DynSysMod: A Framework for Modeling Composite Dynamic Systems

Jennifer Leaf

TCSS 702: Design Project Final Report
August 17, 2007

Committee Chair: George Mobus, Ph.D.
Committee Member: Steve Hanks, Ph.D.
Committee Member: Rogene Eichler West, Ph.D.

Abstract

A system is a collection of elements that interact to function as a single, larger entity. The field of computer simulation and modeling concerns the creation of abstract representations of real systems and the execution of the system models in a virtual environment, to explain or predict the behavior of the real systems. System dynamics is the field of study that models systems whose behavior is heavily influenced by feedback (that is, the outputs of the system are recycled back into the system as inputs) and examines their behaviors over time. While many system dynamics modeling applications have been developed, they are lacking the ability to define multiple, independent types of flows through the system, and the ability to run different sections of the model at varying time resolutions. The application, DynSysMod, described in this project report, provides a description language and simulation execution engine to exercise system dynamics models of varying complexity. This tool will be especially useful in modeling energy systems and other types of systems that require multiple types of flows, and where different parts of the model should be executed at different rates. The framework developed in this project provides a solid foundation for system dynamics modeling and future expansion of modeling functionality.

Table of Contents

INTRODUCTION **5**

WHAT IS MODELING AND SIMULATION? 5
WHAT IS SYSTEM DYNAMICS? 6
MOTIVATION 7
RELATED WORK 7
INTELLECTUAL MERIT 8

THE SOLUTION: DYNSYSMOD **8**

CONCEPTUAL OVERVIEW 8
ARCHITECTURAL OVERVIEW 11
DETAILED COMPONENT DESCRIPTION 12
General Implementation *12*
Model Definitions *12*
Simulation Engine *13*
Mathematics of first-order difference equations 13
System Dynamics Algorithms 15
Equation Interpreter *16*
Compiler Theory 16

VALIDATION AND VERIFICATION **20**

DEFINITIONS 20
Model Validation 21
Simulation System Verification 22
ENERGY SYSTEM MODEL 23

FUTURE WORK **25**

MULTIPLE TIME STEPS 25
GRAPHICAL MODEL EDITOR 26
MODEL VALIDATION 26
DATA ANALYSIS TOOLS 26
LESSONS LEARNED 26

ACKNOWLEDGEMENTS **27**

REFERENCES **27**

APPENDIX A. TECHNICAL SPECIFICATION **30**

TERMINOLOGY 30
DYNSYSMOD COMPONENTS 30
Time Clock *30*
Future Capability 30
Graphical Representation 31
Model Representation 31
Model *31*
Graphical Representation 31
Model Representation 31
Source *31*
Future Capability 32
Graphical Representation 32
Model Representation 32
System *33*
Future Capability 33
Graphical Representation 33
Model Representation 33
Flow *33*

Graphical Representation 34
Model Representation 34
ReceiverList *34*
Graphical Representation 34
Model Representation 34
Reservoir *34*
Graphical Representation 35
Model Representation 35
Process *35*
Future Capability 35
Graphical Representation 35
Model Representation 35
Outflow *36*
Future Capability 36
Graphical Representation 36
Model Representation 36
Sink *36*
Graphical Representation 36
Model Representation 37
QUESTIONS 37
Answered Questions *38*
EQUATIONS 38
Description *38*
Keywords *39*
Requirements *39*
Examples *39*
Future Capability *39*
Notes on Units 39
APPENDIX B. MODEL DEFINITION XSD 41
APPENDIX C. MODEL DEFINITION XSD – GRAPHICAL VIEW 46
APPENDIX D. DYNSYSMOD ENERGY MODEL 47
APPENDIX E. EQUATION GRAMMAR 50
APPENDIX F. UML DIAGRAMS OF INTERPRETER ALGORITHMS 53
APPENDIX G. UML DIAGRAMS OF SIMULATION INITIALIZATION ALGORITHMS 56
APPENDIX H. UML DIAGRAMS OF SIMULATION ALGORITHMS (ONE TIME STEP) 59
APPENDIX I. UML CLASS DIAGRAMS 61
APPENDIX J. BUILDING THE SOURCE CODE AND RUNNING THE APPLICATION 63
APPENDIX K. SOURCE CODE LISTINGS 67

List of Figures

Figure 1. Graphical representation of the model elements in DynSysMod. 9
Figure 2. DynSysMod components. 11
Figure 3. A simple example of Euler's method. 13
Figure 4. The choice of dt can mask oscillations in simulated values. 14
Figure 5. UML activity diagram of simulation algorithm. 15
Figure 6. Compiler Stages. 16
Figure 7. The parse tree for "sun.rawEnergy * 0.08" 20
Figure 8. Graphical representation of energy conversion system. 23
Figure 9. Graphical representation of energy conversion system, using DynSysMod notation. 24

Figure C-1. A graphical representation of the model XSD. 46
Figure F-1. UML Activity diagram of interpreter initialization. 53
Figure F-2. UML Activity diagram of main interpreter Evaluate method. 53
Figure F-3. UML Activity diagram of main arithmetic rule evaluation. 54
Figure F-4. UML Activity diagram of “unary” parser rule evaluation. 54
Figure F-5. UML Activity diagram of “factor” parser rule evaluation. 55
Figure F-6. UML Activity diagram of Compute method in the Interpreter class. 55
Figure G-1. UML Activity diagram of main simulation initialization algorithm. 56
Figure G-2. UML Activity diagram of ProcessElement class initialization. 57
Figure G-3. UML Activity diagram of System class initialization. 58
Figure H-1. UML Activity diagram of main simulation update algorithm (one time step). 59
Figure H-2. UML Activity diagram of updating the Reservoirs for Model elements. 60
Figure H-3. UML Activity diagram of updating the Outflows for Model elements. 60
Figure I-1. Simplified UML Class diagram of Model element classes. 61
Figure I-2. UML Class diagram of other classes used in DynSysMod. 62
Figure J-1. The New Project screen in NetBeans v5.5. 64
Figure J-2. The New Java Project with Existing Sources screen in NetBeans v5.5. 65
Figure J-3. The main DynSysMod application window. 66
Figure J-4. The DynSysMod message window to notify the user that the simulation run is complete. 66

Introduction

A system is a collection of elements that interact to function as a single, larger entity. We can attempt to understand a real system by observing it and analyzing the behavior of its elements. As the system grows more complex, however, more interrelated variables come into play, and it becomes more difficult to understand how a small change to one variable can cause a considerable change in the system behavior as a whole. Additionally, in a real system, it is not always possible to gain an understanding of the system's potential future behavior by actually changing the state or inputs to the system – either the ability to change the desired variable does not exist, or the expense would be too great. Thus, the field of simulation and modeling was created to provide a foundation for creating abstract representations of real systems (models) and executing the representations of those systems in a virtual environment.

What is Modeling and Simulation?

Modeling refers to the activity of representing the essential details of a system in an abstract form [1]. Models have two components – structure, which defines the entities in the system, and behavior, which describes what the entities do and how they interact. Determining the essential structures and behaviors is, to some degree, subjective. After a model is constructed, it undergoes a validation process (described in the Validation and Verification section) to determine whether it sufficiently represents the real system.

A simulation of a model exercises the behaviors defined in the model and calculates one or more future states of the system. (Typically simulations are used to explore behavior over some period of time.) The behaviors are defined as a series of mathematical functions (either equations or state transitions, depending on the type of model), and thus can be calculated given the initial state of the model. With the exponential growth in computing power over the past few decades, and due to the sheer mathematical complexity that define many models, the vast majority of simulation execution is performed on a computer.

Simulations can be classified into two broad categories: continuous and discrete [2]. In continuous simulations, the current state of the system is assumed to flow smoothly from one state to the next and is recalculated at some fixed interval using equations, often one or more differential equations. Typically the flow of information or quantities through the system is what is emphasized, rather than the state of any particular entity at a given moment in time. Physical systems, such as electronic circuits or fluid dynamics, are typical examples of systems that are modeled using continuous simulation techniques.

In discrete simulations, the individual entities are the focus, and entities are described as being in particular states. Events which change the state of the model may happen at any time, and may not occur at a regular interval. Discrete models are used to represent systems that change at distinct moments, such as I/O queues, the use of gates at an airport, and the efficiency of service at a store or restaurant.

What is System Dynamics?

Jay Forrester, in 1958, first proposed the field of system dynamics. [3] It was originally based on the management sciences fields of decision theory and static control systems (systems that sense their environment, compare the active state to the desired state, and use the environmental data plus built-in rules to determine future behavior) to model and explain socioeconomic system management techniques, and thus it was known as industrial dynamics. It has since been applied to study problems in fields such as socioeconomic theory [4], environmental studies [5], sustainable development [6], and the evolution of cities [7], to mention just a few.

There is also a discipline within engineering fields known as system dynamics [8] (also known as dynamic systems), which has considerable overlap with Forrester's definition. Both terms are used to describe the modeling of systems that change over time. Dynamic behavior and feedback loops are central features of both engineering systems and socioeconomic systems.

However, simulating dynamic systems as the term is used in engineering is typically done to either replicate the behavior of a real system, or verify that a system follows the laws of (for example) physics or chemistry. Forrester's contribution was to apply the modeling and simulation techniques from the engineering world to analyzing complex non-linear systems. The focus of simulation changed from verifying or replicating known behavior, to exploring projected scenarios and predicting outcomes to show how variables can potentially change over time. Often these systems are influenced by human decision making and action upon the system under study, and are used to create and test policy decisions. [9] The emphasis on the roles of feedback, time progression, and nonlinear systems is what distinguishes system dynamics from other uses of simulation, since a small change to the system may be amplified over time into a much larger impact with results that are often not obvious or intuitive from simple examination of the model. The precise numerical results of the simulation are often not the primary focus of evaluating a model, but rather demonstrating the general behavioral patterns over time.

Note that while these systems are generally modeled using relatively large time steps (i.e., the data has a meaningful change once a month or once a year), these models are generally classified as continuous due to the smooth advancement of time and the use of numeric values to represent the current state of the system, rather than a set of distinct, well-defined states as one would find in a state diagram. [1]

Stocks and flows are the main components of dynamic systems [5]. A stock represents storage of some type of information or entity (such as money or population) in the system. A flow defines the rate of change to a stock – adding more of the type of information or entity to a stock, removing some to either move elsewhere in the system, or expelling it from the system as waste. Other inputs to the system that are not necessarily part of the system model itself include converters and sources.

Stocks and flows show the structure of the model; however as mentioned earlier it is equally important to demonstrate the interactions between the entities in the model. In

dynamic systems, this is primarily achieved using causal loops [10]. Causal loop diagrams show the cause and effect relationships in the system. The stocks and converters are drawn with arrows drawn from the cause to point at the effect. The arrows are labeled with whether the effect's value changes in the same direction as the cause's value (i.e., either both increase or both decrease), or whether the values change in opposite directions. These two relationships are knows as positive feedback and negative feedback, respectively. Systems that tend toward a stable state contain negative feedback loops, while systems with positive feedback loops can easily destabilize given even a small change in the current state. The converters are never an effect in the system, they are only causes.

Motivation

While many system dynamics modeling and simulation applications have been developed, they are lacking a few features that would allow a broader range of systems to be modeled effectively. Two aspects that are not currently available are the ability to define multiple, independent types of flows through the system, and the ability to run different sections of the model at varying time resolutions. Some subsystems within the model need to be recalculated quickly in order to provide an accurate representation of the behavior, while other subsystems can be processed much more infrequently and still provide reasonable results.

Related Work

CSIM is a system modeling toolkit used to create discrete event based models [11]. It is composed of a set of software libraries and components to assist C++ and Java developers in writing their own simulation applications. The provided components handle the low level simulation operations such as representing queues, processes, storage, and events. The interesting feature of CSIM is its ability to allow a knowledgeable developer to write simulation components directly in code, which provides almost unlimited extensibility of the simulation. In the simulation engine that this proposal describes, subject matter experts, with the assistance of software developers, should be able to extend the engine to support a variety of mathematical functions and capabilities, above and beyond what the core engine will initially support.

DYNAMO [12] was one of the first continuous systems modeling languages, developed in the late 1950s and early 1960s at M.I.T. It was targeted at subject matter experts rather than software developers, thus its design goals were correctness and ease of use, rather than programmable flexibility. It was based on using difference equations [13], which are often used to approximate linear differential equations. A difference equation uses functions that consider the previous value of a variable when computing the successive value of the variable. It also provided a simple graphical output to show the relative values of variables, and was eventually extended to plot smooth graphs using plotter devices.

STELLA is one of the most widely used dynamic systems modeling tools currently on the market [14], and is considered the current successor to DYNAMO. It is a graphical-based tool that allows the user to define models using commonly accepted dynamic

systems symbols, and provide the equations used to describe the behavior. It is focused on modeling simple to moderately complex systems that are defined by ordinary differential equations. The units of the quantity being measured by the flows must be consistent throughout the diagram, which can be a barrier to effectively diagramming relatively complex models. It also (as of version 8.1) is not designed to handle systems defined using partial differential equations, which involve functions with several independent variables [15].

XMILE (XML-based System Dynamics Model Interchange Language Entity) [16] is an XML-based model exchange language that was proposed to permit easier translation of models between different system dynamics simulation tools. Having a common exchange language would enable model builders to collaborate more freely since each modeler could use the modeling application they were most familiar with, and it would provide greater opportunities for model review and reuse. The specific XML structures in the XMILE specification are not sufficient to meet the unique modeling requirements for DynSysMod, but if XMILE were to be adopted by the system dynamics community, it would be straightforward to translate DynSysMod models to XMILE using XSLT (Extensible Stylesheet Language Transformations) stylesheets. [17]

Intellectual Merit

Rather than narrowly focusing on a specific problem, this project integrated topics from several areas of computer science and applied them to a single computer application. The foundation of DynSysMod is in the modeling and simulation field. The equation interpreter component required learning about compiler theory and fundamental computing theory such as grammars and regular expressions.

The solution: DynSysMod

Conceptual Overview

DynSysMod is a modeling language and simulation engine for dynamic systems modeling. DynSysMod is particularly well suited to modeling energy systems that need to consider energy and matter flows separately. While it is possible to model these types of systems using existing applications such as STELLA, the modeler must treat flows as combined energy and matter quantities, which makes it difficult to analyze the behavior of the model. The following figure contains a graphical representation of the core elements that make up a DynSysMod model. The purpose and current behavior of each element is described below.

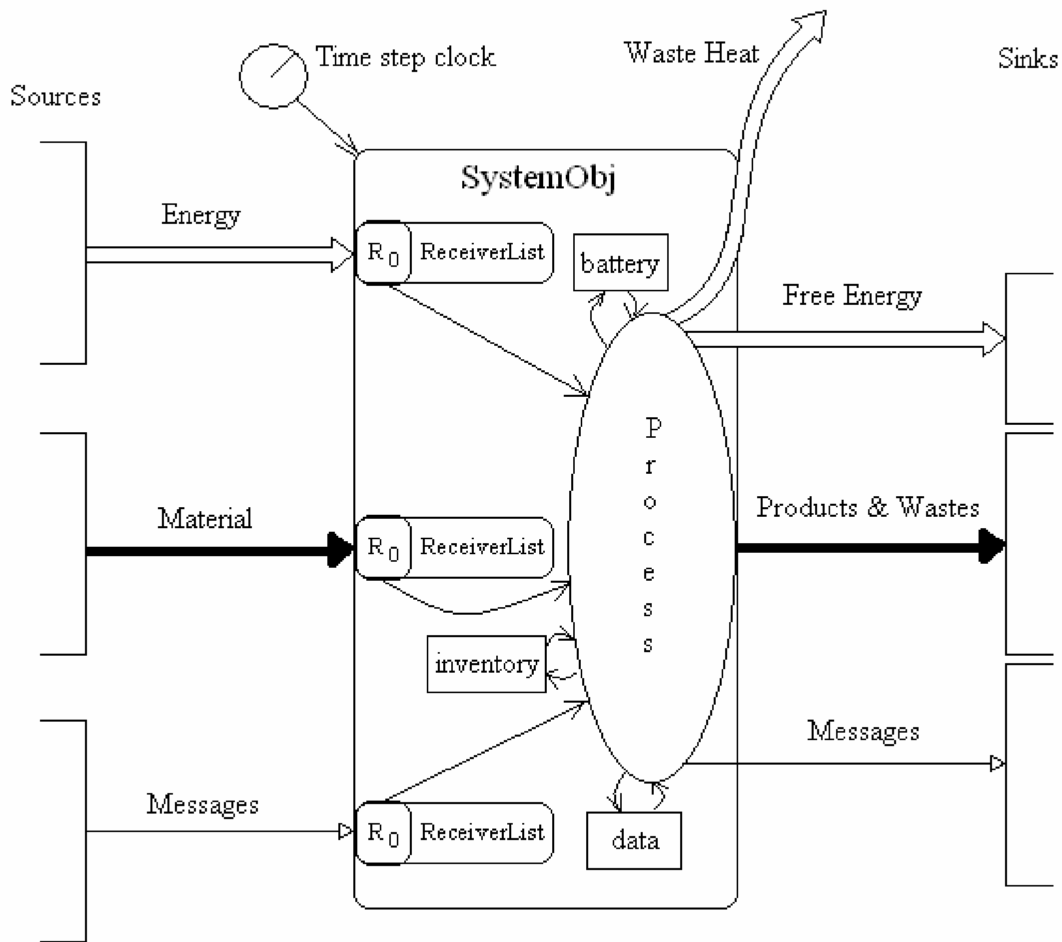


Figure 1. Graphical representation of the model elements in DynSysMod. Created by Dr. George Mobus.

Model

The Model represents the outermost boundary of the entities being modeled and simulated. Models are fairly simple objects, since the interesting behavior is defined within the individual elements.

Sources

Sources represent exogenous inputs to the model. The values provided by a Source are unaffected by the processes defined in the model. Sources are used either when testing the execution of the model against a known set of data, or when the derivation of the data is irrelevant to the model under study. DynSysMod allows Source data to be provided either with a comma-separated values (CSV) file [18], or derived using a mathematical equation to compute the value of the Source at each time step. The Sources are connected to Systems by Outflows, which do the actual work of updating the values at each time step.

System

The central component in a DynSysMod model is a System. Systems can either stand alone as a model, or can be decomposed into subsystems. Subsystems can only be

composed under a single parent system – the same System cannot be added to multiple parent systems. Thus Systems can be modeled in a tree-like data structure, as each node will have a single parent, but can have multiple children (subsystems). Systems contain 3 types of elements that all combine to define the System's behavior: Receivers, Reservoirs, and Processes.

ReceiverList
A ReceiverList is a conceptual collection of inputs to a System or Sink. Each ReceiverList accepts the inputs to the System of a single type – if there are multiple types of Outflows that feed into a single System, there will be one ReceiverList per type. In the current implementation of DynSysMod, these elements have no behavior since the units of the equations are not checked for consistency.

Reservoir
Reservoirs serve as intermediate buffers for the flows in the model. They provide a "snapshot" of important values that change in the model over time. The equations defined in the Process update their values. Reservoirs correspond to "stocks" or "levels" in system dynamics terminology.

Process
The Process component holds the equations that define the behavior of the system. These equations use the values from the Outflows in the ReceiverLists and the System's Reservoirs to calculate the values of the Outflows and update the values of the Reservoirs.

Sink
A Sink is a repository for a flow that has left the scope of all the Systems. The Sink can also be used to siphon off un-reusable "waste" quantities that cannot serve as Sources to other Systems. In the current implementation, DynSysMod treats Sinks as infinite capacity Reservoirs. At each time step, Sinks accumulate the values from the Outflows that flow into the Sink, record the value, and then discard it.

Outflow
System dynamics models are conceptually continuous (even though they are often defined using difference equations), thus values can be thought of as "flowing" from Sources, between Systems, and into Sinks. The Outflows define the relationships and interactions between the Model elements. Outflows that originate from Systems can be updated using the equations defined in the Process component. Outflows that originate from Sources can be updated using whatever data sources are allowed for Sources. Outflows correspond to "rates" or "flows" in standard system dynamics terminology.

Time Clock
The Model as a whole, and each System within it, can have an independent time clock – how frequently the functions that make up the Process should be recomputed. The Model contains the top-level time clock for the model as a whole. Individual Systems can override this clock as appropriate. This allows the modeler to define the time resolution

to be as coarse as possible to maintain performance, yet fine-grained enough to accurately simulate the behavior of the modeled system. It may be the case that one System may require a very small time step, while another System can retain accuracy with a much larger interval between calculations.

Architectural overview

During the course of this project, an overall design and a prototype of the core components were developed and documented. Below is a diagram of the high-level architecture for the system. The shaded components were the focus of this project.

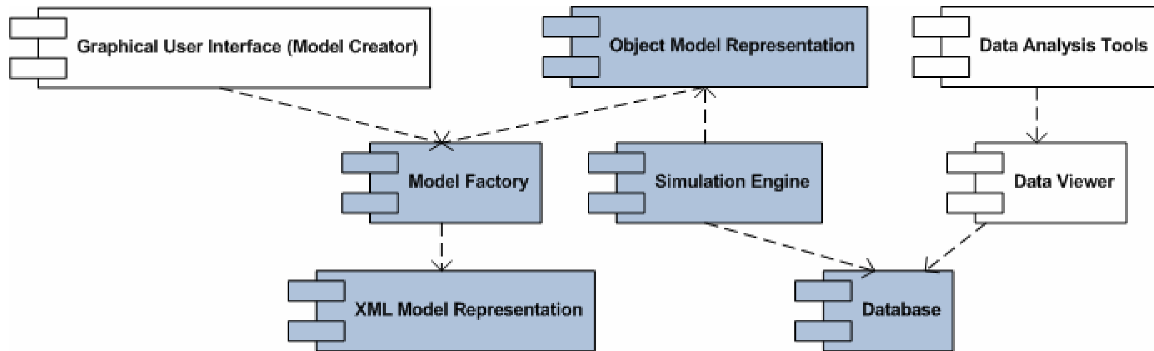


Figure 2. DynSysMod components.

Each vertical column of components in the diagram represents a subsystem within DynSysMod. Moving left to right on the diagram, the components in the first vertical column are part of the user interface subsystem. The second column, which intentionally overlaps the first and third columns, is the model representation subsystem. The third column is the simulation engine subsystem. The fourth subsystem simply contains the data storage for the simulation results. Finally, the rightmost column contains the output data analysis components, which comprise the fifth subsystem.

