;
  }


  public void setParam(String[] param,double[][] input, double[] preOut, int nStep, double fTime)
  {
     this.SLD    = Double.parseDouble(param[0]);
     this.SLS    = Double.parseDouble(param[1]);
     this.T1     = Double.parseDouble(param[2]);
     this.T      = Double.parseDouble(param[3]);
     this.fTime  = fTime;
  }

  public double getOutput()
  {
     double Slv = 0;

     Slv += this.SLD;

     double t = Math.IEEEremainder(this.fTime,this.T);

     if ((0 < t ) && (t <= this.T1 ))
     {
        Slv  += this.SLS*(Math.sin(2*(Math.PI)*(1/0.6)*t));
        return Slv;
     }
     else
     {
        return Slv;
     }
  }

}
```

BComplianceFeedback.java

```
public class BComplianceFeedback extends NCVMLBlock
{

  private String description = "";
  private double Vo;
  private double Po;
  private double F;
  private double C;
  private double dt;

  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
   {

     if (parm4 == 1)
     {
         dt = 0;
         Po = Double.parseDouble(parm1[1]);
     }
     else
```

```
    {
       dt  = parm5/(parm4-1);
       Po  = parm3[0];
    }

    C   = Double.parseDouble(parm1[0]);
    F   = parm2[0][0];
    Vo  = Po*C;
 }

 public String getDescription()
 {
    return description;
 }

 //V=Vo + F*dt, P=V/C, this P will update the PSOURCE P parameter
 public double getOutput()
 {
    double V = Vo + F*dt;

    return V/C;
 }
```

BVariableComplianceFeedback.java

```
public class BVariableComplianceFeedback extends NCVMLBlock
{

 private String description = "";
 private double Vo;
 private double P;
 private double Po;
 private double F;
 private double Cold;
 private double Cnew;
 private double dt;
 private double V;

 public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
  {

    if (parm4 == 1)
    {
      dt    = 0;
      Cold  = Double.parseDouble(parm1[0]);
      Po    = Double.parseDouble(parm1[1]);
    }
    else
    {
       dt  = parm5/(parm4-1);
       Po  = parm3[0];
       Cold = parm2[1][1];
    }

    F    = parm2[0][0];
    Cnew = parm2[1][0];
```

```
    Vo = Cold*Po;
  }

  public String getDescription()
  {
    return description;
  }

  public double getOutput()
  {

    V  = Vo + F*dt;

    P  = V/Cnew;

    return P;
  }

}
```

BComplianceUVFeedback

```
public class BComplianceUVFeedback extends NCVMLBlock
{

  private String description = "";
  private double Vo;
  private double Vu;
  private double Po;
  private double F;
  private double C;
  private double dt;

  //C=parm[0],Po=parm[1],Vu=parm1[2] for normal compliances Vo=0, C=parm1[1], F=input[0][0]
  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
   {

    if (parm4 == 1)
    {
       dt = 0;
       Po = Double.parseDouble(parm1[1]);
    }
    else
    {
      dt  = parm5/(parm4-1);
      Po  = parm3[0];
    }

    C   = Double.parseDouble(parm1[0]);
    Vu  = Double.parseDouble(parm1[2]);
    F   = parm2[0][0];
    Vo  = Vu + Po*C;
  }

  public String getDescription()
  {
```

```
    return description;
  }

  //V=Vo + F*dt, P=(V-Vu)/C, this P will update the PSOURCE P parameter
  public double getOutput()
  {
    double V = Vo + F*dt;

    return (V-Vu)/C;
  }
}
```

BInertanceFeedback.java

```
public class BInertanceFeedback extends NCVMLBlock
{
  private String description = "";
  private double Fo;
  private double P1;
  private double P2;
  private double L;
  private double dt;

  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
   {

    if (parm4 == 1)
    {
      dt = 0;
      Fo = Double.parseDouble(parm1[1]);
    }
    else
    {
       dt  = parm5/(parm4-1);
       Fo  = parm3[0];
    }