User Interface Subsystem

The user interface subsystem contains the graphical model editor. Eventually DynSysMod will have a graphical editor for models that will allow users who are experts in the fields that system dynamics is applied to, rather than experts in writing XML files, to “draw” models and fill in the model parameters (see Future Work section below).

Model Representation Subsystem

Models are represented as human-readable XML files. XML [19] is a widely used method of data definition and exchange. It provides a language and platform independent means of representing and persisting information. Users will be able to write the XML files directly using a text editor, or will construct the models using the graphical model editor described above once it has been developed. The models constructed using the graphical interface will then be converted into the XML format. This layer also includes the model factory component, which translates between the XML, graphical, and object representations of models.

Simulation Engine Subsystem
The simulation engine components execute the behaviors defined in the model, and generate the output data that is used in the analysis stage. The simulation engine consists of the main engine that tracks the simulation clock and drives the algorithm forward, and schedulers that determine when to update particular Systems based on their individual Time Clock settings.

Data Storage Subsystem
As the simulation runs, the data storage subsystem stores each Model element's value after each time step.

Data Analysis Subsystem
The data analysis components are used to review the results of the simulation and draw conclusions about the model.

Detailed Component Description

General Implementation

DynSysMod is written in the Java programming language. The core data structures and algorithms in DynSysMod are statically compiled into machine-independent bytecode, which in turn is interpreted during runtime by a Java Virtual Machine (JVM). [20] This allows the program to run without recompilation on any computer system that has the Java Virtual Machine software installed on it.

DynSysMod uses several 3rd party libraries to accomplish common tasks. The log4j library [21] is used for diagnostics logging and runtime tracing of the simulation process. JUnit [22] provides a framework for writing and executing unit tests, which allow the automation of executing test code to exercise a method or other small part of code. ANTLR (ANother Tool for Language Recognition) [23] is a parser generator, used to automatically build a parser for a custom language definition. The opencsv library [24] is used to parse CSV (comma-separated values) files.

The technical specification, documented in Appendix A, describes many of the design decisions and implementation details involved in creating the current prototype of DynSysMod. Some of the key features and algorithms are outlined immediately below.

Model Definitions

The central aspect of DynSysMod is the model definition. While several simulation languages have been developed to represent dynamic systems (such as STELLA and DYNAMO), there are a few features that are unique to DynSysMod that require a custom modeling language. The parent system-subsystem relationships between systems, the potential for independent time clocks for each system, and the representation of distinct types of receivers and reservoirs for varying types of flows through the systems are supported in the model definition.

The format of the XML files is defined using an XML Schema Definition (XSD) [25] document, which can be found in Appendix B. The XSD makes extensive use of complexType definitions in order to promote reuse of definitions throughout the file. For example, equation-based Sources, Reservoirs, and Outflows all require the same information to define the equations, so a separate type was created and reused throughout the XSD file.

Simulation Engine

Mathematics of first-order difference equations

Recall that a model defines one or more entities and the behaviors of each of those entities. In a numerical simulation, we can represent the current state of an entity by one or more numeric values, and the behavior by one or more equations. These behavior equations define the rate of change in the state as a function of some period of time. In order to find the state of the entity after some interval of time (called dt, for delta or change in time), we can take the current state and use the current rate of change to determine the next value. This method of computing successive values is known as Euler's method or Euler integration. It assumes that dt is the same between all calculations. This assumption does hold for all models supported by the current implementation of DynSysMod.

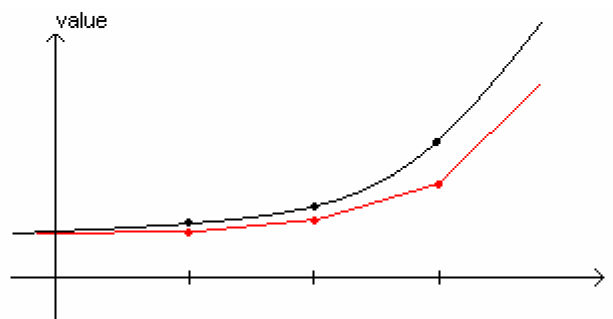


Figure 3. A simple example of Euler's method. The darker line represents the real-world values, the lighter colored line represents what Euler's method would approximate for the value and rate at each time step. Adapted from Figure 22.3.1 in [26].

Many system dynamics simulations use Euler's method to compute successive states for the model. The most common application of system dynamics is to understand the general patterns of behavior and interaction between several entities, not necessarily to calculated detailed future values of individual entities. Frequently, the parameters that control the behavior of the model are estimated, rather than measured, and so may differ from the real value by a relatively large factor. Thus the integration method need be no more accurate than the accuracy of the model parameters. [27] From a computational standpoint, Euler's method is the simplest for a software developer to implement, for a system modeler to understand, and for a computer to execute. So as long as the modeler chooses dt well, Euler's method is often sufficient for simulating systems dynamics models.

Since the rate of change is recalibrated after each dt, the rate will more closely reflect the behavior of the entities being simulated as dt becomes smaller. If we wait a relatively long time before recording a new state and recalibrating the rate of change, the greater likelihood that the rate we were using is not reflective of the behavior of the underlying system. Therefore it is important to choose a reasonably small dt.

Qualitatively, dt should be short enough to ensure the changes in the model Reservoirs between time steps are small, in order to not cause a discontinuity in the solution. For example, if the value of a Reservoir becomes negative in a single time step, then dt is too large. Choosing a dt that is too large can also mask oscillations or significant changes in the value, as is demonstrated in the following figure.

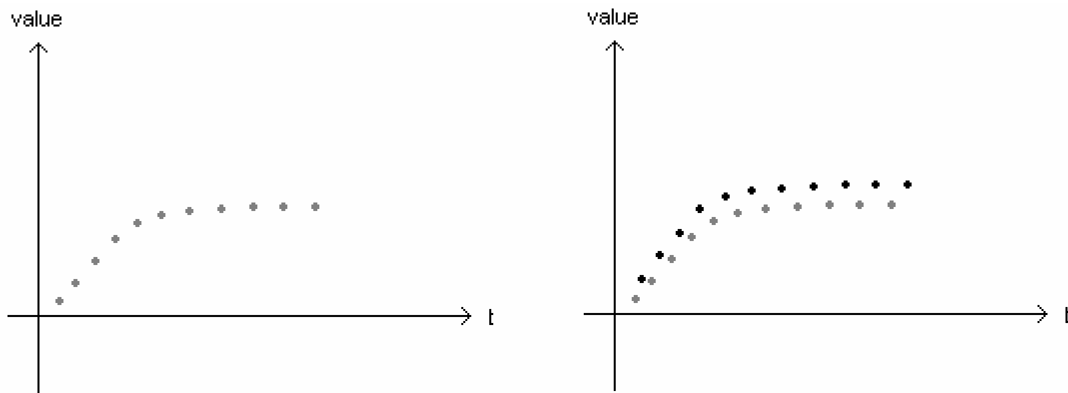


Figure 4. The choice of dt can mask oscillations in simulated values. The diagram on the left shows what appears to be regular logarithmic growth. However, by halving dt to measure the value twice as often, an underlying oscillation is detected (the darker data points) that is masked in the original results. Adapted from Fig. 2 and Fig. 3 in [27].

Determining an acceptable value for dt requires a blend of experimental and computational techniques. Examining the output of a simulation run for obvious problems or unexpected behavior such as the draining of a Reservoir is a simple indicator that dt is too large. Another common technique is to run the simulation with a particular dt, and run it again with a dt that is half or one quarter of the original dt. If the difference between the results is less than the predetermined tolerable error, then the larger dt can be used. [27, 28] There are other integration techniques, such as the commonly used Runge-Kutta methods [26], that compute the rate of change at several points between two time steps, and use a weighted average of the rates. While these algorithms generally provide more accurate results, they are more computationally expensive than the Euler method. As discussed earlier, the increased accuracy may be unnecessary due to the imprecision of the model parameters.

System Dynamics Algorithms

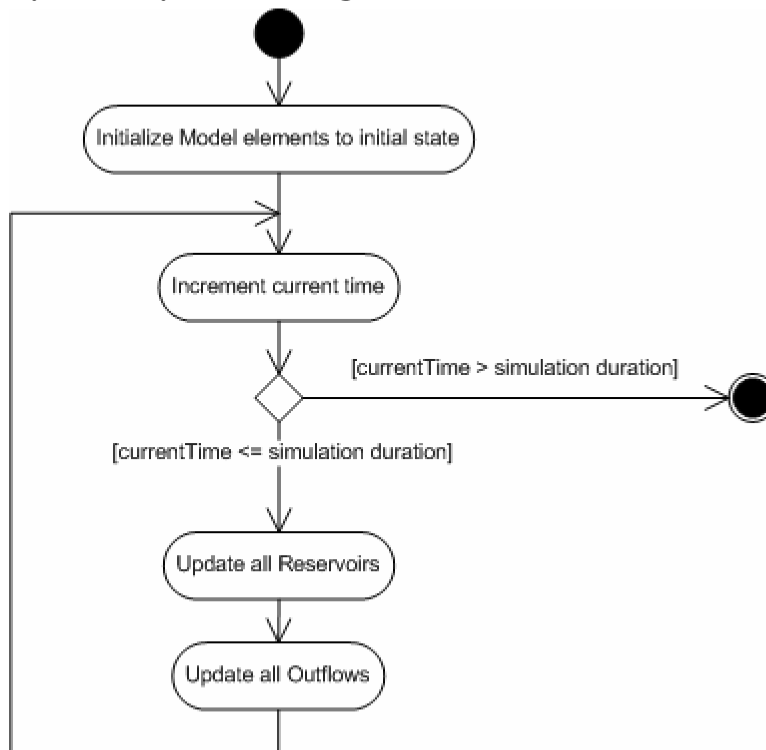


Figure 5. UML activity diagram of simulation algorithm.

For this first version of DynSysMod, the simulation algorithm used is straightforward. [4, 9] It assumes all model elements are recalculated after each time interval. First the model elements are initialized, since Euler's method requires at least one point and its rate of change to be defined. As the initialization process serves to "compute" the values of each model element at the initial time step, the clock is then moved forward by a single time step. Since we know the value of each element at the previous time step and its rate of change, all the reservoirs are updated first, and then the flows are recalculated. The clock is moved forward once again, and the cycle repeats until the simulation clock has surpassed the user-supplied duration.

Within a single time step, it does not matter which order the reservoirs or flows are calculated in (as long as all reservoirs are recalculated before the flows). The new value of reservoirs depends only on the values of reservoirs at the previous time step, and the values of flows across the previous time interval. The new value of flows depends only on the values of reservoirs at the current time step, and the values of flows across the previous time interval. Therefore there are no dependencies between the values of two reservoirs or two flows at the same time step.

The Scheduler objects in DynSysMod provide the implementations of the simulation algorithm. The LinearScheduler class implements the algorithm described above. As DynSysMod is enhanced to permit multiple time clocks, additional Schedulers can be written to encapsulate the scheduling algorithms so that only the appropriate Systems are

updated after each time interval. This permits the classes that represent the model elements, such as the System, Source, and Outflow classes to be algorithm-independent, limiting the number and scope of future design and code changes.

Equation Interpreter

We need to allow the modeler considerable flexibility in constructing the equations that define the behavior of each element in the model. Since the equations themselves are not known when the DynSysMod application is compiled, DynSysMod includes an interpreter that can examine the text of the functions supplied in the model definition and compute the value on demand.

Compiler Theory

Compiler theory is the area of computer science that deals with how to convert computer programs written in some language into a different, yet equivalent form. [29] A language is defined simply as a set of strings. Two main types of applications fall into this category. A compiler translates a program written in one language, called the source language, into an equivalent program in another language, known as the target language. Often the target language is machine code that can be executed directly by a computer processor. Interpreters read the source program and execute the statements immediately, rather than creating another program to be executed later. Applications such as translators and code generators also apply the concepts in compiler theory. Regardless of the application, all of these activities are commonly grouped under the term “compiler”.

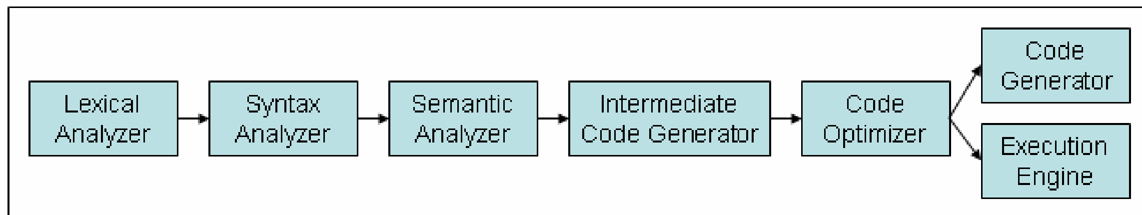


Figure 6. Compiler Stages. Adapted from Figure 1.6 in [29].

Figure 6 shows the main stages of the compilation and interpretation processes. The components up to (and including) the intermediate code generator are commonly known as the compiler front-end, while the remaining steps are called the compiler back-end. The only divergence between compilers and interpreters is at the end of the process, where a compiler uses a code generator to create the target program, and an interpreter uses an execution engine to execute the statements.

Due to the relative simplicity of the grammar, the equation interpreter in DynSysMod implements a 3-stage process – the lexical analyzer, the syntax analyzer, and a combined semantic analyzer/execution engine. These components are described later in this section. An intermediate code generation step is often used to decouple the reading and parsing of the source program from the generation of the target program. By decoupling these two activities, it is relatively straightforward to write a new back-end to provide support for a new operating system or hardware platform using a pre-existing programming language. Using an intermediate code generator is an unnecessary step for

DynSysMod, since it is designed to run solely on a standard JVM. Code optimizers analyze the output from the front-end and determine where there are inefficient, redundant, or unused steps and replace them with streamlined operations. For example, if a variable is assigned, but not read until it is assigned a different value at a later time, then there is no need to store the intermediate value. Since this prototype of DynSysMod focuses on functionality, rather than maximal performance, there are no optimization steps in the DynSysMod equation interpreter.

It is a relatively straightforward task to create a lexical analyzer (also referred to as a lexer) and syntax analyzer (also referred to as a parser) from a grammar definition. Rather than manually coding these two components for an individual grammar, tools called parser-generators can analyze a grammar file and automatically output code that will perform these two stages. DynSysMod uses one such tool called ANTLR to create a lexer and parser for the equations.

ANTLR creates language recognizers – the code that it generates from grammar definitions is able to determine whether a given string is valid for that language. The code generated by ANTLR for the DynSysMod equation interpreter accepts a single equation as a Java String object as input, and creates a parse tree as its output. The Interpreter class in DynSysMod then performs a preorder traversal of the parse tree and performs a specific action based on the name of the node as each node is processed. The specific algorithms used by the interpreter are defined using UML activity diagrams and can be found in Appendix F.

The lexical analyzer converts the source program (for DynSysMod, a mathematical function) into a series of tokens that the syntax analyzer later consumes. Tokens serve as the building blocks of a language. The lexical analyzer groups the characters in the source program into objects called lexemes, and then assigns each lexeme to a particular type of token. In order to know the type of token, the lexeme is matched against a pattern, generally a regular expression. By defining the set of lexemes for a token using a pattern, later stages of the compilation process can treat all instances of a token type identically if desired.

In the DynSysMod grammar, the lexical rules are divided between keywords and operators, which are defined using literal strings, and more complex structures such as variable names, whose definitions are built up using a series of regular expressions. Below is an example of a token type, using the ANTLR format for defining lexer rules:

```
IDENTIFIER : NAME (NAME_DELIMITER NAME)*;
```

The name of the token type is `IDENTIFIER`. This particular token is used to define variable or function names in DynSysMod equations. `IDENTIFIER` tokens are strings that start with a `NAME` token (which is defined in its own rule), followed by 0 or more sets of a delimiter token (which is a period) and another `NAME`. Examples of lexemes that match this pattern, and thus would be assigned to an `IDENTIFIER` token type, are “atmosphere” (an example of a lexeme that matches only the required `NAME` pattern) and

"sun.rawEnergy" (an example of a lexeme that matches the optional `(NAME_DELIMITER NAME)` pattern).

The ordering and grouping of the tokens in the source program is equally as important as simply identifying and classifying the tokens. A grammar specifies the ordering and grouping of tokens through a series of substitution rules. It serves to identify all the strings that make up a particular language. A syntax analyzer examines a sequence of tokens (called a string) from the lexical analyzer and determines whether the grammar is able to generate that string using the grammar's rules.

The syntax analyzer does not need to differentiate between different lexemes – all lexemes of a particular type can be processed generically. This is why the lexical analyzer assigns each lexeme to a token. For example, the syntax analyzer does not determine whether a particular variable name is legal, it only verifies that an identifier is permitted at a certain location within the string. It is up to the semantic analyzer to examine the token's lexeme and determine whether it is valid.

Context-free grammars (CFG) are a common type of grammar used to define a language. Formally, a CFG consists of 4 elements:

- A set of terminal symbols. For a compiler, the names of the tokens are the terminal symbols.
- A set of non-terminal symbols. These are used in the production rules to build up the valid token sequences via substitution.
- A set of production rules. A production rule consists of a non-terminal symbol called the head or left-hand side, and a string of terminal and non-terminal symbols.
- The start symbol, which is always a non-terminal symbol. This symbol is used to identify the first rule that should always be applied when verifying a string.

Production rules are also known as substitution rules or derivations. The string of symbols on the right side of the rule shows the valid substitutions for the non-terminal on the left-hand side of the rule. Starting with the rule that has the start symbol on the left-hand side, the parser attempts to find a sequence of substitutions that matches the string of tokens by applying the production rules recursively.

DynSysMod uses the ANTLR grammar notation to define the syntax of valid equations. The grammar definition can be found in Appendix E. The grammar file only lists the production rules and start symbol explicitly – the set of terminal and non-terminal symbols are inferred from the parser and lexer rule definitions.

Some examples of rules in the DynSysMod equation grammar are:

```
startRule : expr;
expr   : term ( ( PLUS | MINUS )  term )* ;
term   : unary ( ( MULT | DIV ) unary )* ;
unary  : '-' unary | factor;
factor : '(' expr ')' | number | t | dt | identifier;
```

```
identifier :      IDENTIFIER;
number     :      NUMBER;
```

There is only one possible substitution for the non-terminal symbols `startRule`, `expr`, and `term`. The `unary` symbol has two possible substitutions, separated by an or ('`|`') symbol. ANTLR also permits the use of regular expression notation (such as grouping multiple symbols using parentheses or quantification operators such as +, *, or ? to indicate the number of times a symbol or group of symbols can occur) in the parser rules to simplify rule construction. The symbols in capital letters (for example `PLUS` and `MINUS`) are tokens defined elsewhere in the grammar file.

As a simple example, we will show the derivation for the equation "sun.rawEnergy * 0.08". The symbol ⇒ means "derives in a single step" and is used to show the progression of substitutions from the start rule to the string of terminals or tokens from the source program.

```
startRule ⇒ expr ⇒ term MULT term ⇒ unary MULT term ⇒ factor
      MULT term ⇒ identifier MULT term ⇒ IDENTIFIER MULT term ⇒
      IDENTIFIER MULT unary ⇒ IDENTIFIER MULT factor ⇒
      IDENTIFIER MULT number ⇒ IDENTIFIER MULT NUMBER
```

The derivation does not translate the `IDENTIFIER` token into the literal "sun.rawEnergy", nor the `NUMBER` token into the literal 0.08.

The output from the syntax analyzer is a parse tree that represents the production rules followed to generate the string of tokens. The interior nodes show the left-hand side of the rules that were applied during the derivation process, and the leaf nodes contain the lexeme for each token. Below is the parse tree for the equation "sun.rawEnergy * 0.08".

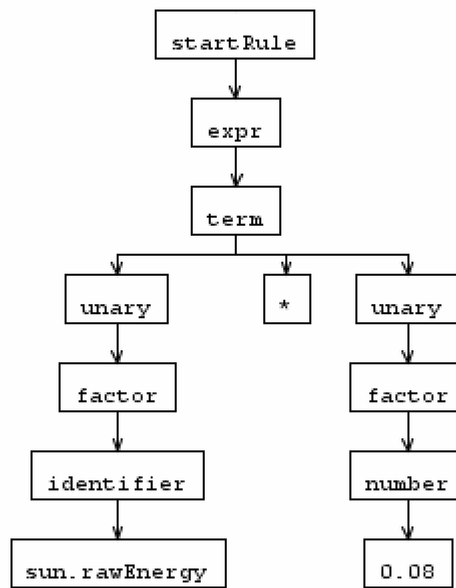


Figure 7. The parse tree for "sun.rawEnergy * 0.08". Created using ANTLRWorks. [30]

Grammars are often grouped into classes, based on the direction that the input is scanned, which non-terminal symbol is substituted during a derivation step, and how many tokens ahead of the current position must be examined to decide which rule to apply. ANTLR is able to generate parsers for what are known as LL(k) grammars. [31, 32] In other words, the parsers that ANTLR generates scan their input from left to right (the first L in LL(k)), produce a leftmost derivation (the leftmost non-terminal in the string is always substituted for in the next derivation step), and look ahead at the next k tokens from the lexer.

The parsers that process LL grammars are known as top-down parsers. [29] These parsers build the parse tree starting at the root (or start symbol), and build each successive interior node until a token is found, at which point a leaf node is constructed. Recursive-descent parsers implement top-down parsing by using a set of recursive method calls, where each method corresponds to a single rule. Each method matches tokens, recursively calls another rule-based method, or both.

Validation and Verification

Definitions

The terms validation and verification have specific definitions in the simulation and modeling domain. Validation is the procedure used to determine whether the model represents the real world with sufficient accuracy, and verification refers to the process of ensuring the simulation behaves in accordance with the model's defined behavior [33]. Another way to describe these terms is that validation tests the definition of the model itself independently of any implementation or simulation of it, while verification tests the correctness of the simulation engine algorithms that execute the model.

Model Validation

In this project, we will assume that the models used to verify DynSysMod's functionality have been externally validated, and thus the evaluation of DynSysMod falls under the category of verification. However, a few words should be said about model validation performed after the simulation has been executed.

Models can demonstrate varying degrees of strength of validity [1]. A model is *replicatively valid* if the model can produce the same results that the real system has already produced – i.e. the model is true after the fact. It is *predictively valid* if it can correctly predict future behavior in the real system for an event that has not yet occurred. Finally, if the model successfully reproduces or predicts the behavior of the real system, and also succeeds in reflecting the internal interactions of the real system that cause the behavior, the model is *structurally valid*. This distinction is important to make, since it may be the case where the model, given the same inputs as the real system, generates the same outputs, yet uses a drastically different means of producing them. Thus either the real system has multiple explanations, or there is no real cause-and-effect in the behaviors being modeled. A classic example of this is the use of sunspot activity to predict changes in economic cycles [2]. Even if a correlation between sunspot activity and stock market behavior was found, it does not necessarily prove that they are intertwined – they may only be accidentally related. Such models do not provide any real insights into the real system. Much of the model validation process is designed to determine whether the model's results are purely coincidental, or truly do isolate and replicate the interesting behaviors of the system.

When possible, the results of the simulation should be compared against actual data collected while the real-world process that was modeled actually occurs. There are a variety of statistical tests that can be used to determine whether the distributions of the model's results are the same as the real process's results [34] The specific techniques used to perform this analysis depend on the nature of the model.

In the absence of real data to use as a comparison, for example when the model is used to predict future events with no previous occurrences, then the primary methods of validation are subject matter expert analysis and sensitivity analysis [33]. The sensitivity analysis technique takes a single input variable to the model, holds all other inputs constant, and then examines the output results as the input variable changes. The ratio of the changes in the input variable to the ratio of the changes in the output variable is known as the sensitivity. If the ratio is relatively constant over a range of inputs, then the relationship is linear and thus the model's behavior will not vary wildly given a small change in the initial state. Models that exhibit low sensitivity are generally considered to be more accurate [2]. Models may be highly sensitive either due to the inherent variability of the underlying system (for example, due to a positive or self-reinforcing feedback loop), or may illuminate a subsystem that should be modeled in more detail to accurately represent real-world behavior.

Simulation System Verification

Verification of the simulation system is composed of two phases. First, standard software testing techniques during product development such as individual component testing using unit tests to ensure individual functions and classes execute correctly, integration testing to ensure the components work properly together, and system testing to determine whether the system meets the initial requirements, are used to verify correctness of the software components. However, it is also necessary, once the system is fully implemented, to execute models with known results (and as many sets of results as possible) to ensure the simulation system can replicate the results.

In its final state, DynSysMod will be able to model common dynamic systems models, as well as more complex models with varying time representations and multiple flow types. However, the scope of this prototype did not permit us to create a realistic model of a real system. The current prototype of DynSysMod integrated the main components and algorithms, but does not support enough mathematical functions to model anything but the most trivial system. Unit tests for the interpreter and some of the model element classes provided much of the testing of individual components. The simulation algorithms were designed using UML diagrams and reviewed by Dr. Mobus. After the code was implemented, it was carefully compared to the design documentation. Several simple test models were created and the results were computed for several time steps using a calculator, which were then compared to simulation runs from DynSysMod. This testing demonstrated that the simulation and interpreter algorithms were correctly implemented

After more planned features of DynSysMod are implemented (such as supporting separate time steps for individual System objects), there are two types of sample models that are required to fully verify its behavior. First, DynSysMod should be able to handle any system dynamics model that an existing tool such as STELLA can handle. A particular model would be constructed and executed using STELLA. Then, the same model would be constructed in DynSysMod's XML language and executed using the DynSysMod application. The results of the two executions should produce similar results that can be judged equivalent.

Once we have established that the baseline functionality works, we can then move onto a model that STELLA cannot simulate. In order to demonstrate the capabilities of multiple flows and multiple time resolutions of subsystems, an additional model must be developed. The model will have at least 3 levels of subsystem composition, in other words, the main model will have at least one System object that can be decomposed into a subsystem, and one of the subsystems can be decomposed into yet another subsystem. Each of these three subsystems will run at a different time resolution to demonstrate the ability of DynSysMod to handle multiple time resolutions. This model will also demonstrate multiple types of flows through the system that are tracked and computed individually. Since no other simulation engine exists that is capable of executing such a model, the results of simulating the model will have to be checked by hand to verify their accuracy.

Energy System model

The genesis of the DynSysMod project was the attempt to build a dynamic model of an sustainable energy system. While it is possible to construct such models using existing tools, a modeling language specifically designed to model such systems provides equivalent expressive power in a more compact yet less complex representation.

The model used to compare the expressiveness of existing tools vs. DynSysMod is a simple solar energy conversion model. Figure 8 shows the model as it would be drawn using the STELLA tool, while Figure 9 shows the same model using the DynSysMod graphical notation. Refer to Appendix D for the XML version of the DynSysMod model.

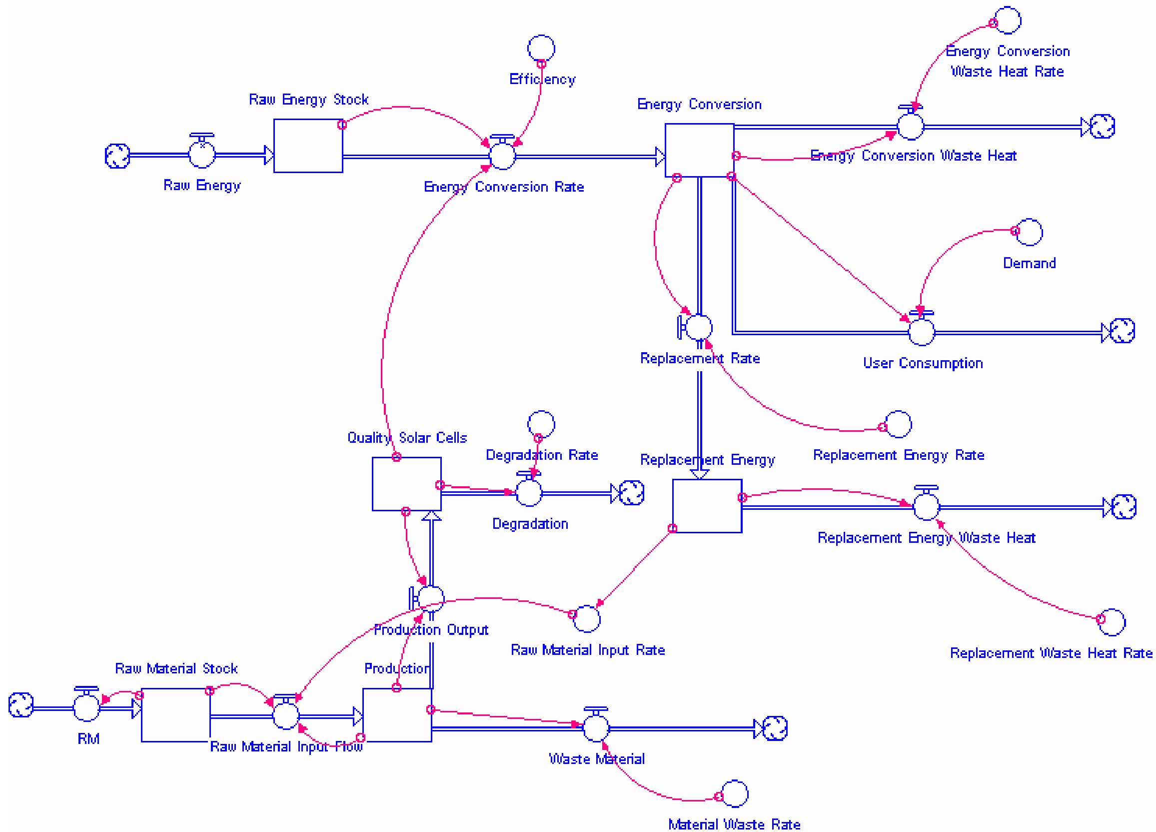


Figure 8. Graphical representation of energy conversion system. Created using the STELLA system dynamics modeling application.

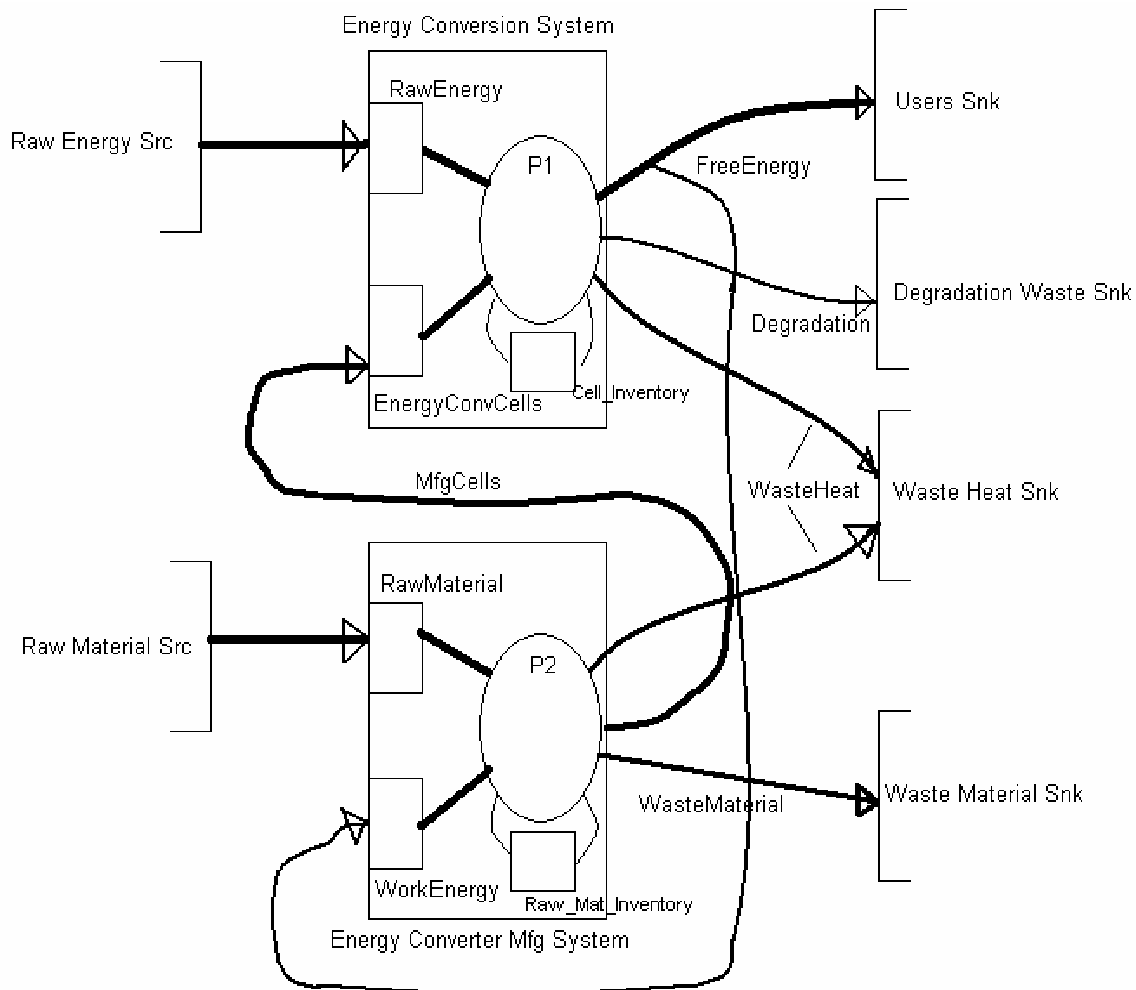


Figure 9. Graphical representation of energy conversion system, using DynSysMod notation. Created by Dr. George Mobus.

This model focuses on the conversion of solar energy to usable electricity and the uses of the generated electricity. The percentage of raw solar energy that can be converted to electricity is controlled by the overall efficiency of the solar cells, as well as the quality of the cells. The quality degrades over time, limiting the amount of energy that can be converted per unit time. A production process is in place to replace low quality solar cells, and it is assumed that adequate raw materials are available to support the production process. However the production process consumes a percentage of the electricity generated by the conversion process, and generates waste material and heat.

This model can be used to explore the sustainability of the energy conversion process. Is there enough electricity generated to support the replacement solar cell production cycle as well as providing sufficient free energy to meet consumer demand? If the solar cells are not able to generate enough electricity to support a sufficiently high production rate, the quality of the solar cells will degrade over time since the degradation process will outstrip the ability to produce replacement cells quickly enough. Sufficient free energy

must also be produced to meet customer demand – there is no point in building the solar energy conversion plant simply to provide for its own upkeep.

We built and executed each model and analyzed the results. In both models, the maximum energy conversion rate was set to 8% with perfect quality solar cells, and the conversion rate decreased as the solar cells degraded. As a result, the amount of energy going to the cell replacement energy stock is very small. Since the production process is not receiving sufficient energy to build replacement solar cells, the quality of the solar cells degrades faster than they can be replaced. The system goes into a slow decline over the simulation run, and eventually the whole thing will grind to a halt since there are no functional solar cells remaining.

However, we were unable to draw direct numerical comparisons between the two models. The DynSysMod version reaches its steady decline after 2-3 time steps. The STELLA version took close to 50 time steps to show a similar trend. The two models used different equations and different constants, to account for the different structures. In order to draw better comparisons, a deeper understanding of both representations of the model must be achieved in order to ensure that both models are truly representing the same information.

Even though we were unable to perform the quantitative model comparisons that we planned to do, the prototype has shown where additional work is needed, both in extending DynSysMod's functionality, and in understanding the energy system model itself.

Future work

The prototype created during this project demonstrates the feasibility of designing a simulation application that can support the needs of system dynamics modelers, and provide the ability to model separate types of flows independently. While a solid framework is in place, there are several additional features that can be added to make DynSysMod a more robust and useful tool, many of which would constitute separate projects in their own right. Many potential enhancements to features implemented in the prototype are noted in the Technical Specification in Appendix A. The following sections describe new functionality.

Multiple time steps

One of the key proposed features of DynSysMod is the ability to specify different time clocks for the different systems in a model. Due to time constraints this feature is not considered in the design or implementation for the current prototype. The two affected algorithms are analyzing the multiple time clocks in the model definition to determine the appropriate simulation clock interval, and implementing another Scheduler class to perform the needed updates at the appropriate times based on each System's clock.

It may be the case that a series of Scheduler classes, each encapsulating a particular algorithm of determining the correct time to update each model element, should be written. In this case the preferred solution is to create a Scheduler Factory using the

Factory design pattern. [35] The Scheduler Factory would analyze the time clocks in the model and determine the appropriate Scheduler class to use.

Graphical model editor

While XML provides sufficient expressive power to construct models with as much detail and sophistication as desired, it is not necessarily the easiest format to read or write directly. The graphical model editor will allow subject matter experts to create models by dragging and dropping objects onto a diagram editor, and allowing the user to connect the objects using flows. Analyzing the human-computer interface factors and determining the most effective means of providing graphical editing tools would be a project in its own right.

Model Validation

To assist the user in creating valid models, DynSysMod should provide validation tools to both prevent the user from creating an invalid construct whenever possible, and notifying the user after model creation of any errors in the final structure.

Most, if not all, of the preventative validation steps would be part of the graphical model editor. Examples of preventative validation steps include disallowing an Outflow from a Sink object, or attempting to connect a Source directly to a Reservoir (it should connect to the System that contains the Reservoir).

Examples of validation that can only occur after the user believes the model is fully defined include ensuring that all Reservoirs and Outflows have an equation to define their behavior, checking that units are used consistently throughout the equations (e.g., not allowing the user to add a variable that represents a volume to a variable that represents a length of time), and in the case of matter and energy flows, verifying that the laws of thermodynamics are observed.

Data analysis tools

From an end user standpoint, creating and simulating the model, while not trivial, is just one step in the analysis of a problem. Sophisticated tools that allow the user to study the data generated by the simulation are important in providing a robust toolset to the user. The envisioned design for the data analysis components is a plug-in architecture that would provide straightforward tools such as charts and graphs, but also permit the development of customized representations such as an animated a 3-D representation of the data to “watch” the system in action.

Lessons Learned

One of the more valuable contributions that this project provided is that we don’t quite know enough about how to model the energy systems in question to effectively define the requirements for the simulation application. The prototype as it is now will be a good launching point to continue exploring these ideas, and it can be expanded in the future to help refine the requirements for this energy systems modeling tool.

From a long term standpoint, once the requirements are better scoped, it would be worth reassessing whether to adapt existing tools, frameworks, and libraries, or to continue to write a new simulation system from scratch. Sometimes the act of building a prototype shows where the development process can benefit more from reusing or licensing 3rd party products. In particular, the raw mathematical computation is probably best served by reusing another library since that problem has been solved over and over again, and it is no trivial thing to implement. However, developing the equation interpreter prototype was a good learning experience, and without having prototyped that component directly, it is difficult to determine what functionality and features are required from a 3rd party library. Developing a prototype is often a quick way to determine whether to write the code from scratch, or reuse another implementation.

The end state of DynSysMod quite possibly will be a hybrid between 3rd party libraries that do the math and the unit conversions, and new development to build up the specific energy system modeling needs such as the graphical editor and XML representations and overlaying well-known simulation algorithms with the multiple time clock management. These last two areas are where the current tools really don't meet the needs of energy system modeling, whereas the number crunching is more or less a solved problem. There is much more work that can be done to expand the systems modeling field.

Acknowledgements

The DynSysMod project was originally proposed and defined by Dr. George Mobus of the University of Washington, Tacoma. Through his research in energy systems, he identified the shortcomings of current modeling tools and created the basic simulation approach described in this paper.

References

[1] B. P. Zeigler, *Theory of Modelling and Simulation*. New York: John Wiley & Sons, 1976.
[2] R. E. Shannon, *Systems Simulation: the Art and Science*. Englewood Cliffs, N. J.: Prentice-Hall, 1975.
[3] J. W. Forrester, “Industrial Dynamics,” *Harvard Business Review*, vol. 36, pp. 37-66, 1958.
[4] J. W. Forrester, *Industrial Dynamics*. Cambridge, MA: The M.I.T. Press, 1961.
[5] A. Ford, *Modeling the Environment: An Introduction to System Dynamics Modeling of Environmental Systems*. Washington, DC: Island Press, 1999.
[6] D. Meadows, J. Randers, and D. Meadows, *Limits to Growth: The 30-year Update*. White River Junction, VT: Chelsea Green Publishing Company, 2004.
[7] J. W. Forrester, *Urban Dynamics*. Cambridge, MA: The M.I.T. Press, 1969.
[8] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg, *System Dynamics*, 3rd ed. New York: John Wiley & Sons, Inc., 2000.
[9] R. G. Coyle, *System Dynamics Modeling: A Practical Approach*. London, UK: Chapman & Hall, 1996.
[10] N. Roberts et. al., *Introduction to Computer Simulation: A System Dynamics Modeling Approach*. Reading, MA: Addison-Wesley, 1983.

[11] H. Schwetman, "CSIM19: a powerful tool for building system models," in *Proc. 2001 Winter Simulation Conf*. Arlington, Virginia: IEEE Computer Society, 2001.

[12] R. E. Nance, "A history of discrete event simulation programming languages," in *The second ACM SIGPLAN conf. History programming languages*. Cambridge, Massachusetts, United States: ACM Press, 1993.

[13] "Difference Equation," Encyclopædia Britannica, 2006, in Britannica Concise Encyclopedia; http://concise.britannica.com/ebc/article-9362727/differenceequation. Access date: Nov. 20 2006.

[14] isee systems, *STELLA v9.0.1*; http://www.iseesystems.com/softwares/Education/StellaSoftware.aspx Access date: Nov. 20 2006.

[15] N. Chonacky, "Stella: growing upward, downward, and outward [software review]," *Computing in Science & Engineering*, vol. 6, pp. 8-15, 2004.

[16] V. G. Diker and R. B. Allen, "XMILE: towards an XML interchange language for system dynamics models," *System Dynamics Review*, vol. 21, pp. 351-9, 2005.

[17] W3C, "XSL Transformations (XSLT)," Nov. 1999; http://www.w3.org/TR/xslt. Access Date: Aug. 8 2007.

[18] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, IETF RFC 4180, October 2005; http://tools.ietf.org/html/rfc4180.

[19] W3C, "Extensible Markup Language (XML)," Sept. 2006; http://www.w3.org/TR/xml/. Access Date: Aug. 15 2007.

[20] J. Gosling and H. McGilton, "The Java Language Environment," Sun Microsystems, 1996; http://java.sun.com/docs/white/langenv/. Access Date: Aug. 8 2007.

[21] Apache Software Foundation, *log4j v1.2.14*; http://logging.apache.org/log4j. Access Date: Aug. 4 2007.

[22] E. Gamma and K. Beck, *JUnit v4.3.1;* http://www.junit.org. Access Date: Aug. 4 2007.

[23] T. Parr, *ANTLR v3.0*; http://www.antlr.org. Access Date: Aug. 4 2007.

[24] G. Smith, *opencsv v1.7*; http://opencsv.sourceforge.net/. Access Date: Aug. 4 2007.

[25] W3C, "XML Schema Part 1: Structures," Oct. 2004; http://www.w3.org/TR/xmlschema-1/. Access Date: Aug. 15 2007

[26] R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed. New York, NY: Dover, 1973.

[27] A. L. Pugh, "Integration Method: Euler or Other for System Dynamics," in *TIMS Studies in the Management Sciences*, vol. 14, J. Augusto et. al., Eds. New York, NY: North-Holland Publishing Company, 1980, pp. 179-188.

[28] P. M. Barton and A. M. Tobias, "Accurate estimation of performance measures for system dynamics models," *System Dynamics Review*, vol. 14, pp. 85-94, 1998.

[29] A. V. Aho et. al., *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA: Pearson/Addison-Wesley, 2007.

[30] J. Bovet, *ANTLRWorks v1.0.2*; http://www.antlr.org/works/. Access Date: Aug. 4 2007.

[31] T. Parr and R. Quong, "ANTLR: A Predicated-LL(k) Parser Generator," *Software - Practice and Experience*, vol. 25, pp. 789-810, 1995.

[32] T. J. Parr and R. W. Quong, "LL and LR translators need k>1 lookahead," *SIGPLAN Not.*, vol. 31, pp. 27-34, 1996.

[33] R. G. Sargent, "Verification and Validation of Simulation Models," in *Progress in Modelling and Simulation*, F. E. Cellier, Ed. New York: Academic Press, Inc., 1982, pp. 159-169.

[34] J. A. Payne, *Introduction to Simulation: Programming Techniques and Methods of Analysis*. New York: McGraw-Hill, 1982.

[35] E. Gamma et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995.

[36] *NetBeans v5.5*; http://www.netbeans.org. Access Date: Aug. 4 2007.