    L  = Double.parseDouble(parm1[0]);

    P1  = parm2[0][0];
    P2  = parm2[1][0];
  }

  public String getDescription()
  {
    return description;
  }

  //V=Vo + P*dt, P=V/L, this F will update the FSOURCE F parameter
  public double getOutput()
  {
    double F = Fo + (P1-P2)*dt/L;

    return F;
  }
}
```

BInverter.java

```
public class BInverter extends NCVMLBlock
{
  private double f;
  String description = "";

  public String getDescription()
  {
    return description;
  }

  public void setParam(String[] param,double[][] input, double[] preOut, int nStep, double fTime)
  {
    this.f    = input[0][0];
  }

  public double getOutput()
  {
    return (1/f);
  }

}
```

BFilter.java

```
public class BAbsoluteDelay extends NCVMLBlock
{
    private String description =  "";
    private int    paramCount;
    private double  delay;
    private double  nStep;
    private double  dt;
    private double  fTime;
    private static int    indx;

    static double[] delayArr = new double[10000];

    public String getDescription()
    {
       return description;
    }

    public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
    {
       if (parm4 == 1)
       {
          this.dt = 0;
          indx    = 0;
       }
       else
       {
          this.nStep       = parm4;
```

```
            this.dt         = parm5/(nStep-1);
            int index       = (int) Math.IEEEremainder((double)indx,(double)10000);
            delayArr[index]  = parm2[0][0];
            delay           = Double.parseDouble(parm1[0]);
            indx++;
         }

      }

      public double getOutput()
      {

         int Indx    = (int)(nStep - (int)Math.ceil(this.delay/this.dt));

         if (Indx <= 0)
         {
            return 0;
         }
         else
         {
            int retIndx = (int)Math.IEEEremainder((double)Indx, (double) 10000);
            return delayArr[retIndx];
         }
      }

}
```

BIntegrateAndFire.java

```
public class BIntegrateAndFire extends NCVMLBlock
{

  private static double u;
  private double T;
  private double dt;
  String description = "";

  public String getDescription()
  {
     return description;
  }

  public void setParam(String[] param,double[][] input, double[] preOut, int nStep, double fTime)
  {
     T  = input[0][0];
     if (nStep == 1)
     {
       this.dt = 0;
       u = 0;
     }
     else
       this.dt = fTime/(nStep-1);
  }

  public double getOutput()
  {
     u += dt/T;
```

```
    if (u >=1)
    {
      u = 0;
      return 1;
    }
    else
    {
      return 0;
    }
  }
}
```

BPulseFunction.java

```
public class BPulseFunction extends NCVMLBlock
{

  private String description = "This function generates pulse series at given intervals";
  private double T;
  private double fTime;
  private double dt;
  private int   stepno;
  private int   pulseStep;


  public String getDescription()
  {
     return description;
  }

  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
  {
     this.T      =Double.parseDouble(parm1[0]);
     this.stepno = parm4;
     this.fTime  = parm5;

     if ((this.stepno ==1))
     {
       this.dt   = 0;
       pulseStep = 1;
     }
     else
     {
       this.dt   = this.fTime/(this.stepno-1);
       pulseStep = (int)Math.round(this.T/this.dt);
     }
  }

  public double getOutput()
  {
     double rem = Math.IEEEremainder(this.stepno,this.pulseStep);
     if (rem == 0)
        return 1;
     else
        return 0;
  }
```

```
}
```

BSimpleIntegral.java

```
public class BSimpleIntegral extends NCVMLBlock
{
 private String description = "";
 private static double Vo;
 private double F;
 private double dt;

 public double getOutput()
 {
    double V = Vo + F*dt;
    Vo = V;

    return V;
 }
 public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
  {

    if (parm4 == 1)
    {
      dt = 0;
      Vo = Double.parseDouble(parm1[0]);
    }
    else
    {
       dt  = parm5/(parm4-1);
    }