Appendix A. Technical Specification

Terminology

- Short name: consists of alphanumeric characters, with at least 1 alphabetic character. No other symbols or whitespace are allowed. This is the standard for defining an element in a model.
- Fully qualified name: consists of a series of System names, starting at the top-most level of composition, followed by the short name for the element. Each section of the name (each System name, and the short name) is separated by periods (.). This is the standard for referring to another element in the model. Exogenous Sources and Sinks have no System names – their fully qualified name and their short name are identical.

DynSysMod Components

This section describes the major components in DynSysMod. Each component has an overall description of its purpose, and then a description of how it is represented graphically using a graphical model editor, and how it is represented in a serialized model format. The standard serialization of a model is to an XML file. Please refer to "DynSysMod.xsd" for the details of the XML schema – this document contains only a verbose description of the important features.

Time Clock

The Time Clock defines the temporal resolution the system (or subsystem) will execute at. This permits different subsystems within a single system model to run at different rates. The performance of the model simulation can thus be optimized. Additionally, different resolutions of external data may exist for different subsystems, allowing mismatched data sets to be used within the same model (for example, one process has external data points at 1 year intervals, while another process has data points at 1 month intervals).

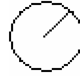
One can run separate sections of a model at different time resolutions by decomposing the model into subsystems, and specifying different Time Clock settings for each subsystem.

Future Capability

Currently time is represented by a value and a unit – i.e. 1 year, 12 hours. This simplistic representation makes it difficult to convert between different time units. Once permitting separate time clocks for each System is implemented, the model parser will need to be able to examine each time clock and determine a common interval. For example, a Model with a System that has a time clock of 1 hour and another System that has a time clock of 45 minutes (admittedly, this is a fairly contrived scenario) should use a time interval of 15 minutes for the base simulation engine clock, to guarantee both Systems will be recalculated at appropriate intervals. Since there are several implementations of

time-related objects in open source libraries, these should be explored rather than trying to roll your own conversion routines.

Graphical Representation

The Time Clock is represented by a clock face. 

Model Representation

Time has both a value (1, 25, 0.01) and a temporal unit (year, second, century).

It is assumed that all Model elements are calibrated to a single, consistent time clock. DynSysMod does not convert equations or values between different units at this time. (See Notes on Units section.)

Model

The Model represents the outermost boundary of the entities being modeled and simulated. Models consist of one or more Systems, zero or more exogenous Sources, and zero or more Sinks.

Graphical Representation

The Model is the collection of model elements as a whole and thus has no individual graphical representation.

Model Representation

Model names are short names. The name is not used to build up fully qualified names for other model elements; it is simply a descriptive string for the model.

Models are fairly simple objects, since the interesting behavior is defined within the individual elements. In addition to the list of Sources, Sinks, and Systems, the Model also contains the top-level TimeClock for the model as a whole. Individual subsystems can override this clock as desired.

Source

Sources represent exogenous inputs to the model. That is, **how** the specific values of the input are derived is not included in the model, only the raw values are used. One would use exogenous inputs either when testing the execution of the model against a known set of data, or when the derivation of the data is irrelevant to the model under study. The values provided by a Source are also unaffected by the processes defined in the model.

Standard Sources are flows of energy, matter, or data defined by the rate of inflow. Examples include solar energy, raw steel, or an order for supplies.

The values of each Source are derived in one of several ways. Sources can be defined using an equation whose value is recalculated at each time step. The values can also

come from a CSV (comma-separated values) text file or from a database source. (The database source is not currently supported by the XSD or simulation engine.)

Future Capability

If the System object under inspection is a decomposed subsystem, a Source may also represent a Sink from another subsystem, when it is composed back into a larger System. In this case, the Source is an independent data source from the perspective of the subsystem, but is not an independent data source from the perspective of the model as a whole.

A special type of Source is used to represent inputs to the system that, from a modeling standpoint, lay outside the boundary of the system being modeling, yet are regulated or controlled by a behavior inside the boundary of the system. These Sources are available so that the graphical view of the system makes conceptual sense.

One example where this type of Source might be used is in modeling a manufacturing process. The manufacturer relies on an external source of materials in order to create the inventory that the manufacturer then sells. The supplier lies outside the model, yet the rate at which the supplier provides the raw materials is determined by the manufacturer placing an order. For representational purposes, we want the user to be able to model the supply process as completely external.

Graphical Representation

Sources are represented by a rectangle, open on the left side. ( ])

Model Representation

Exogenous Source names are short names, since they are not a part of a particular system.

1. Standard Sources can be defined using a variety of input types.
    a. Equation-based Sources are defined by a Process (equation) and a unit type (quantity/time).
    b. CSV-based Sources are defined by a filename and optionally a column index. A single CSV file can contain multiple values, where each column represents a different flow. The first line in the CSV file may contain column headers, but it is not required to. If the first line in the file contains headers, then the column whose name matches the fully qualified name of this flow is used. If the first line in the file contains the first data point, then the model must specify which column index to use. If the model specifies a column index but the CSV file contains a header row, the model's column index is the setting that is used.

2. The special Sources have a special designation marking them as a "pseudo-source". These Sources are not included in the ReceiverList for the System. The actual input from the source is modeled as a feedback loop, and the rate is defined by the Process equations. (not currently supported by the XSD or simulation engine)

System

The central component in DynSysMod is a System. Systems can either stand alone as a model, or can be decomposed into subsystems.

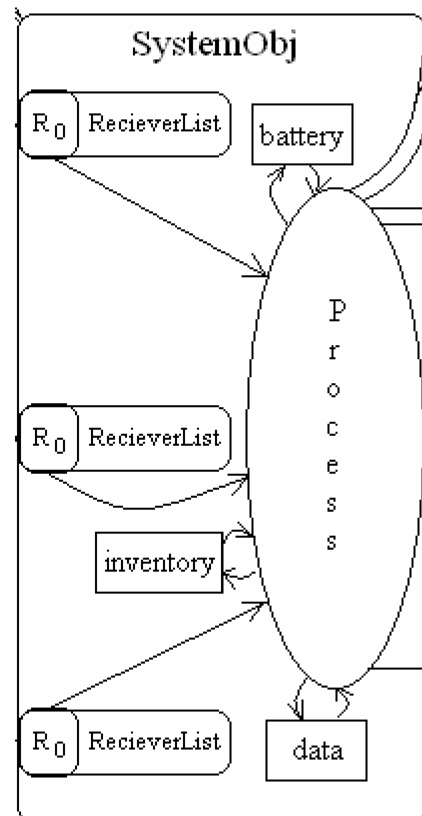
Subsystems can only be composed under a single parent system – the same subsystem cannot be added to multiple parent systems. Thus Systems can be modeled in a tree-like data structure, as each node will have a single parent, but can have multiple children (subsystems).

Future Capability

Each System is simulated at its own time rate, specified by a Time Clock. This allows the modeler to define the timer resolution to be as coarse as possible to maintain performance, yet fine-grained enough to accurately simulate the behavior of the modeled system. This also allows subsystems to be simulated at a different resolution than its parent system, so that the accuracy vs. performance tradeoff is considered in individual components throughout the model, rather than forcing some subsystems to be recomputed unnecessarily. It may be the case that one subsystem may require a very small time step, while another subsystem can retain accuracy with a much larger interval between calculations. Thus each subsystem can be executed at the rate appropriate to that subsystem.

Graphical Representation

A System is represented as a large rectangular symbol with rounded corners. All other symbols except Source, Sink, and Flow arrows are drawn within the System symbol. Systems have ReceiverList input points, and Outflow output points. All Flow arrows connect to one of these two locations.



Model Representation

System names are short names.

Systems can contain either their constituent subsystems, or if the System is at its lowest level of decomposition, it can contain the TimeClock, Sources, Reservoirs, Sinks, and Processes of the System.

If a System does not have its own TimeClock, the clock is inherited from its parent System's settings, or the Model's settings if no parent System has its own TimeClock settings.

Flow

Flows define how energy, materials, and information move through the system.

Graphical Representation

A Flow is represented by an arrow symbol. (→) The head of the arrow indicates where the flow is going to, while the tail of the arrow indicates where the flow is coming from.

The tail end of the arrow is attached to the actual source of the flow – the head end is simply a mechanism for connecting the flow to another piece of the model.

Model Representation

Flows themselves are not part of the model representation – just their endpoints.

Flows originate either from external Sources, or as Outflows from a Process. Flows attach (at the head end) either to Systems or Sinks.

ReceiverList

Flows into the system go into ReceiverLists. There is one ReceiverList per type of input. ReceiverLists are simply a way to group similar inputs, as it is likely that the Process operations will operate on all inputs of the same type similarly.

Graphical Representation

ReceiverLists are not explicitly defined in a model; their existence is derived from analyzing the types of flows into the System. Thus there is no direct graphical representation of them.

In the abstract view of the System object, ReceiverLists are represents as rectangles on the left side of a System symbol.

Model Representation

The ReceiverList is a series of flow names that represent the inflows to the System.

Reservoir

Reservoirs serve as intermediate buffers for the flows in the model. They provide a "snapshot" of important values that change in the model over time. The equations defined in the Process update these values. These roughly correspond to "stocks" in system dynamics terminology.

Reservoirs have a maximum capacity. If the capacity is overflowed during runtime, one of three behaviors can occur:

1. An error is raised to the user notifying them that their model is invalid. (default behavior)
2. The value of the Reservoir is set to the maximum capacity, and any overflow is discarded.
3. The overflow is ignored and the value of the Reservoir is set to the calculated value.

Graphical Representation

Reservoirs are represented by rectangles that have a bi-directional arrow connecting it to the Process symbol. (See the System section for an example.)

Model Representation

Reservoir names are short names. Referencing a Reservoir from another element requires using the Reservoir's fully qualified name.

Reservoirs contain an initial value, a unit specification (quantity type), the equation used to update their value, and the maximum capacity.

Reservoirs are used as input and/or output variables in Process equations.

Process

The Process component holds the equations that define the behavior of the system. These equations use the values in the ReceiverLists and Reservoirs to calculate the values of the Outflows and update the values of the Reservoirs.

Future Capability

For clarity and flexibility, allow additional variables or constants to be defined in the model. These variables aren't Reservoirs or Outflows, but represent intermediate values that could be used to simplify equations and make the model definition more readable. These include the "auxiliary" or "converter" variables in systems dynamics, but could also include supplementary equations [4].

Graphical Representation

Processes are represented as ovals in the middle of the System symbol. They are connected via arrows (although not Flow arrows) to ReceiverLists, Reservoirs, and Outflows. (See the System section for an example.)

Model Representation

All variable names used in Process equations must be fully qualified names. The variables are assumed to be Receivers or Reservoirs referenced within that System (in the Receivers or Reservoirs sections).

Processes consist of one or more equations. In general, a set of equations for a given Process should use every Source at least once. There must be exactly 1 equation per Reservoir and Outflow variable that assigns a value to that variable – i.e.

For each (Reservoir r)
    r = r(t- dt) + f(sources)
For each (Sink s)
    s = f(sources, reservoirs)

If a flow feeds back into the same system that it originated from, and its name appears in a Process equation, the name refers to the Receiver, not the Outflow. We choose the Receiver since only the value of an Outflow at the previous time step can be used to calculate the current value of a Reservoir or Outflow.

Outflow

Outflows are the output equivalent of ReceiverLists for inputs. These provide a venue for a Flow to either exit the system completely to Sinks, or for a Flow to be recycled back into a System as an input (either back into the same system as a direct feedback loop, or to serve as an inflow to another System).

Future Capability

Message (data) flows allow information to pass between elements. Use cases for messages include:

- A System or Sink notifying another System that its capacity has been reached or exceeded, triggering a behavioral change in the receiving System.
- A System monitoring a Reservoir or Flow value and notifying an exogenous Source or System to increase production, thereby increasing the inflow of materials or energy.

Implementing message flows would allow more sophisticated behavior of Sinks and Systems based on the state of the Model, rather than depending solely on writing equations.

Graphical Representation

Outflows are represented as rectangles on the right side of a System symbol. (See the System section for an example.)

Model Representation

Outflow names are short names. Referencing an Outflow from another element requires using the Outflow's fully qualified name.

Outflows are assigned a unit, much like Sources. However their quantities are derived as a function that includes one or more ReceiverList or Reservoir values.

Sink

A Sink is a repository for a quantity of a flow that has left the scope of the System. It may be an output of the model, or if the current System is a subsystem of a larger model, the Sink may be used as a Source to another subsystem. The Sink can also be used to siphon off un-reusable "waste" quantities that cannot serve as Sources to other subsystems.

Graphical Representation

Sinks are represented by a rectangle, open on the right side. ([ )

Model Representation

Sink names are short names, since they are not a part of a particular system.

Sinks are defined by a unit, but are not expressed as rates. The rates at which values flow into a sink are controlled by the Process equations. A Sink holds only the most recent value, but DynSysMod provides logging/data collection capabilities to track the values in the Sink over time.

Questions

Note – do NOT renumber these questions, they are referenced in the code.

1. Message flows are not yet considered, these may be a special case.
2. It might be possible to allow "uneven" (I have NO idea what the right word is) input data sets, since we are tracking time carefully throughout the model. By uneven, I mean data sets where the data points are unevenly spaced. There are a lot of ramifications of allowing this.
3. Clarification – a flow arrow coming out of a system that circles back into the same system does NOT indicate material leaving the system – it simply indicates that the flow depends on some value in the system. Or would that be represented by the arrow connecting the reservoir and the process? Perhaps you **would** want to show this by an external flow arrow, simply so the value would be recorded as leaving and then re-entering the system.
4. Should Sinks be initialized to 0 at time t = 0? My reasoning for not doing so is to avoid confusion as to whether the Sink was truly empty at t = 0, or simply has an undefined value.
5. During runtime, I can't see a use for modeling Sinks as entities with any useful behavior. Their values at each time step are assigned by equations in the Process block for the System that uses the sink, so as long as the Process equations of the System are evaluated at each time step, the behavior of the Sinks attached to the System are completely handled without any extra steps. Is this a correct assessment?
6. Systems with different time steps – let's say a particular System is evaluated at some interval, say 1 month. The System uses the output of another System, where this second system is evaluated at a longer interval, say 3 months. If the first System asks for the value of the second System after 1 month, what should the second System provide? The previously calculated value? 1/3 of the previous calculated value? Some approximated value? Same questions apply if one of the two elements is a Source rather than a System.
7. Source initial value is assumed to be 0. It takes at least one tick of the clock for the Source to cause an effect on the model. Is this statement true?
8. Sources – not sure how to handle calculating their values. If the value of a Source is simply a matter of a text-based equation that is a function of literal numbers, t, and dt, then I can already handle it. What other formats should I expect? Text files? Arbitrarily formatted text files, or can I specify the requisite format? Database tables? How to designate the SQL query, or which column contains the time and which column contains the value? How to specify the unit?

9. When evaluating an equation that uses the current time - what if the time doesn't exist in the hashtable? In other words, the caller asks for a value at a previous time, but that time exactly doesn't exist due to how the time clocks for each System are set up. For example, we have values for t = 3 and t = 4, but the caller asked for t = 3.5. Do we round up (to the future) to the nearest time where we do have a value? Do we throw an exception? What if the caller requests a time in the future thus there is no data for the given time?
10. Rates – are rates of Sources always a percentage, or is there some other representation of them?

Answered Questions

- It is unclear, if a single Source feeds into two different Systems, whether the flow is divided between the different Systems, or whether each System receives the same input. i.e. does the Source represent the total inflow, and a junction in the flow arrow represent a siphoning off of part of that flow like a water system, or does it represent some quantity that is reproduced/duplicated at each junction? If so, then a junction needs to be a model element in its own right with a % attribute. Same question exists for outflows. [Answer: it depends. For some types, such as energy or matter, then the laws of thermodynamics demands that, at a junction point, the total sum of the inputs is exactly equal to the total sum of the outputs from the junction point. For data flows, there is no such constraint and so the user would be able to specify whether the data is duplicated across all junction point outputs or if the data only continues along a subset of the junction point outputs.]
- When a system is simply a composition of two or more subsystems, would the simulation engine still run the subsystems as distinct entities and store the results at each time step as though it were two subsystems instead of a single larger system? [Answer: yes]

Equations

Description

In this first version of DynSysMod, equations for Process elements are simple plain-text strings. Future alternatives for exploration include representing equations as XML (MathML) or other string formats that a 3rd party mathematical expression evaluation library uses.

The basic grammar that defines legal equations is found in the grammar file Equations.g. DynSysMod uses the ANTLR Parser Generator tool (http://www.antlr.org/) to automatically generate a lexer and parser for the grammar defined in the .g file. The lexer and parser can validate many, but not all, of the possible legal equations.

For example, the grammar can validate that parentheses are balanced, that arithmetic operators have two operands, and that numbers have a single decimal point. The grammar alone cannot validate that identifiers (names of model elements) used in the equations are defined somewhere in the model definition – this must be verified at runtime.

All values are converted to Java BigDecimal (java.math.BigDecimal) objects for evaluation.

All identifiers and keywords are case sensitive.

Keywords

Keywords have a special meaning in the DynSysMod equation language and cannot be used as identifiers.

- t
- dt

Requirements

- +
- -
- *
- /
- *id*[*x*] - Get value of any model element *id* at any previous time step *x*
- *id* – Get the value of any model element *id* at the last calculated time step
- *id* = <*equation*> - Sets the value of the model element *id* at the current time step to the value calculated by evaluating <*equation*>
- ( ) for overriding precedence rules when evaluating an expression
- t = current time
- dt = time step for that model element
- literal double values
    - integers
    - doubles
    - negative values
    - exponentials

Examples

Assume all identifiers are defined in the model.

```
materials = materials[t - dt] + (restock – distribution - waste) * dt
waste = 0.01 * materials
restock = 50
distribution = materials[t – 2 * dt] * 0.5
```

Future Capability

- Additional mathematical operators.
- The ability to call static Java methods, such as those in the java.lang.Math class.

Notes on Units

Every Model element (Source, Reservoir, Flow, etc.) has a Unit associated with it. One of the features of DynSysMod is that it will enforce consistency of units in the equations.

Thus you won't be able to add a length and a weight together – the result would be meaningless.

However, the subject matter experts that create the models and specify the equations may not be mathematical experts, so DynSysMod will be able to step in and perform conversions between units when needed. Thus one equation may express the value of a particular model element in terms of cubic feet per second, while another equation may express the value in terms of cubic meters per minute. These equations both represent volume per time, thus the result of one equation can be used as a term in the other equation.

Building a robust conversion library is time consuming, and it may be possible to reuse open source libraries to do some or most of the work.

At this time, DynSysMod assumes that the model writer has considered the units of the equations and has written the equations accordingly – DynSysMod neither converts the units nor validates the consistency of the units between terms in an equation.

Appendix B. Model Definition XSD

```xml
<?xml version="1.0" encoding="utf-8"?>
<!-- edited with XMLSpy v2005 rel. 3 U (http://www.altova.com) by Jennifer Leaf (UW
Tacoma) -->
<xs:schema xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata" id="NewDataSet">
	<xs:element name="Model">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="TimeClock" type="TimeClockType"/>
				<xs:sequence maxOccurs="unbounded">
					<xs:annotation>
						<xs:documentation>Sources may be true
exogenous variables that do not flow from another system, or may be inputs from another
system.  Exogenous sources are also added as elements to the Model element by name.
Names of Sources must be fully qualified (i.e. System1.System2.flowName) in order to
disambiguate flows - two Systems having outflows with the same basic
name.</xs:documentation>
					</xs:annotation>
					<xs:choice>
						<xs:element name="EquationSource"
type="EquationSourceType"/>
						<xs:element name="CsvFileSource"
type="CsvFileSourceType"/>
					</xs:choice>
				</xs:sequence>
				<xs:element name="System" type="SystemType"
maxOccurs="unbounded"/>
				<xs:element name="Sink" maxOccurs="unbounded">
					<xs:complexType>
						<xs:sequence>
							<xs:element name="InitialValue"
type="xs:double" minOccurs="0"/>
							<xs:element name="Receiver"
type="ReceiverType" maxOccurs="unbounded"/>
							<xs:element name="Unit"
type="xs:string" msdata:Ordinal="3"/>
						</xs:sequence>
						<xs:attribute name="name"
type="xs:string"/>
					</xs:complexType>
				</xs:element>
			</xs:sequence>
			<xs:attribute name="name" type="xs:string"/>
		</xs:complexType>
```

```
	</xs:element>
	<xs:complexType name="EquationSourceType">
		<xs:sequence>
			<xs:element name="Outflow" type="EquationOutflowType"
maxOccurs="unbounded"/>
		</xs:sequence>
		<xs:attribute name="name" type="xs:string" use="required"/>
	</xs:complexType>
	<xs:complexType name="EquationType">
		<xs:sequence>
			<xs:element name="InitialValue" type="xs:double" default="0"
minOccurs="0"/>
			<xs:element name="Equation" type="xs:string"/>
			<xs:element name="Unit" type="xs:string"/>
		</xs:sequence>
	</xs:complexType>
	<xs:complexType name="CsvFileSourceType">
		<xs:sequence>
			<xs:element name="Outflow" maxOccurs="unbounded">
				<xs:complexType>
					<xs:sequence>
						<xs:element name="Filename"
type="xs:string">
							<xs:annotation>
								<xs:documentation>The file
name can be either a relative path, or the full filename.</xs:documentation>
							</xs:annotation>
						</xs:element>
						<xs:element name="ColumnNumber"
type="xs:int" default="-1" minOccurs="0">
							<xs:annotation>
								<xs:documentation>We can
handle CSV files with or without an initial header line.
this element is not present, DynSysMod assumes there is a header row and will figure out
which column of data to use by performing a string match on sourceName.flowName
from the header row.
this element is present, DynSysMod assumes there is no header row and the first row is
assumed to be the data at t = 0.</xs:documentation>
							</xs:annotation>
						</xs:element>
					</xs:sequence>
					<xs:attribute name="name" type="xs:string"
use="required"/>
					<xs:attribute name="partof" type="xs:string"
use="optional">
						<xs:annotation>
```

```
						<xs:documentation>If the partof
attribute is not present, then this Outflow is not split into multiple flows and used as an
input to multiple systems - the flow goes to a single System or Sink.
DynSysMod application does not read this attribute - it is ignored at the current
time.</xs:documentation>
					</xs:annotation>
				</xs:attribute>
			</xs:complexType>
		</xs:element>
	</xs:sequence>
	<xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="ReceiverType">
	<xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="SystemType">
	<xs:sequence>
		<xs:element name="TimeClock" type="TimeClockType"
minOccurs="0"/>
		<xs:choice>
			<xs:sequence>
				<xs:element name="Receiver"
type="ReceiverType" maxOccurs="unbounded"/>
				<xs:element name="Reservoir" minOccurs="0"
maxOccurs="unbounded">
					<xs:annotation>
						<xs:documentation>Reservoir names
can be "short" names, i.e. they don't have to have their System name as a
prefix.</xs:documentation>
					</xs:annotation>
					<xs:complexType>
						<xs:sequence>
							<xs:element name="Process"
type="EquationType"/>
							<xs:element
name="Capacity" minOccurs="0" msdata:Ordinal="2">
								<xs:annotation>

	<xs:documentation>If there is no Capacity element, then the Reservoir has
infinite capacity.</xs:documentation>
								</xs:annotation>
								<xs:complexType>

	<xs:simpleContent>

	<xs:extension base="xs:string">
```

```
<xs:attribute name="overflowBehavior" use="optional" default="Ignore">

<xs:simpleType>

    <xs:restriction base="xs:string">

        <xs:enumeration value="Error"/>

        <xs:enumeration value="Discard"/>

        <xs:enumeration value="Ignore"/>

    </xs:restriction>

</xs:simpleType>

</xs:attribute>

</xs:extension>

</xs:simpleContent>
                                        </xs:complexType>
                                    </xs:element>
                                </xs:sequence>
                                <xs:attribute name="name"
type="xs:string"/>
                            </xs:complexType>
                        </xs:element>
                        <xs:element name="Outflow"
type="EquationOutflowType" maxOccurs="unbounded"/>
                    </xs:sequence>
                    <xs:sequence>
                        <xs:element name="System" type="SystemType"
maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:choice>
            </xs:sequence>
            <xs:attribute name="name" type="xs:string"/>
        </xs:complexType>
        <xs:complexType name="TimeClockType">
            <xs:sequence>
                <xs:element name="Value" type="xs:string"/>
                <xs:element name="Unit" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
```

```
	<xs:complexType name="EquationOutflowType">
		<xs:complexContent>
			<xs:extension base="EquationType">
				<xs:attribute name="name" type="xs:string"
use="required"/>
				<xs:attribute name="partof" type="xs:string"
use="optional">
					<xs:annotation>
						<xs:documentation>If the partof attribute is
not present, then this Outflow is not split into multiple flows and used as an input to
multiple systems - the flow goes to a single System or Sink.
DynSysMod application does not read this attribute - it is ignored at the current
time.</xs:documentation>
					</xs:annotation>
				</xs:attribute>
			</xs:extension>
		</xs:complexContent>
	</xs:complexType>
</xs:schema>
```