    F  = parm2[0][0];
 }
 public String getDescription()
 {
    return description;
 }
}
```

BSinus.java

```
public class BSinus extends NCVMLBlock
{

 private String description = "This block generates sin waveform!";
 private double f;      //frequency of the sin function
 private double fTime;  //current simulation time

 public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
 {
    this.f     = Double.parseDouble(parm1[0]);
    this.fTime = parm5;
 }
```

```
 public String getDescription()
 {
    return description;
 }

 public double getOutput()
 {
     double val = Math.sin(2*(Math.PI)*(this.f)*(this.fTime));
     return val;
 }

}
```

BSquareWaveFunction.java

```
public class BSquareWaveFunction extends NCVMLBlock
{
 private String description = "This block generates square wave signal!";
 private double a;
 private double b;
 private double T;
 private double fTime;
 private double dt;
 private int   stepno;
 private int   pulseStep;
 private static int   level = 0; // low (=a)


 public String getDescription()
 {
    return description;
 }

 public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
 {
    this.a      = Double.parseDouble(parm1[0]);
    this.b      = Double.parseDouble(parm1[1]);
    this.T      = Double.parseDouble(parm1[2]);
    this.stepno = parm4;
    this.fTime  = parm5;

    if ((this.stepno ==1))
    {
      this.dt = 0;
      pulseStep = 1;
    }
    else
    {
      this.dt = this.fTime/(this.stepno-1);
      pulseStep = (int)Math.round(this.T/this.dt);
    }
 }

 public double getOutput()
 {
```

```
     double rem = Math.IEEEremainder(this.stepno,this.pulseStep);
    if (rem == 0)
    {
       if (level == 0)
       {
        level = 1;  // =b
        return b;
       }
       else
       {
        level = 0;
        return a;
       }
    }
    else
    {
       if (level == 0)
       {
        return a;
       }
       else
       {
        return b;
       }
    }
 }

}
```

BStepFunction.java

```
public class BStepFunction extends NCVMLBlock
{

 private String description = "This block generates a step function.";
 private double a;
 private double b;
 private double t;
 private double fTime;

 public String getDescription()
 {
    return description;
 }

 public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
 {
    this.a      = Double.parseDouble(parm1[0]);
    this.b      = Double.parseDouble(parm1[1]);
    this.t      = Double.parseDouble(parm1[2]);
    this.fTime  = parm5;
 }

 public double getOutput()
 {
    if (this.fTime <= this.t)
```

```
      return this.a;
    else
      return this.b;
  }
}
```

ComplianceVUVFeedback.java

```
public class BComplianceVUVFeedback extends NCVMLBlock
{

  private String description = "";
  private double Vo;
  private double Vu;
  private double Vu_old;
  private double Po;
  private double F;
  private double C;
  private double dt;

  //C=parm[0],Po=parm[1],Vu=parm1[2] for normal compliances Vo=0, C=parm1[1], F=input[0][0]
  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
   {

     if (parm4 == 1)
     {
         dt = 0;
         Po = Double.parseDouble(parm1[1]);
     }
     else
     {
        dt  = parm5/(parm4-1);
        Po  = parm3[0];
     }

     C      = Double.parseDouble(parm1[0]);
     F      = parm2[0][0];
     Vu     = parm2[1][0];
     Vu_old = parm2[1][1];
     Vo     = Vu_old + Po*C;
  }

  public String getDescription()
  {
     return description;
  }

  //V=Vo + F*dt, P=(V-Vu)/C, this P will update the PSOURCE P parameter
  public double getOutput()
  {
     double V = Vo + F*dt;

     return (V-Vu)/C;
  }
```

BUAfferentPathway.java

```
public class BUAfferentPathway extends NCVMLBlock
{
 private String description = "";

 private double tavp ;//= (double)2.076;    // s
 private double tavz ;//= (double)6.37;     // s
 private double Pn   ;//= (double)92;       // mmHg
 private double fmin ;//= (double)2.52;     // spikes/s
 private double fmax ;//= (double)47.78;    // spikes/s
 private double ka   ;//= (double)11.758;   // mmHg

 private double Pt;
 private double Past;
 private double fcs;
 private static double Pt_1;
 private static double Past_1;

 private double  dt;
 private int    nStep;

 public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
 {
    if (parm4 == 1)
    {
      this.dt   = 0;
      this.Pt_1 = 0;
    }
    else
    {
      this.dt = parm5/(parm4-1);
    }

    tavp  = Double.parseDouble(parm1[0]);
    tavz  = Double.parseDouble(parm1[1]);
    Pn    = Double.parseDouble(parm1[2]);
    fmin  = Double.parseDouble(parm1[3]);
    fmax  = Double.parseDouble(parm1[4]);
    ka    = Double.parseDouble(parm1[5]);

    Past       = parm2[0][0];
    Past_1     = parm2[0][1];
    this.nStep = parm4;
 }

 public String getDescription()
 {
    return description;
 }

 public double getOutput()
 {
     //Afferent pathway - Calculate Dynamic Carotid Sinus Pressure
     if (this.nStep == 1)
```

```
        this.Pt  = 0;
      else
        this.Pt  = this.Pt_1 + dt/tavp* (Past + tavz*(Past-Past_1)/dt - Pt_1);

      //frequency of spikes in the afferent fibers
      this.fcs     = (fmin + fmax*(Math.exp((Pt-Pn)/ka)))/(1 + Math.exp((Pt-Pn)/ka));

      this.Pt_1       = this.Pt;

      return fcs;
  }

}
```

BUEfferentSympathetic.java

```
public class BUEfferentSympathetic extends NCVMLBlock
{
  public String description = "";

  private double fesinf ;//= (double)2.10;   // spikes/s
  private double fes0   ;//= (double)16.11;  // spikes/s
  private double kes    ;//= (double)0.0675; // s
  private double fcs;

  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
  {
     fesinf = Double.parseDouble(parm1[0]);
     fes0   = Double.parseDouble(parm1[1]);
     kes    = Double.parseDouble(parm1[2]);

     this.fcs = parm2[0][0];
  }

  public String getDescription()
  {
     return description;
  }

  public double getOutput()
  {
     //efferent sympathetic pathway
      double fes = (fesinf + (fes0 - fesinf)*(Math.exp(-kes*this.fcs)));

      return fes;
  }

}
```

BUEfferentVagalPath.java

```
public class BUEfferentVagalPath extends NCVMLBlock
{

  private String description = "";

  private double fev0   ;//= (double)3.2;    // spikes/s
  private double fevinf ;//= (double)6.3;    // spikes/s
  private double fcs0   ;//= (double)25;     // spikes/s
  private double kev    ;//= (double)7.06;   // spikes/s
  private double fcs;

  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
  {
      fev0   = Double.parseDouble(parm1[0]);
      fevinf = Double.parseDouble(parm1[1]);
      fcs0   = Double.parseDouble(parm1[2]);
      kev    = Double.parseDouble(parm1[3]);

      this.fcs = parm2[0][0];
  }

  public String getDescription()
  {
      return description;
  }

  public double getOutput()
  {
     //efferent vagal pathway
     double fev = (fev0 + fevinf*(Math.exp((fcs - fcs0)/kev)))/(1 + Math.exp((fcs - fcs0)/kev));
     return fev;
  }

}
```

BUHeartPeriodRegulation.java

```
public class BUHeartPeriodRegulation extends NCVMLBlock
{
  private String description = "";

  private double  Gts;
  private double  fes_min;
  private double  fes_t_D;
  private double  sigma_ts;
  private double  tav_ts;
  private double  delta_ts_t;
  private static double  delta_ts_1;

  private double  dt;

  private double  Gtv;
  private double  fev_t_D;
```

```
private double  sigma_tv;
private double  tav_tv;
private static double delta_tv_1;
private double  delta_tv_t;

private double  T_0;


public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
{
    if (parm4 == 1)
    {
      this.dt        = 0;
      this.delta_ts_1 = 0;
      this.delta_tv_1 = 0;
    }
    else
    {
      this.dt = parm5/(parm4-1);
    }