Appendix C. Model Definition XSD – Graphical View

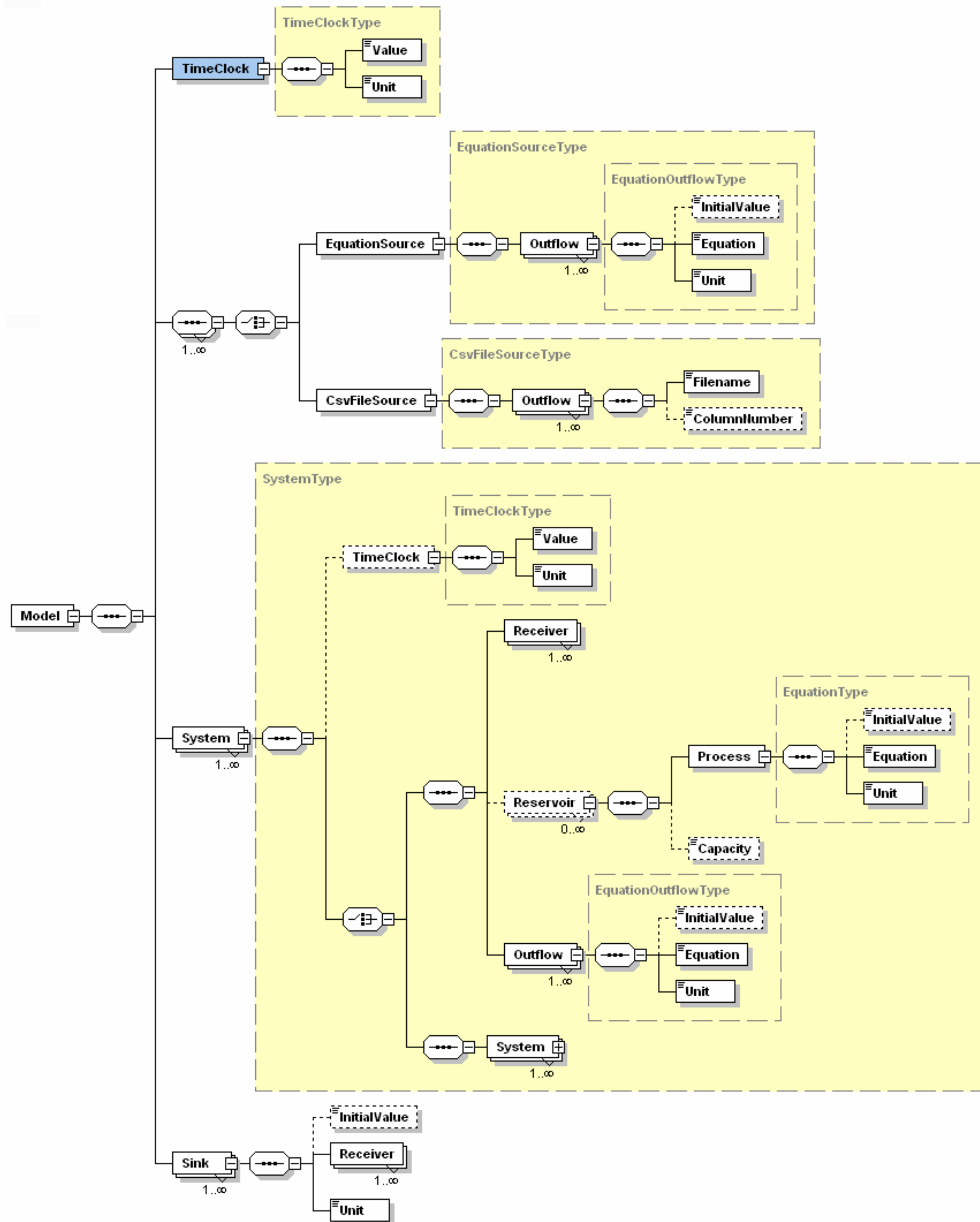


Figure C-1. A graphical representation of the model XSD.

Appendix D. DynSysMod Energy Model

```
<?xml version="1.0" encoding="UTF-8"?>
<Model xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="C:\UW Tacoma\Capstone\Final Paper\Appendix -
XSD\DynSysMod.xsd" name="">
	<TimeClock>
		<Value>1</Value>
		<Unit>year</Unit>
	</TimeClock>
	<EquationSource name="manufacturing">
		<Outflow name="rawMaterial">
			<InitialValue>1</InitialValue>
			<!-- was rawMaterial = 1-->
			<Equation>rawMaterial = 1</Equation>
			<Unit>material</Unit>
		</Outflow>
	</EquationSource>

	<EquationSource name="sun">
		<Outflow name="rawEnergy">
			<InitialValue>0.816695</InitialValue>
			<Equation>rawEnergy = 0.816695</Equation>
			<Unit>material</Unit>
		</Outflow>
	</EquationSource>


	<System name="energyConversion">

		<Receiver name="sun.rawEnergy"/>
		<Receiver name="production.manufacturedCells"/>

		<Reservoir name="cellInventory">
			<Process>
				<InitialValue>1</InitialValue>
				<Equation>cellInventory =
energyConversion.cellInventory + (1 - energyConversion.cellInventory) *
production.manufacturedCells - degradation</Equation>
				<Unit>material</Unit>
			</Process>
			<Capacity overflowBehavior="Error">1.0</Capacity>
		</Reservoir>

		<Outflow name="userConsumption" partof="freeEnergy">
```

```
				<InitialValue>0.08</InitialValue>
				<Equation>userConsumption = sun.rawEnergy * 0.08</Equation>
				<Unit>energy</Unit>
			</Outflow>

			<Outflow name="wasteHeat">
				<InitialValue>0.92</InitialValue>
				<Equation>wasteHeat = sun.rawEnergy - (sun.rawEnergy * 0.08)</Equation>
				<Unit>energy</Unit>
			</Outflow>

			<Outflow name="degradation">
				<InitialValue>0.002</InitialValue>
				<Equation>degradation = energyConversion.cellInventory * 0.002</Equation>
				<Unit>material</Unit>
			</Outflow>

			<Outflow name="workEnergy" partof="freeEnergy">
				<InitialValue>0.05</InitialValue>
				<Equation>workEnergy = 0.05 * sun.rawEnergy</Equation>
				<Unit>energy</Unit>
			</Outflow>

		</System>

		<System name="production">

			<Receiver name="manufacturing.rawMaterial"/>
			<Receiver name="energyConversion.workEnergy"/>

			<Reservoir name="rawMaterialInventory">
				<Process>
					<InitialValue>1</InitialValue>
					<Equation>rawMaterialInventory = production.rawMaterialInventory + (1 - production.rawMaterialInventory) * manufacturing.rawMaterial</Equation>
					<Unit>material</Unit>
				</Process>
				<Capacity overflowBehavior="Error">1.0</Capacity>
			</Reservoir>

			<Outflow name="manufacturedCells">
				<InitialValue>0.24</InitialValue>
```

```
				<Equation>manufacturedCells = 0.8 *
production.rawMaterialInventory * 0.3 * energyConversion.workEnergy</Equation>
				<Unit>material</Unit>
			</Outflow>

			<Outflow name="wasteHeat">
				<InitialValue>0.7</InitialValue>
				<Equation>wasteHeat = 0.7 *
energyConversion.workEnergy</Equation>
				<Unit>energy</Unit>
			</Outflow>

			<Outflow name="wasteMaterial">
				<InitialValue>0.2</InitialValue>
				<Equation>wasteMaterial = 0.2 *
production.rawMaterialInventory</Equation>
				<Unit>material</Unit>
			</Outflow>

		</System>

		<Sink name="userConsumption">
			<InitialValue>0</InitialValue>
			<Receiver name="energyConversion.userConsumption"/>
			<Unit>energy</Unit>
		</Sink>
		<Sink name="wasteHeatConversion">
			<InitialValue>0</InitialValue>
			<Receiver name="energyConversion.wasteHeat"/>
				<Receiver name="production.wasteHeat"/>
			<Unit>energy</Unit>
		</Sink>
		<Sink name="solarCellDegradation">
			<InitialValue>0</InitialValue>
			<Receiver name="energyConversion.degradation"/>
			<Unit>material</Unit>
		</Sink>
		<Sink name="wasteMaterial">
			<InitialValue>0</InitialValue>
			<Receiver name="production.wasteMaterial"/>
			<Unit>material</Unit>
		</Sink>

</Model>
```

Appendix E. Equation Grammar

```
// Author:  Jennifer Leaf, with significant help - The base of this grammar was
// reused from a sample expression parser grammar on the ANTLR website, by Kunle
// Odutola.
// http://www.antlr.org/wiki/display/ANTLR3/Five+minute+introduction+to+ANTLR+3

// University of Washington, Tacoma
// TCSS 702 - Design Project
// DynSysMod

// This grammar file defines the legal syntax for equations in the DynSysMod
// dynamic systems modeling application.  This grammar file is used by the
// ANTLR Parser Generator tool (ANother Tool for Language Recognition),
// developed by Terence Parr of the University of San Francisco.  ANTLR
// generates Java source files for a lexer and parser for the language
// defined by this grammar file.  The Java source code is then incorporated
// into the DynSysMod application.  The code, when executed, consumes string-based
// equations, and generates a parse tree for the sentence.  DynSysMod then
// walks the source tree to evaluate the equation's value.

// In order to convert this file into a lexer and parser, run the following command
// at the command line.  Assumes all ANTLR libraries are on the CLASSPATH.

// The ANTLR software can be downloaded from http://www.antlr.org/download.html.
// Once the software is downloaded and installed, you will need to add the ANTLR
// jar files to the CLASSPATH environment variable.  The README file that comes
// with the ANTLR software should explain which files to add to the CLASSPATH.

// To generate the Java code from the grammar file, run the following command
// from a command prompt.

// java org.antlr.Tool [grammarname].g -debug

// The -debug flag is necessary to generate a parse tree, rather than the
// simple AST (abstract syntax tree) which strips out which rules were used
// to identify the leaf tokens.

// Then just overwrite the appropriate files in the DynSysMod source code
// folders.



// The name of the grammar.  The name is also used to construct the names of
// the Java lexer and parser code.  It is possible to write separate grammar
// files for the lexer and parser rules.
grammar Equations;
// Not sure if this next line is needed.
options {output=AST;}

// Literal tokens.  This section is used to define literal tokens that don't need
// the firepower or complexity of an individual lexer rule, such as keywords.
tokens {
        PLUS          = '+' ;
        MINUS         = '-' ;
        MULT          = '*' ;
        DIV           = '/' ;
        T             = 't' ;
        DT            = 'dt';
        DECIMAL_POINT = '.';
}

// Header lines that will be copied verbatim into the top of the autogenerated
// lexer code.
@lexer::header {

// This file autogenerated by the ANTLR Tool.  Do not manually edit this file.
// Refer to Equations.g in the DynSysMod documentation for more information.
```

```
package DynSysMod.Equations;

}

// Header lines that will be copied verbatim into the top of the autogenerated
// parser code.
@header {

// This file autogenerated by the ANTLR Tool.  Do not manually edit this file.
// Refer to Equations.g in the DynSysMod documentation for more information.

package DynSysMod.Equations;

import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import org.antlr.runtime.debug.*;

}

// The following Java code is copied verbatim into the generated Java parser class.
@members {

    // This constructor is needed in order to create a parse tree.
    public EquationsParser(TokenStream input, ParseTreeBuilder builder) {
        super(input, builder);
    }

    // This main method is useful for testing the interpreter in a standalone mode.
    public static void main(String[] args) throws Exception {
        EquationsLexer lex = new EquationsLexer(new ANTLRStringStream("materials(t -
dt)"));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        ParseTreeBuilder builder = new ParseTreeBuilder("expr");

        EquationsParser parser = new EquationsParser(tokens, builder);

        try {
            // The start rule of the grammar.
            parser.expr();
            System.out.println(builder.getTree().toStringTree());
        } catch (RecognitionException e)  {
            e.printStackTrace();
        }
    }
}

/*------------------------------------------------------------------
 * PARSER RULES
 *------------------------------------------------------------------*/

// The parser rules define names for different types of tokens.  Parser rules can
// be constructed by literal strings, fragments, tokens (defined in the tokens
// section at the top of the file), lexer rules, and other parser rules.  By ANTLR
// convention parser rule names are written using all lowercase letters.

// The left side of each parser rule is the name of the method that
// performs the expansion of the rule.

// The first set of parser rules serve to tease apart complex pieces
// of the equation into its constituent tokens.

// This is the start rule.  If all parser rules in a grammar refer to another rule,
// then ANTLR isn't able to figure out which rule should be the start rule.  In this
// grammar, all rules are reachable via another rule, so you have to make a "dummy"
// start rule that no other rule refers to.
startRule : expr;

// These rules enforce operator and parentheses precedence.  Basically the
// operators with lower precedence appear in rules earlier that rules that
// contain operators with higher precedence.  The later rules will be fully
```

```
// expanded and interpreted first, enforcing operator precedence.
expr    : term ( ( PLUS | MINUS )  term )* ;

term    : unary ( ( MULT | DIV ) unary )* ;

unary   : '-' unary | factor;

factor  : '(' expr ')' | number | t | dt | identifier;

// The second set of parser rules are used to identify types of leaf nodes.
// The leaf nodes are the actual tokens (text) of the equation.

// Why are these separate parser rules?  If we generate a parse tree
// from this grammar, it will be easy to distinguish these special
// keywords.
identifier :   IDENTIFIER;
number     :   NUMBER;
t          :   T;
dt         :   DT;

/*------------------------------------------------------------------
 * LEXER RULES
 *------------------------------------------------------------------*/

// The lexer rules define names for different types of tokens.  Lexer rules can
// be constructed by literal strings, fragments, tokens (defined in the tokens
// section at the top of the file), and other lexer rules.  By ANTLR convention
// lexer rule names are written using all uppercase letters.

// Allows any string representation of a floating point number, following
// the Java language Double.toString() formatting.
// http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Double.html#toString(double)
NUMBER  : (DIGIT)+ (DECIMAL_POINT (DIGIT)+)? ('E' ('-')? (DIGIT)+)?;

// The action (in curly brackets) causes these tokens to be handled separately
// from the rest of the input tokens.  The parser and lexer communicate using
// channels.  The HIDDEN channel still sends tokens to the parser from the lexer,
// but by default the parser does not listen to the hidden channel so it ignores
// any tokens that are assigned to the channel.  This allows the equations to have
// any random whitespace in the sentence without impacting the parse tree.
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+        { $channel = HIDDEN; } ;

// NOTE: identifiers are not able to distinguish a variable from a function name.
// The interpreter will have to make this determination.  Identifiers must start
// with a letter, and can be followed by any number of letters, numbers, or the
// underscore character.  Identifiers can consist of multiple names.
IDENTIFIER : NAME (NAME_DELIMITER NAME)*;

// Fragments are used to make lexer rules (or parser rules) easier to read.
// Fragments do not identify tokens - they have to be used in other rules.
// Using fragments allows you to build more complex constructions than the
// tokens section (see the top of the file), without having the lexer or
// parser recognize these constructions individually.

fragment DIGIT : '0'..'9' ;

fragment LETTER : 'A'..'Z' | 'a'..'z' ;

fragment NON_INITIAL_CHARACTER : LETTER | DIGIT | '_';

fragment NAME :       LETTER (NON_INITIAL_CHARACTER)*;

// This rule has to be defined here instead of in the tokens section,
// since there is another rule in the tokens section that assigns a name
// to the '.' character.
fragment NAME_DELIMITER : '.';
```

Appendix F. UML Diagrams of Interpreter Algorithms

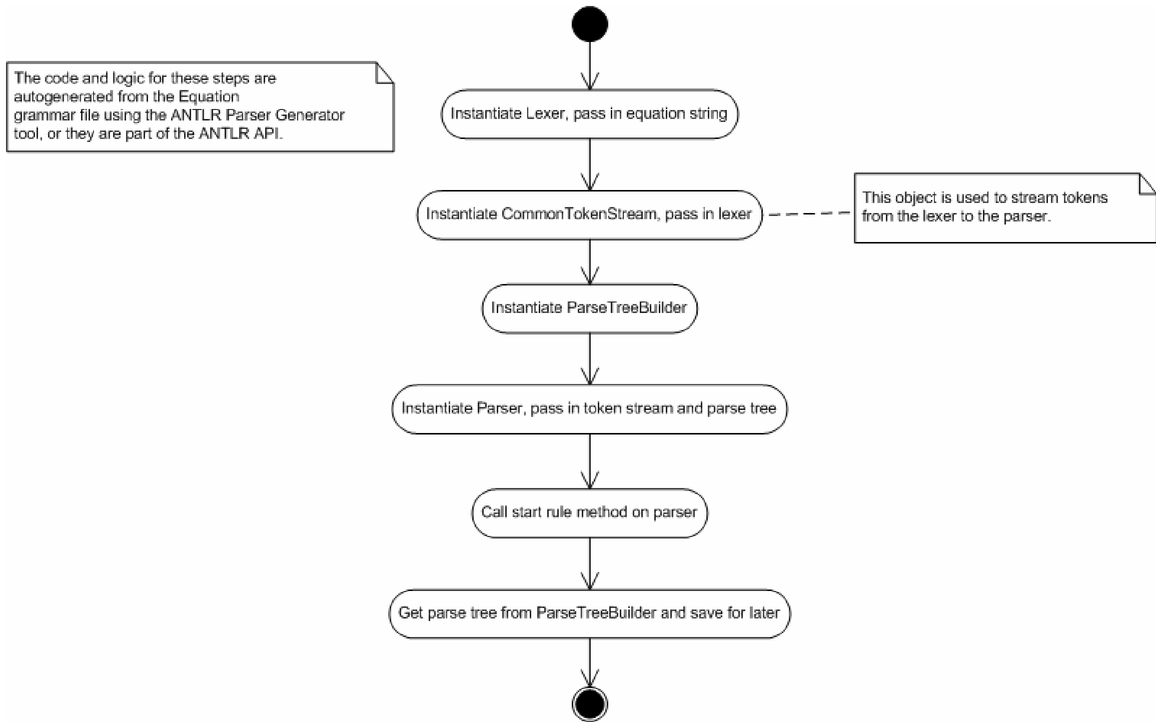


Figure F-1. UML Activity diagram of interpreter initialization.

Figure F-2. UML Activity diagram of main interpreter Evaluate method.

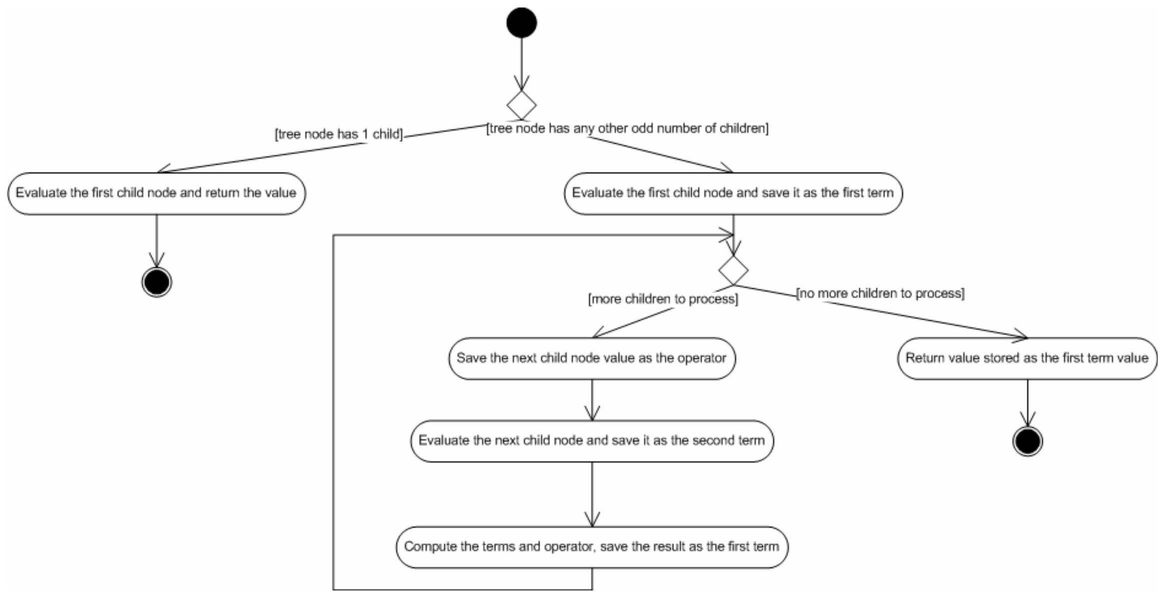


Figure F-3. UML Activity diagram of main arithmetic rule evaluation.

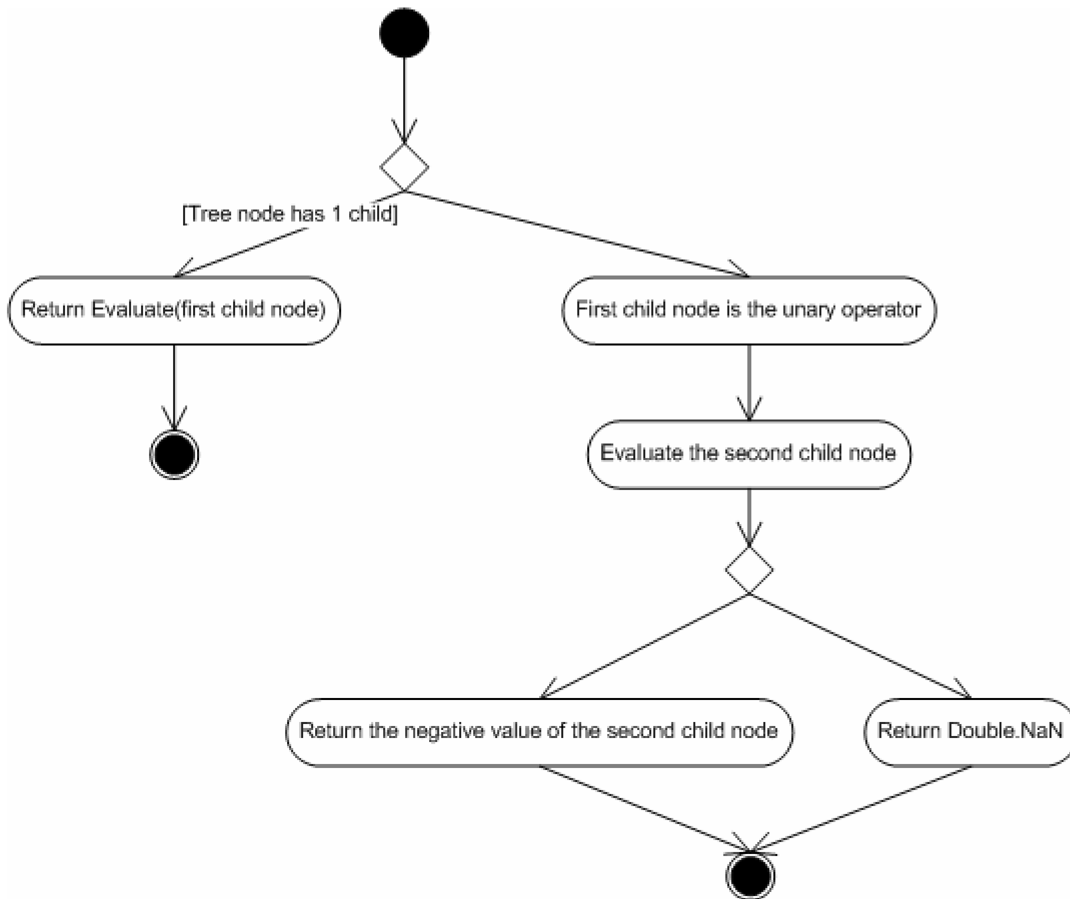


Figure F-4. UML Activity diagram of "unary" parser rule evaluation.

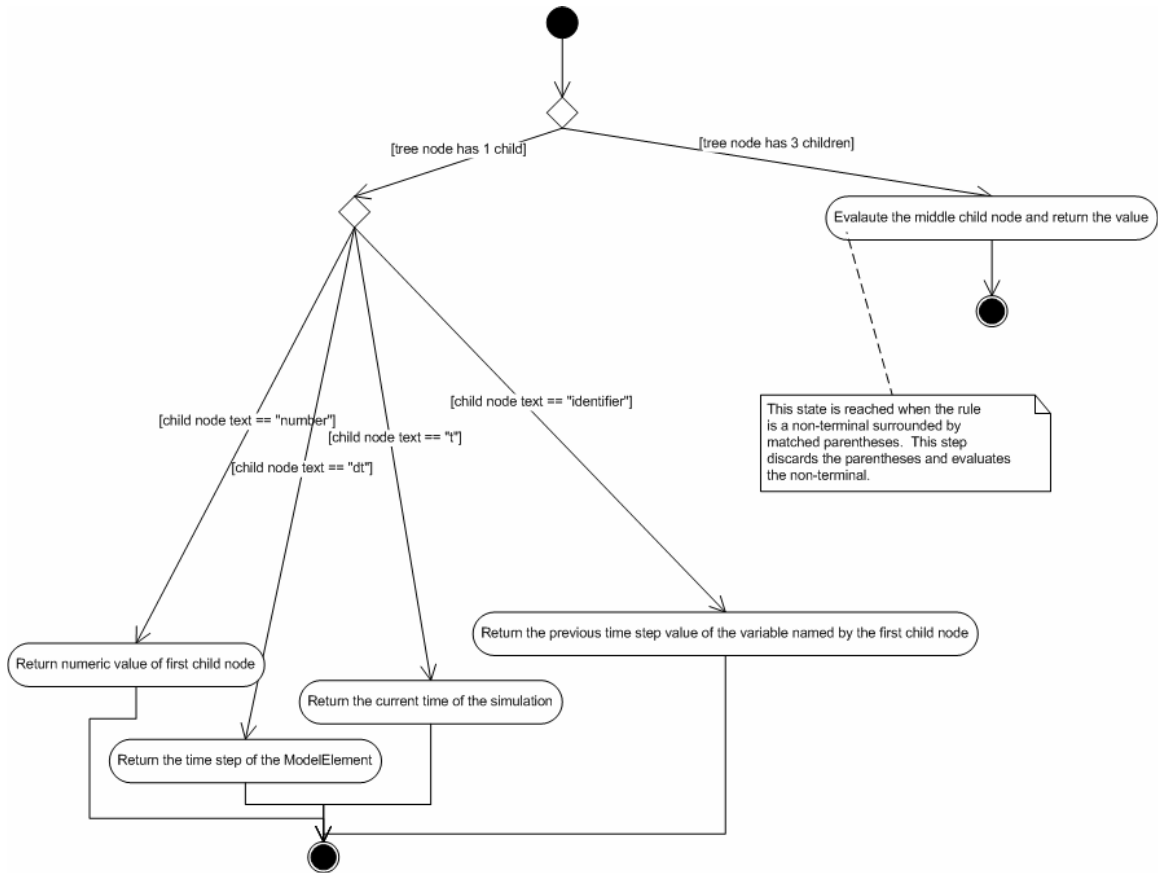


Figure F-5. UML Activity diagram of “factor” parser rule evaluation.

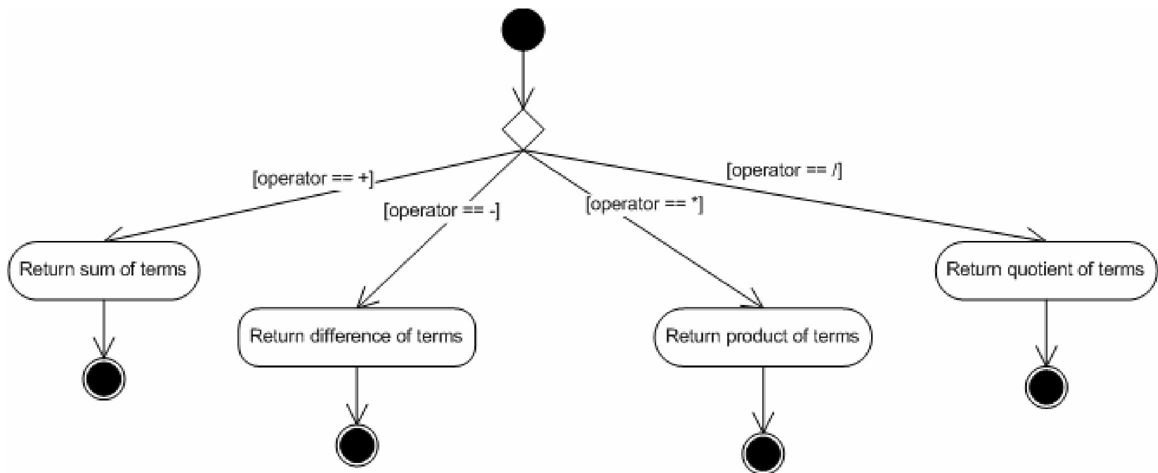


Figure F-6. UML Activity diagram of Compute method in the Interpreter class.

Appendix G. UML Diagrams of Simulation Initialization Algorithms

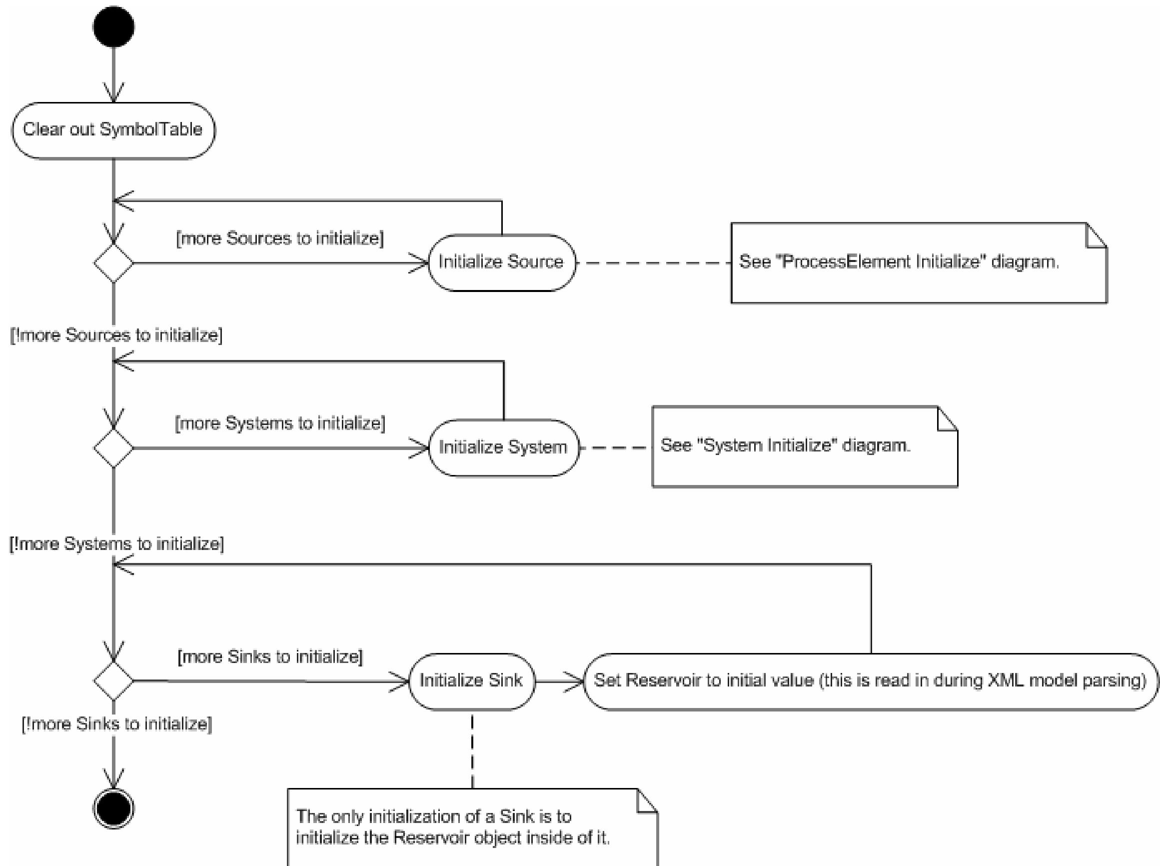


Figure G-1. UML Activity diagram of main simulation initialization algorithm.

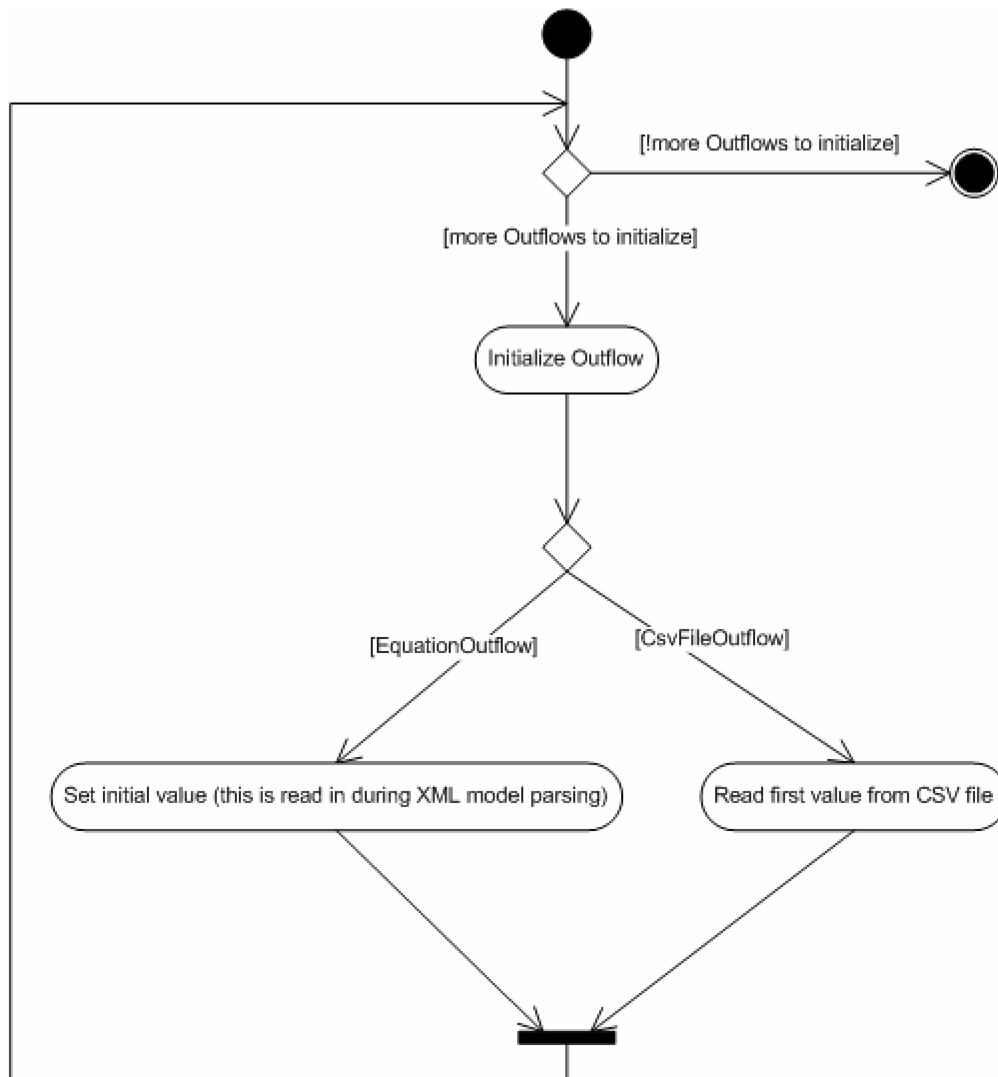


Figure G-2. UML Activity diagram of ProcessElement class initialization.

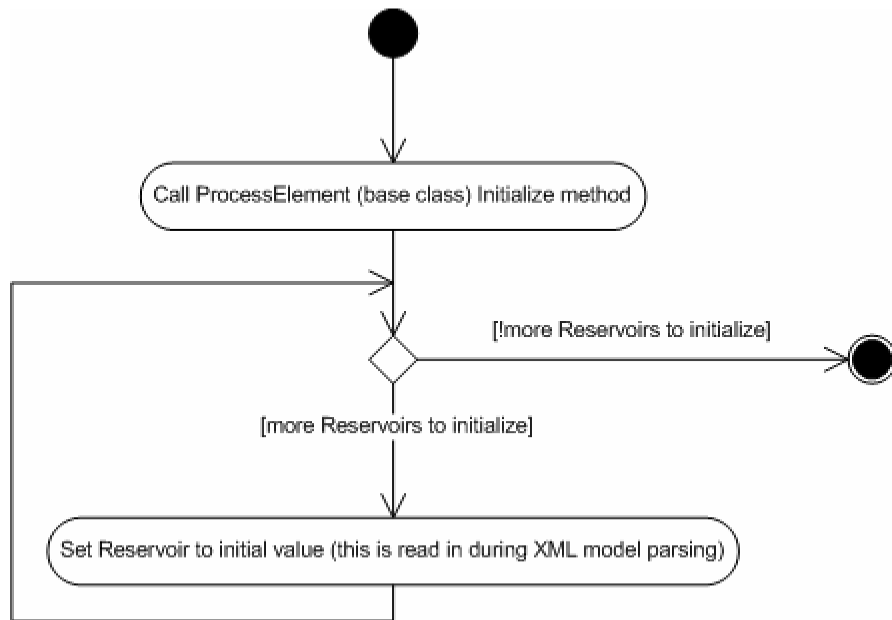


Figure G-3. UML Activity diagram of System class initialization.

Appendix H. UML Diagrams of Simulation Algorithms (one time step)

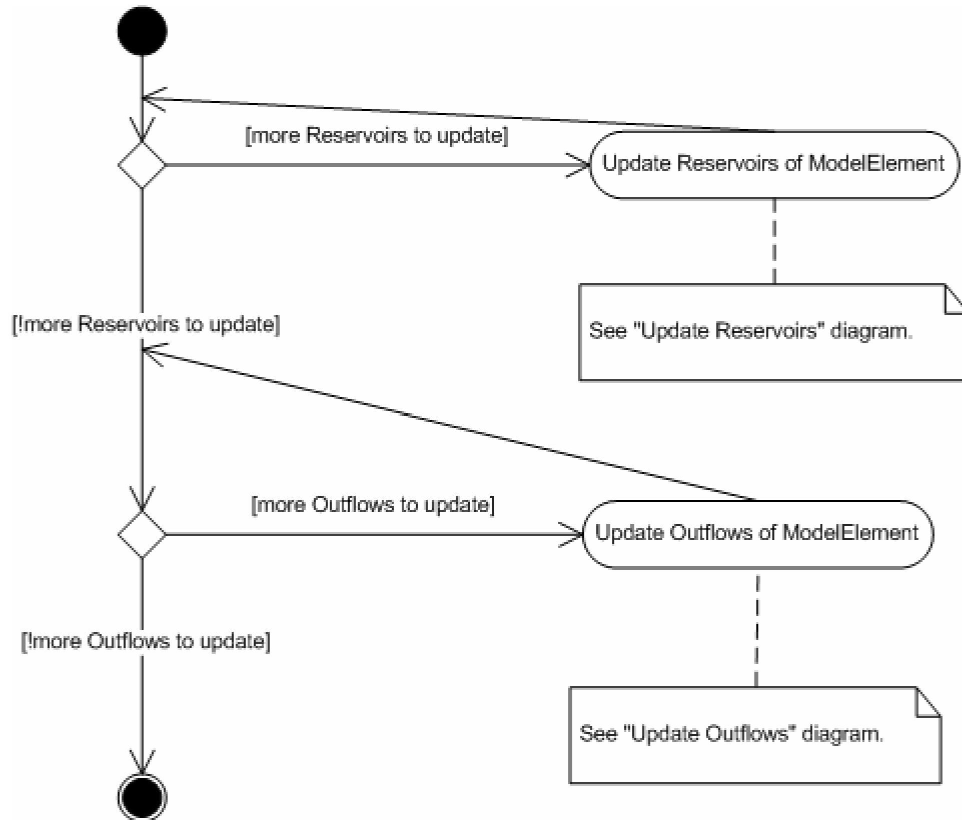


Figure H-1. UML Activity diagram of main simulation update algorithm (one time step).

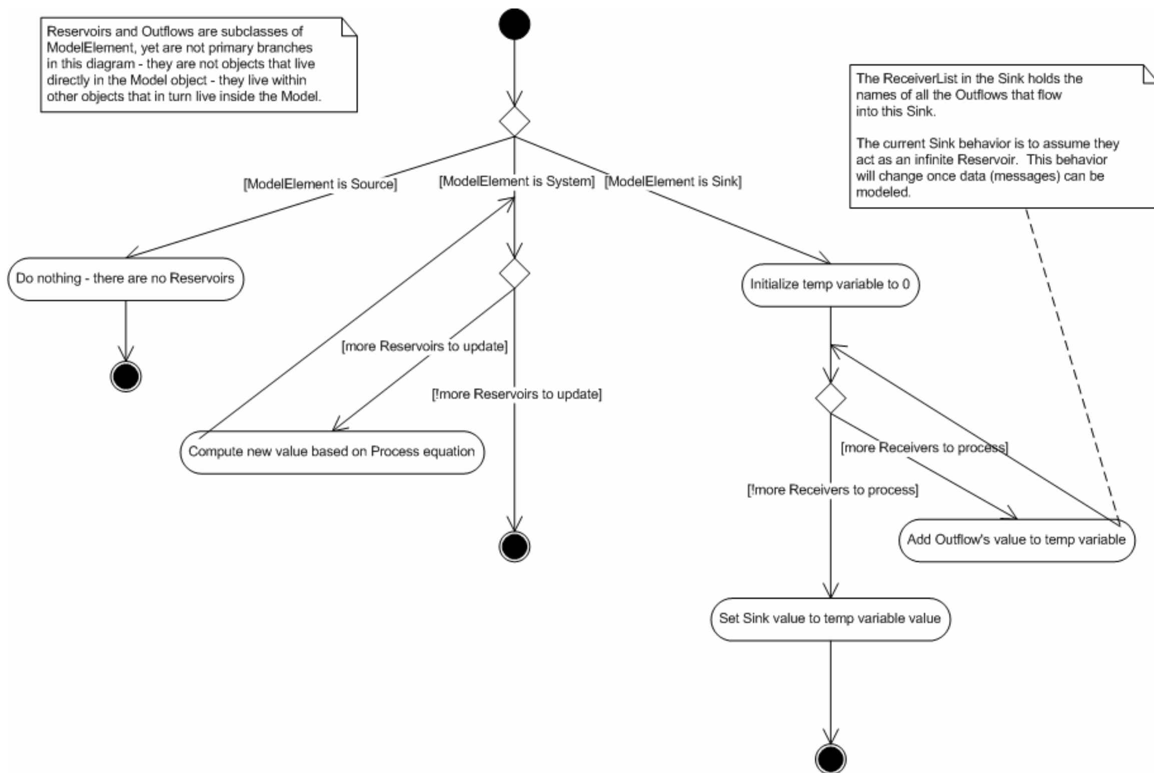


Figure H-2. UML Activity diagram of updating the Reservoirs for Model elements.