    Gts     = Double.parseDouble(parm1[0]);
    fes_min  = Double.parseDouble(parm1[1]);
    tav_ts   = Double.parseDouble(parm1[2]);

    Gtv     = Double.parseDouble(parm1[3]);
    tav_tv  = Double.parseDouble(parm1[4]);

    T_0     = Double.parseDouble(parm1[5]);

    fes_t_D = parm2[0][0];
    fev_t_D = parm2[1][0];
}

public String getDescription()
{
   return description;
}

public double getOutput()
{
    if (fes_t_D >= fes_min)
       sigma_ts = Gts*Math.log((fes_t_D - fes_min + 1));
    else
       sigma_ts = 0;

    delta_ts_t = delta_ts_1 + dt*(1/tav_ts)*(-delta_ts_1 + sigma_ts);

    sigma_tv = Gtv*fev_t_D;

    delta_tv_t = delta_tv_1 + dt*(1/tav_tv)*(-delta_tv_1 + sigma_tv);

    delta_ts_1 = delta_ts_t;
    delta_tv_1 = delta_tv_t;

    double T = delta_ts_t + delta_tv_t + T_0;

    return T;
```

```
      //return 0.833;

  }

}
```

BUActivationFunction.java

```
public class BUActivationFunction extends NCVMLBlock
{
  private String destription = "";
  private double  Tsys;
  private double  Tsys_0;
  private double  ksys;
  private double  T;
  private static double  u_t;
  private double  fi_t;
  private double  dt;

  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
  {
     if (parm4 == 1)
     {
        u_t = 0;
        dt  = 0;
     }
     else
     {
        dt        = parm5/(parm4-1);
     }

     ksys   = Double.parseDouble(parm1[0]);
     Tsys_0 = Double.parseDouble(parm1[1]);

     T      = parm2[0][0];
  }
  public String getDescription()
  {
     return description;
  }

  public double getOutput()
  {
     Tsys = Tsys_0 - ksys*(1/T);

     u_t += dt*1/T;
     if (u_t >= 1)
       u_t = 0;

     if ((u_t >= 0) && (u_t <= Tsys/T))
     {
       //  fi_t = (1 - Math.cos(2*(Math.PI*T*u_t/Tsys)))/2; //cos2x = 1-sin^2x
        fi_t = Math.sin((Math.PI*T*u_t/Tsys))*Math.sin((Math.PI*T*u_t/Tsys));
     }
     else
     {
```

```
        fi_t = 0;
    }

    return fi_t;

  }

}
```

BUPMax.java

```
public class BUPMax extends NCVMLBlock
{

  private String description = "";
  private double  Vuv;
  private double  Pov;
  private double  kEv;

  private double  fi;
  private double  Emax;
  private double  Vv;

  public void setParam(String[] parm1, double[][] parm2, double[] parm3, int parm4, double parm5)
  {
      Vuv  = Double.parseDouble(parm1[0]);
      Pov  = Double.parseDouble(parm1[1]);
      kEv  = Double.parseDouble(parm1[2]);

      Emax = Double.parseDouble(parm1[3]);

      fi   = parm2[0][0];
      //Emax = parm2[1][0];
      //Vv   = parm2[2][0];
      Vv   = parm2[1][0];
  }

  public String getDescription()
  {
      return description;
  }

  public double getOutput()
  {
      double Pmax;

      Pmax = fi*Emax*(Vv-Vuv) + (1-fi)*Pov*(Math.exp(kEv*Vv)-1);

      return Pmax;
  }

}
```