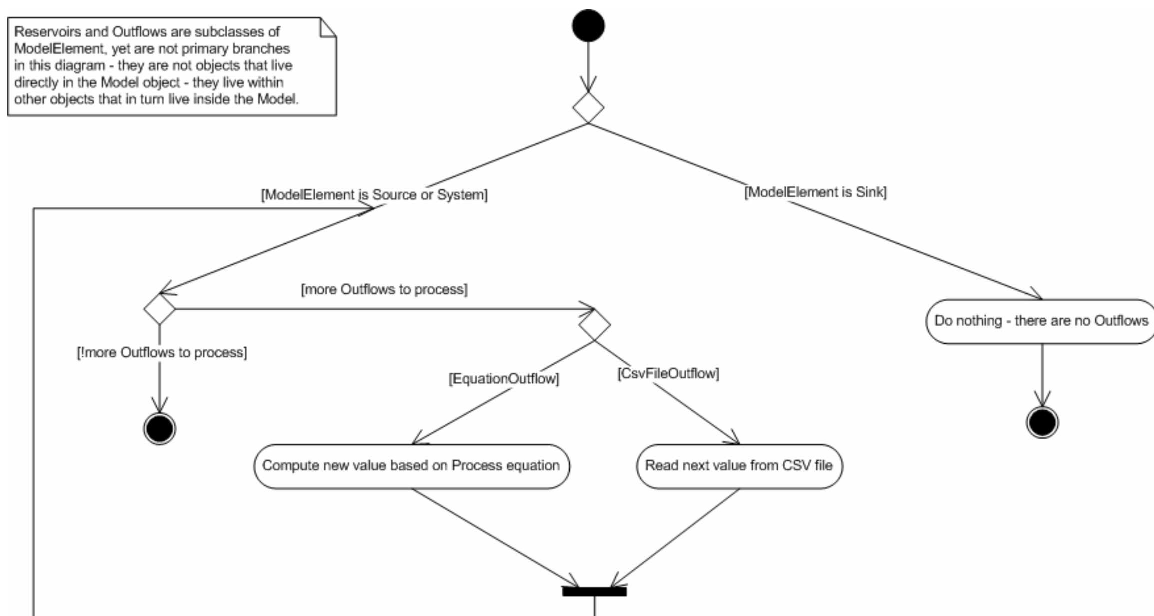


Figure H-3. UML Activity diagram of updating the Outflows for Model elements.

Appendix I. UML Class Diagrams

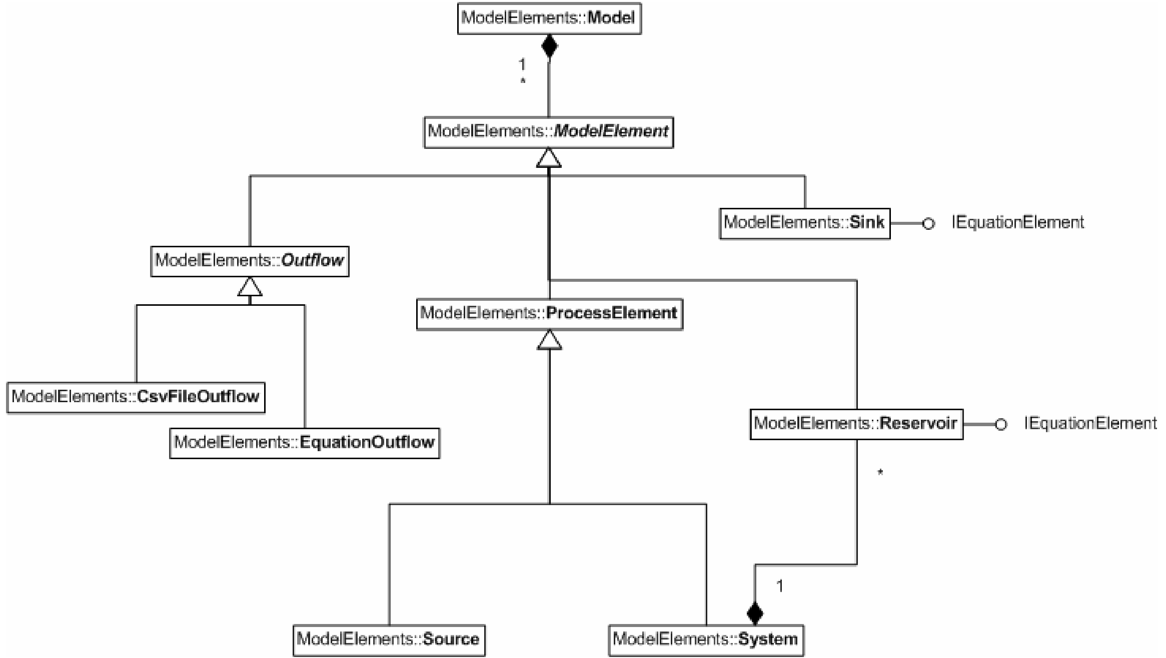
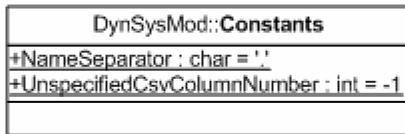
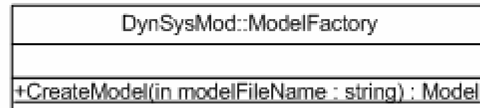
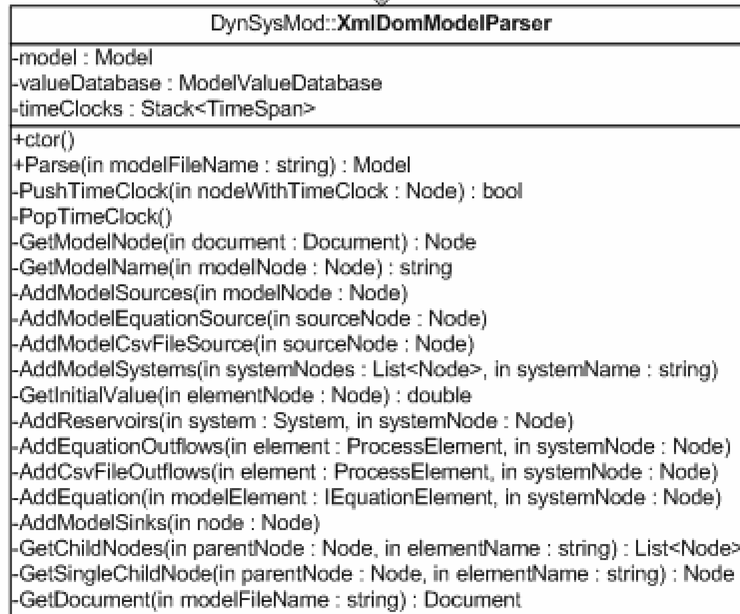
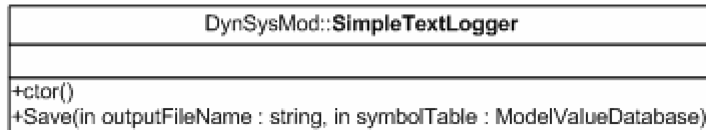


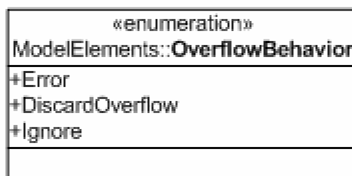
Figure I-1. Simplified UML Class diagram of Model element classes.











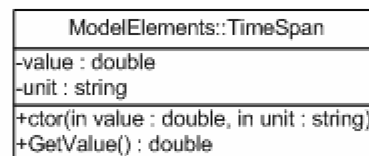


Figure I-2. UML Class diagram of other classes used in DynSysMod.

Appendix J. Building the Source Code and Running the Application

DynSysMod was developed using the NetBeans Integrated Development Environment (IDE). [36] This appendix assumes a basic familiarity with IDEs in general. While the source code can be compiled using the basic javac compiler and can be edited using any Java IDE, the instructions below assume you are using NetBeans.

The source code files should be installed to the filesystem in the following relative paths. Remember where the "src" folder is.

```
src\DynSysMod\Constants.java
src\DynSysMod\ModelFactory.java
src\DynSysMod\SimpleMainFrame.java
src\DynSysMod\ValueComputationException.java
src\DynSysMod\XmlDomModelParser.java
src\DynSysMod\Equations\EquationInterpreterException.java
src\DynSysMod\Equations\EquationsLexer.java
src\DynSysMod\Equations\EquationsParser.java
src\DynSysMod\Equations\Interpreter.java
src\DynSysMod\Logging\SimpleTextLogger.java
src\DynSysMod\ModelElements\CapacityOverflowException.java
src\DynSysMod\ModelElements\ColumnHeaderNotFoundException.java
src\DynSysMod\ModelElements\CsvFileOutflow.java
src\DynSysMod\ModelElements\DuplicateElementException.java
src\DynSysMod\ModelElements\DuplicateValueException.java
src\DynSysMod\ModelElements\EquationOutflow.java
src\DynSysMod\ModelElements\IEquationElement.java
src\DynSysMod\ModelElements\Model.java
src\DynSysMod\ModelElements\ModelElement.java
src\DynSysMod\ModelElements\ModelFormatException.java
src\DynSysMod\ModelElements\ModelInitializationException.java
src\DynSysMod\ModelElements\ModelValueDatabase.java
src\DynSysMod\ModelElements\Outflow.java
src\DynSysMod\ModelElements\ProcessElement.java
src\DynSysMod\ModelElements\Reservoir.java
src\DynSysMod\ModelElements\Sink.java
src\DynSysMod\ModelElements\Source.java
src\DynSysMod\ModelElements\System.java
src\DynSysMod\ModelElements\TimeSpan.java
src\DynSysMod\ModelElements\ValueNotFoundException.java
src\DynSysMod\SimulationEngine\LinearScheduler.java
src\DynSysMod\SimulationEngine\Scheduler.java
src\DynSysMod\SimulationEngine\SimulationEngine.java
src\DynSysMod\UnitTests\InterpreterTests.java
src\DynSysMod\UnitTests\ReservoirTests.java
src\DynSysMod\UnitTests\SinkTests.java
```

Once the source code has been installed on the computer, create a new project in the NetBeans IDE, by using the File -> New Project… menu option. On the New Project screen, select the "Java Project with Existing Sources" option from the General category:

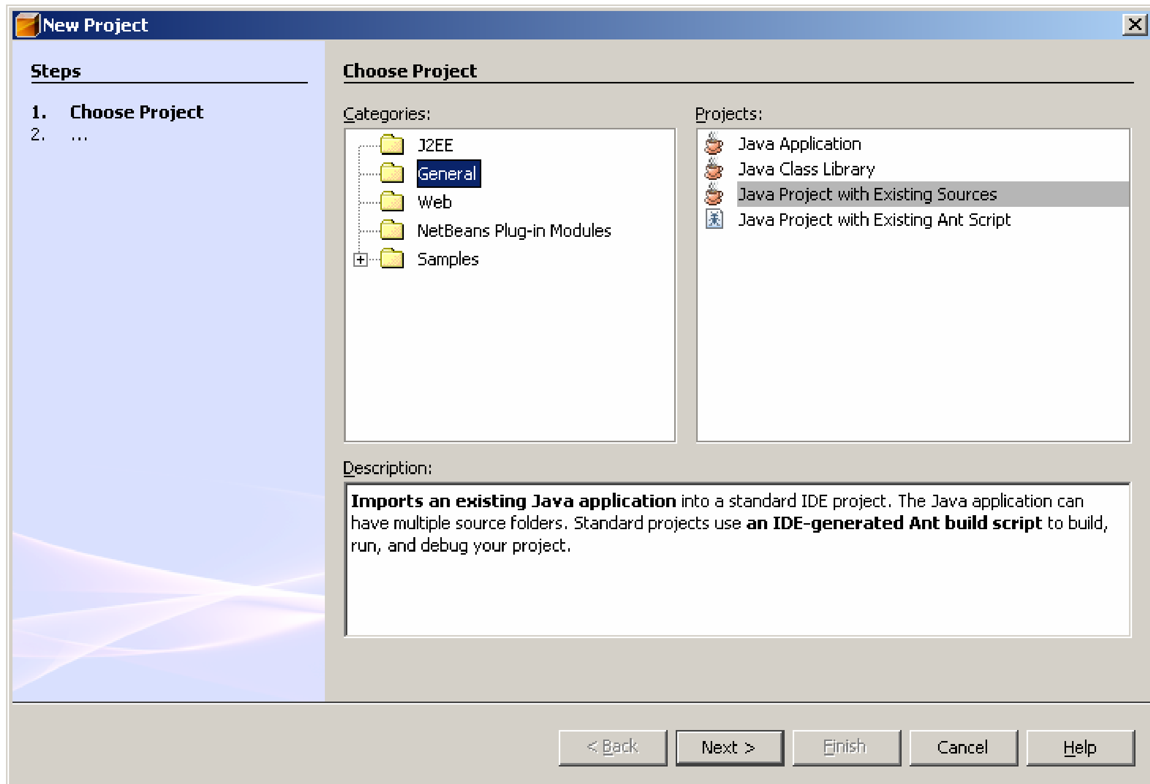


Figure J-1. The New Project screen in NetBeans v5.5. [36]

On the next screen, enter the project name (DynSysMod) and the root folder for the project (user-dependent). Finally, on the last screen, use the Add Folder… button next to the Source Package Folders listbox to navigate to the src folder that the source code is installed to.

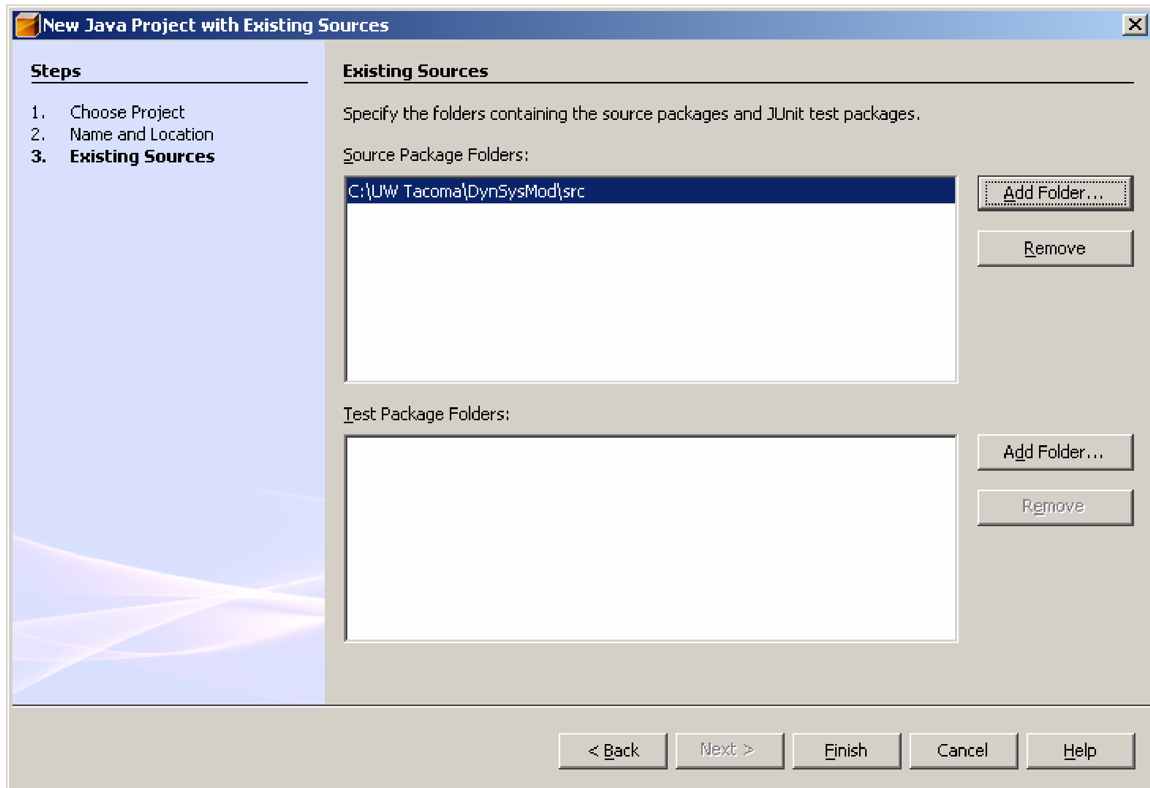


Figure J-2. The New Java Project with Existing Sources screen in NetBeans v5.5. [36]

Click the Finish button on the screen.

After the project is set up, build the application by using the Build -> Build Main Project menu option.

You can run the application using the NetBeans IDE by using the Run -> Run Main Project. Alternatively, you can run the application from the command line, by navigating to the "dist" folder under the project folder and running the following command:

```
java -jar "DynSysMod.jar"
```

Once the application is running, enter the filename of the XML model, the name of the file that the output should go to, and the duration of the simulation. The time step for the simulation is taken from the TimeClock XML element directly under the Model element. The simulation clock starts at 0. After each element in the Model is updated once, the simulation clock is incremented by the value of the Model's time clock. The simulation will stop when the simulation clock's value is larger than the duration, so keep these factors in mind when entering the values for the Model time clock and the duration to ensure the desired number of time steps occur.

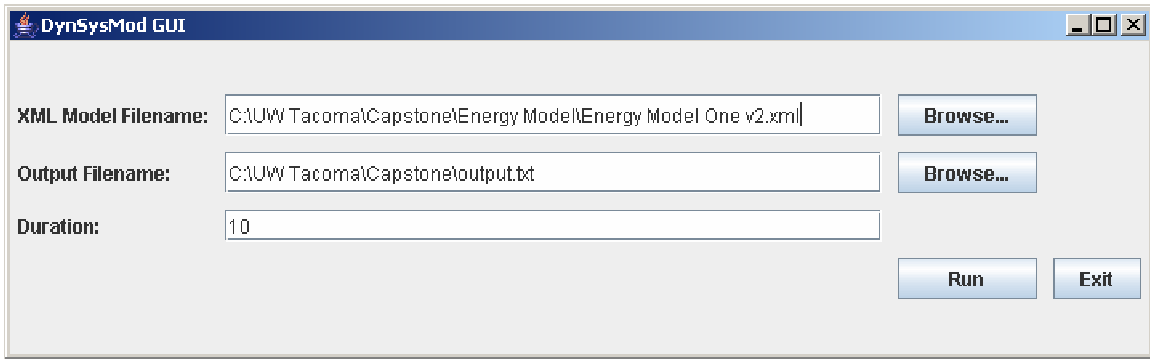


Figure J-3. The main DynSysMod application window.

After you run the simulation, a message will pop up once the simulation run has completed. At this time you can open the output file using any text editor and view the results of the simulation.

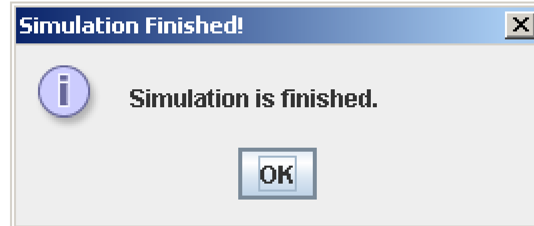


Figure J-4. The DynSysMod message window to notify the user that the simulation run is complete.

If you run the application using the NetBeans IDE in a debugging mode (by using the Run -> Debug Main Project menu option), you will be able to view the runtime logging statements generated through the log4j [21] diagnostics logging library. These logging statements enable the user to monitor the progress of the simulation as it executes. The logging statements are useful for debugging logic errors or understanding the simulation algorithms.

Appendix K. Source Code Listings

The source code files are listed in alphabetical order, ordered by package.

Constants.java

```
/**
 * File: Constants.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod;

// Import statements

/**
 * Defines constants used throughout the framework.
 * @author Jennifer Leaf
 */
public class Constants {

    /** Constructor is hidden to prevent instantiation. */
    private Constants() {
    }

    /** Character used to separate system names from model element short names. */
    public static char NameSeparator = '.';

    /** Used to indicate that the column number to use in the CSV file is unspecified. */
    public static int UnspecifiedCsvColumnNumber = -1;
}
```

ModelFactory.java

```
/**
 * File: ModelFactory.java
 * Creation Date: April 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod;

// Import statements
import DynSysMod.ModelElements.Model;
import DynSysMod.ModelElements.ModelInitializationException;
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;

/**
 * Builds a Model from the given XML file.
 * @author Jennifer Leaf
 */
public class ModelFactory {

    /**
     * Creates a new instance of the ModelFactory class.  Constructor is hidden
     * to prevent instantiation.
     */
    private ModelFactory() {
```

```
    }

    /**
     * Creates a Model instance from the contents of the given file, using an appropriate
Model parser.
     * @param modelFileName The full path and filename of the model definition.
     * @return The Model instance.
     */
    public static Model CreateModel(String modelFileName)
        throws java.io.IOException, SAXException, ParserConfigurationException,
ModelInitializationException {

        XmlDomModelParser modelParser = new XmlDomModelParser();
        return modelParser.Parse(modelFileName);
    }
}
```

SimpleMainFrame.java

```
/*
 * SimpleMainFrame.java
 *
 * Created on July 8, 2007, 7:01 PM
 */

// Package declaration
package DynSysMod;

// Import statements
import DynSysMod.Logging.SimpleTextLogger;
import DynSysMod.ModelElements.Model;
import DynSysMod.SimulationEngine.SimulationEngine;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.*;
import javax.swing.event.*;

import org.apache.log4j.*;

/**
 * A simple GUI application for simulating DynSysMod models.
 * @author Jennifer Leaf
 */
public class SimpleMainFrame extends javax.swing.JFrame {

    private static Logger logger = Logger.getLogger(SimpleMainFrame.class.getName());

    /** Creates a new instance of the SimpleMainFrame class.*/
    public SimpleMainFrame() {
        initComponents();
        this.SetUpLayout();
        this.SetUpCallbacks();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    // <editor-fold defaultstate="collapsed" desc=" Generated Code ">
    private void initComponents() {

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setTitle("DynSysMod GUI");
        org.jdesktop.layout.GroupLayout layout = new
org.jdesktop.layout.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
```

```
            layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
            .add(0, 711, Short.MAX_VALUE)
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
            .add(0, 236, Short.MAX_VALUE)
        );
        pack();
    }// </editor-fold>

    /**
     * Runs the application.
     * @param args The command line arguments.
     */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new SimpleMainFrame().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    // End of variables declaration

    private JButton buttonModelFileBrowse = new JButton("Browse...");
    private JButton buttonOutputFileBrowse = new JButton("Browse...");
    private JButton buttonRun = new JButton("Run");
    private JButton buttonExit = new JButton("Exit");

    private JTextField textFieldModelFilename = new JTextField("");
    private JTextField textFieldOutputFilename = new JTextField("");
    private JTextField textFieldDuration = new JTextField("");

    private JLabel labelModelFilename = new JLabel("XML Model Filename:");
    private JLabel labelOutputFilename = new JLabel("Output Filename:");
    private JLabel labelDuration = new JLabel("Duration:");

    private void SetUpLayout() {
        this.setLayout(new GridBagLayout());

        GridBagConstraints constraints = new GridBagConstraints();
        constraints.fill = GridBagConstraints.BOTH;
        constraints.insets = new Insets(5,5,5,5);

        // Set up the labels.
        constraints.gridx = 0; constraints.gridy = 0; constraints.gridwidth = 1;
constraints.gridheight = 1;
        constraints.weightx = 0.0;
        this.add(this.labelModelFilename, constraints);

        constraints.gridx = 0; constraints.gridy = 1; constraints.gridwidth = 1;
constraints.gridheight = 1;
        constraints.weightx = 0.0;
        this.add(this.labelOutputFilename, constraints);

        constraints.gridx = 0; constraints.gridy = 2; constraints.gridwidth = 1;
constraints.gridheight = 1;
        constraints.weightx = 0.0;
        this.add(this.labelDuration, constraints);

        // Set up the textboxes.
        constraints.gridx = 1; constraints.gridy = 0; constraints.gridwidth = 2;
constraints.gridheight = 1;
        constraints.weightx = 1.0;
        this.add(this.textFieldModelFilename, constraints);

        constraints.gridx = 1; constraints.gridy = 1; constraints.gridwidth = 2;
constraints.gridheight = 1;
        constraints.weightx = 1.0;
        this.add(this.textFieldOutputFilename, constraints);
```

```
        constraints.gridx = 1; constraints.gridy = 2; constraints.gridwidth = 2;
constraints.gridheight = 1;
        constraints.weightx = 1.0;
        this.add(this.textFieldDuration, constraints);

        // Set up the buttons.
        constraints.gridx = 3; constraints.gridy = 0; constraints.gridwidth = 1;
constraints.gridheight = 1;
        constraints.weightx = 0.0;
        this.add(this.buttonModelFileBrowse, constraints);

        constraints.gridx = 3; constraints.gridy = 1; constraints.gridwidth = 1;
constraints.gridheight = 1;
        constraints.weightx = 0.0;
        this.add(this.buttonOutputFileBrowse, constraints);

        constraints.gridx = 3; constraints.gridy = 4; constraints.gridwidth = 1;
constraints.gridheight = 1;
        constraints.weightx = 0.0;
        this.add(this.buttonRun, constraints);

        constraints.gridx = 4; constraints.gridy = 4; constraints.gridwidth = 1;
constraints.gridheight = 1;
        constraints.weightx = 0.0;
        this.add(this.buttonExit, constraints);
    }

    private void SetUpCallbacks() {
        this.buttonModelFileBrowse.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JFileChooser fileChooserModel = new JFileChooser();
                fileChooserModel.setSelectedFile(new
File(textFieldModelFilename.getText()));
                int returnVal = fileChooserModel.showOpenDialog(SimpleMainFrame.this);

                if (returnVal == JFileChooser.APPROVE_OPTION) {
                    File file = fileChooserModel.getSelectedFile();
                    textFieldModelFilename.setText(file.getAbsolutePath());
                } else {
                    // The user canceled the dialog.
                }
            }
        });

        this.buttonOutputFileBrowse.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JFileChooser fileChooserModel = new JFileChooser();
                int returnVal = fileChooserModel.showSaveDialog(SimpleMainFrame.this);

                if (returnVal == JFileChooser.APPROVE_OPTION) {
                    File file = fileChooserModel.getSelectedFile();
                    textFieldOutputFilename.setText(file.getAbsolutePath());
                } else {
                    // The user canceled the dialog.
                }
            }
        });

        this.buttonRun.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                BasicConfigurator.configure();

                try {
                    String modelFileName =
SimpleMainFrame.this.textFieldModelFilename.getText();

                    Model model = ModelFactory.CreateModel(modelFileName);

                    SimulationEngine engine = new SimulationEngine(model,
                            model.GetTimeClock().GetValue(),
```

```
Double.parseDouble(SimpleMainFrame.this.textFieldDuration.getText())
                            );

                    engine.Run();

                    // We're done, save the results.
                    SimpleTextLogger sl = new SimpleTextLogger();
                    sl.Save(SimpleMainFrame.this.textFieldOutputFilename.getText(),
                            model.GetValueDatabase());

                } catch (Exception exception) {
                    // For now, just die.
                    logger.fatal("Application shutting down due to an exception.",
exception);
                    java.lang.System.exit(0);
                }

                JOptionPane.showMessageDialog(SimpleMainFrame.this, "Simulation is
finished.", "Simulation Finished!", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        this.buttonExit.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                dispose();
            }
        });
    }
}
```

ValueComputationException.java

```
/**
 * File: ValueComputationException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod;

// Import statements

/**
 * Indicates an exceptional circumstance while computing a value.
 * @author Jennifer Leaf
 */
public class ValueComputationException extends Exception {

    /** Constructs a new exception with null as its detail message. */
    public ValueComputationException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public ValueComputationException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public ValueComputationException(String message, Throwable cause) {
        super(message, cause);
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
```

```
    public ValueComputationException(Throwable cause) {
        super(cause);
    }

}
```

XmlDomModelParser.java

```
/**
 * File: XmlDomModelParser.java
 * Creation Date: April 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod;

// Import statements
import DynSysMod.ModelElements.*;
// The following import statement hides ambiguous reference between this and
java.lang.System.
import DynSysMod.ModelElements.System;

import java.io.*;
import java.util.*;
import javax.xml.parsers.*;

import org.w3c.dom.*;
import org.xml.sax.*;

import org.apache.log4j.*;

/**
 * Constructs a Model from an XML definition, using an XML DOM parser engine
 * All XML parsing happens in here, in order to make it as easy as possible to
 * tweak the XML parsing algorithms if necessary.
 * @author Jennifer Leaf
 */
public class XmlDomModelParser {

    /** Logs statements and other information during runtime. */
    private static Logger logger = Logger.getLogger(XmlDomModelParser.class.getName());

    /** The model instance being constructed. */
    private Model model;
    /** The symbol table instance - shared by all elements in this.model. */
    private ModelValueDatabase valueDatabase;
    /** A collection of time clocks for the model. The top element in stack is the
currently scoped time clock.
     * As child nodes of the model or systems are found to have their own time clock, it
is pushed onto this
     * stack, and popped when that node goes out of scope (i.e., all of its children, and
itself, have been
     * processed. */
    private Stack<TimeSpan> timeClocks;

    /** Creates a new instance of the XmlDomModelParser class. */
    public XmlDomModelParser() {
        // To turn logging back on, just uncomment this line.
        logger.setLevel(Level.OFF);
    }

    /**
     * Creates a Model instance from the contents of the given file.
     * @param modelFileName The full path and filename of the model definition.
     * @return The Model instance.
     */
    public Model Parse(String modelFileName)
```

```java
    throws IOException, SAXException, ParserConfigurationException,
ModelInitializationException {
        Document document = this.GetDocument(modelFileName);

        this.timeClocks = new Stack<TimeSpan>();

        Node modelNode = this.GetModelNode(document);
        String modelName = this.GetModelName(modelNode);

        logger.info("Model name = " + modelName);

        this.valueDatabase = new ModelValueDatabase();

        this.PushTimeClock(modelNode);

        this.model = new Model(modelName, this.valueDatabase, this.timeClocks.peek());

        this.AddModelSources(modelNode);

        this.AddModelSystems(this.GetChildNodes(modelNode, "System"), "");

        this.AddModelSinks(modelNode);

        return this.model;
    }

    /**
     * Finds the TimeClock element within the given Node if one exists, and pushes a new
TimeSpan
     * onto the stack.  If no such child element exists then the stack is left unchanged.
     * @param nodeWithTimeClock The XML element that has a child element TimeClock.
     * @returns true if a child TimeClock element was found, otherwise false.
     */
    private boolean PushTimeClock(Node nodeWithTimeClock) throws
ModelInitializationException {
        Node timeSpanNode = this.GetSingleChildNode(nodeWithTimeClock, "TimeClock");
        if (timeSpanNode != null) {
            String valueAsString = this.GetSingleChildNode(timeSpanNode,
"Value").getTextContent();
            double value = Double.parseDouble(this.GetSingleChildNode(timeSpanNode,
"Value").getTextContent());
            String unit = this.GetSingleChildNode(timeSpanNode, "Unit").getTextContent();
            this.timeClocks.push(new TimeSpan(value, unit));
            return true;
        }
        else {
            return false;
        }
    }

    private void PopTimeClock() {
        this.timeClocks.pop();
    }

    private Node GetModelNode(Document document) throws ModelInitializationException {
        NodeList modelNodes = document.getElementsByTagName("Model");

        if (modelNodes.getLength() == 1) {
            return modelNodes.item(0);
        }
        else if (modelNodes.getLength() > 1) {
            String errorMessage = "More than 1 Model node found.";
            logger.error(errorMessage);
            throw new ModelFormatException(errorMessage);
        }
        else { // if (modelNodes.getLength() == 0)
            String errorMessage = "No Model nodes found.";
            logger.error(errorMessage);
            throw new ModelFormatException(errorMessage);
        }
    }
```

```
    private String GetModelName(Node modelNode) {
        return modelNode.getAttributes().getNamedItem("name").getNodeValue();
    }

    private void AddModelSources(Node modelNode) throws ModelInitializationException,
java.io.FileNotFoundException, java.io.IOException {
        // The DOM API has no way to request nodes with a specific name that are
        // immediate descendants only, so we have to walk the children ourselves
        // and pick out what we are interested in.
        for (int i = 0; i < modelNode.getChildNodes().getLength(); i++) {
            Node childNode = modelNode.getChildNodes().item(i);
            logger.debug("Found child node " + childNode.getNodeName() + " of the
Model.");

            if (childNode.getNodeName() == "EquationSource") {
                this.AddModelEquationSource(childNode);
            }
            else if (childNode.getNodeName() == "CsvFileSource") {
                this.AddModelCsvFileSource(childNode);
            }
            else {
                // Skip it and move on.
            }

        }
    }

    private void AddModelEquationSource(Node sourceNode) throws
ModelInitializationException {
        String sourceName =
sourceNode.getAttributes().getNamedItem("name").getNodeValue();
        logger.debug("Initializing model level Equation Source object " + sourceName +
".");

        Source source = new Source(sourceName, this.valueDatabase,
this.timeClocks.peek());
        this.model.AddSource(sourceName, source);

        // Add the Process equations.
        this.AddEquationOutflows(source, sourceNode);
    }

    private void AddModelCsvFileSource(Node sourceNode) throws
ModelInitializationException, java.io.FileNotFoundException, java.io.IOException {
        String sourceName =
sourceNode.getAttributes().getNamedItem("name").getNodeValue();
        logger.debug("Initializing model level CSV File Source object " + sourceName +
".");

        Source source = new Source(sourceName, this.valueDatabase,
this.timeClocks.peek());
        this.model.AddSource(sourceName, source);

        this.AddCsvFileOutflows(source, sourceNode);
    }

    private void AddModelSystems(List<Node> systemNodes, String systemName) throws
ModelInitializationException {
        for (int i = 0; i < systemNodes.size(); i++) {
            // Check for a TimeSpan node.
            boolean wasNewTimeSpanFound = this.PushTimeClock(systemNodes.get(i));

            // Check for the base case BEFORE we recurse.
            List<Node> childNodes = this.GetChildNodes(systemNodes.get(i), "System");
            String currentNodeName =
systemNodes.get(i).getAttributes().getNamedItem("name").getNodeValue();
            logger.debug("System node " + currentNodeName + " has " + childNodes.size() +
" subsystems.");
            if (childNodes.size() > 0) {
                String newSystemName = systemName + currentNodeName + ".";
```

```
                this.AddModelSystems(childNodes, newSystemName);
            }
            else {
                // We have found a base case (a System with no sub-Systems), so
                // add this node to the model.
                logger.debug("Initializing System object " + systemName + currentNodeName
+ ".");
                System system = new System(systemName + currentNodeName,
this.valueDatabase, this.timeClocks.peek());
                this.model.AddSystem(systemName + currentNodeName, system);

                // Build the Reservoir objects, but don't initialize it.  It will
                // get initialized on the second pass of the model initialization.
                this.AddReservoirs(system, systemNodes.get(i));

                // Build the Flow objects.
                this.AddEquationOutflows(system, systemNodes.get(i));
            }

            if (wasNewTimeSpanFound) {
                this.PopTimeClock();
            }
            else {
                // This System doesn't have its own time clock, so leave the current
                // clock alone.
            }
        }
    }

    private double GetInitialValue(Node elementNode) throws ModelInitializationException
{
        if (this.GetSingleChildNode(elementNode, "InitialValue") != null) {
            String initialValueAsString = this.GetSingleChildNode(elementNode,
"InitialValue").getTextContent();
            logger.debug("Initial value (string) = " + initialValueAsString);
            return Double.parseDouble(initialValueAsString);
        }
        else {
            // There is no InitialValue node in the XML file for this element so use the
default value.
            return 0;
        }
    }

    private void AddReservoirs(System system, Node systemNode) throws
ModelInitializationException {
        List<Node> reservoirNodes = this.GetChildNodes(systemNode, "Reservoir");
        for (int i = 0; i < reservoirNodes.size(); i++) {
            Node reservoirNode = reservoirNodes.get(i);
            String reservoirName =
reservoirNodes.get(i).getAttributes().getNamedItem("name").getNodeValue();
            logger.debug("Initializing Reservoir object " + reservoirName + ".");

            Node processNode = this.GetSingleChildNode(reservoirNode, "Process");

            // Figure out the initial value parameter for the Reservoir object.
            double initialValue = this.GetInitialValue(processNode);

            // Figure out the capacity-based parameters for the Reservoir object.

            // These are the default values for the parameters - if they are specified
            // in the XML file then use the values from the XML file instead.
            double capacity = Double.POSITIVE_INFINITY;
            Reservoir.OverflowBehavior overflowBehavior =
Reservoir.OverflowBehavior.Ignore;

            if (this.GetSingleChildNode(reservoirNode, "Capacity") != null) {
                String capacityAsString = this.GetSingleChildNode(reservoirNode,
"Capacity").getTextContent();
                logger.debug("Capacity (string) = " + capacityAsString);
                capacity = Double.parseDouble(capacityAsString);
```

```
                if (this.GetSingleChildNode(reservoirNode, "Capacity").hasAttributes()) {
                    String overflowBehaviorAsString =
this.GetSingleChildNode(reservoirNode,
"Capacity").getAttributes().getNamedItem("overflowBehavior").getNodeValue();
                    overflowBehavior = Enum.valueOf(Reservoir.OverflowBehavior.class,
overflowBehaviorAsString);
                }
                else {
                    overflowBehavior = Reservoir.OverflowBehavior.Error;
                }
            }
            else {
                // There is no Capacity node in the XML file so use the default values.
            }

            // Now that the XML parsing is done, instantiate the Reservoir object.
            Reservoir reservoir = new Reservoir(reservoirName, this.valueDatabase,
system, initialValue, capacity, overflowBehavior);
            system.AddReservoir(reservoirName, reservoir);

            this.AddEquation(reservoir, processNode);
        }
    }

    private void AddEquationOutflows(ProcessElement element, Node systemNode) throws
ModelInitializationException {
        List<Node> flowNodes = this.GetChildNodes(systemNode, "Outflow");
        for (int i = 0; i < flowNodes.size(); i++) {
            Node flowNode = flowNodes.get(i);
            String flowName =
flowNode.getAttributes().getNamedItem("name").getNodeValue();
            logger.debug("Initializing Outflow object " + flowName + ".");
            double initialValue = this.GetInitialValue(flowNode);

            EquationOutflow flow = new EquationOutflow(flowName, this.valueDatabase,
element, initialValue);
            element.AddFlow(element.GetName() + "." + flowName, flow);

            // Add the equation.
            this.AddEquation(flow, flowNode);
        }
    }

    private void AddCsvFileOutflows(ProcessElement element, Node systemNode) throws
            java.io.FileNotFoundException, java.io.IOException,
ModelInitializationException {
        List<Node> flowNodes = this.GetChildNodes(systemNode, "Outflow");
        for (int i = 0; i < flowNodes.size(); i++) {
            Node flowNode = flowNodes.get(i);
            String flowName =
flowNode.getAttributes().getNamedItem("name").getNodeValue();
            logger.debug("Initializing Outflow object " + flowName + ".");

            String filename = this.GetSingleChildNode(flowNode,
"Filename").getTextContent();

            // Figure out the column number parameter for the Outflow object.

            // This is the default value for the parameter - if it is specified
            // in the XML file then use the value from the XML file instead.
            int columnNumber = Constants.UnspecifiedCsvColumnNumber;
            if (this.GetSingleChildNode(flowNode, "ColumnNumber") != null) {
                columnNumber = Integer.parseInt(this.GetSingleChildNode(flowNode,
"ColumnNumber").getTextContent());
            }
            else {
                // Just use the default value set above.
            }
```

```
            CsvFileOutflow flow = new CsvFileOutflow(flowName, this.valueDatabase,
element, filename, columnNumber);
            element.AddFlow(element.GetName() + "." + flowName, flow);
        }
    }

    private void AddEquation(IEquationElement modelElement, Node systemNode) throws
ModelInitializationException {
        Node equationNode = this.GetSingleChildNode(systemNode, "Equation");
        modelElement.SetProcessEquation(equationNode.getTextContent());
    }

    private void AddModelSinks(Node modelNode) throws ModelInitializationException {
        // The DOM API has no way to request nodes with a specific name that are
        // immediate descendants only, so we have to walk the children ourselves
        // and pick out what we are interested in.
        for (int i = 0; i < modelNode.getChildNodes().getLength(); i++) {
            Node childNode = modelNode.getChildNodes().item(i);
            logger.debug("Found child node " + childNode.getNodeName() + " of the
Model.");
            if (childNode.getNodeName() == "Sink") {
                String sinkName =
childNode.getAttributes().getNamedItem("name").getNodeValue();
                logger.info("Found a model level Sink node - " + sinkName);

                // Figure out the initial value parameter for the Reservoir object.
                double initialValue = this.GetInitialValue(childNode);

                // Model level elements operate on the same time clock as the model
itself.
                Sink sink = new Sink(sinkName, this.valueDatabase, initialValue,
this.timeClocks.peek());

                List<Node> receiverList = this.GetChildNodes(childNode, "Receiver");
                for (int j = 0; j < receiverList.size(); j++) {

sink.AddReceiver(receiverList.get(j).getAttributes().getNamedItem("name").getNodeValue())
;
                }

                this.model.AddSink(sinkName, sink);
            }
        }
    }

    // Gets all nodes of a particular name that are an immediate child of the given node.
    private List<Node> GetChildNodes(Node parentNode, String elementName) {
        List<Node> list = new ArrayList<Node>();
        for (int i = 0; i < parentNode.getChildNodes().getLength(); i++) {
            Node childNode = parentNode.getChildNodes().item(i);
            logger.debug("Found child node " + childNode.getNodeName() + ".");
            if (childNode.getNodeName() == elementName) {
                if (childNode.getAttributes().getLength() > 0) {
                    if (childNode.getAttributes().getNamedItem("name") != null) {
                        String nodeName =
childNode.getAttributes().getNamedItem("name").getNodeValue();
                        logger.info("Found a " + elementName + " node - " + nodeName);
                    }
                }

                list.add(childNode);
            }
        }
        return list;
    }

    // Gets the one and only node of a particular name that is an immediate child of the
given node.
    // If there are no child nodes with the given name, the method will return null.
    // If there are more than 1 child nodes with the given name, the method will throw an
exception.
```

```
    private Node GetSingleChildNode(Node parentNode, String elementName) throws
ModelFormatException {
        List<Node> list = this.GetChildNodes(parentNode, elementName);
        if (list.size() == 1) {
            return list.get(0);
        } else if (list.size() == 0) {
            logger.debug("No child nodes named " + elementName + " under the given
node.");
            return null;
        } else { //if (list.size() > 1)
            String errorMessage = "More than one child nodes named " + elementName + "
under the given node.";
            logger.error(errorMessage);
            throw new ModelFormatException(errorMessage);
        }
    }

    private Document GetDocument(String modelFileName)
    throws IOException, SAXException, ParserConfigurationException {
        // DOM code from "Java in a Nutshell" book

        javax.xml.parsers.DocumentBuilderFactory documentFactory =
                javax.xml.parsers.DocumentBuilderFactory.newInstance();
        documentFactory.setValidating(false);
        javax.xml.parsers.DocumentBuilder parser = documentFactory.newDocumentBuilder();

        // set up error handlers.
        parser.setErrorHandler(new org.xml.sax.ErrorHandler() {
            public void error(SAXParseException exception) throws SAXException {
                logger.error("Failure parsing XML model.", exception);
                throw exception;
            }
            public void fatalError(SAXParseException exception) throws SAXException {
                logger.error("Failure parsing XML model.", exception);
                throw exception;
            }
            public void warning(SAXParseException exception) throws SAXException {
                logger.error("Failure parsing XML model.", exception);
                throw exception;
            }
        });

        return parser.parse(new File(modelFileName));
    }
}
```

EquationInterpreterException.java

```
/**
 * File: EquationInterpreterException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.Equations;

// Import statements
import DynSysMod.ValueComputationException;

/**
 * Indicates an exceptional circumstance while interpreting an equation.
 * @author Jennifer Leaf
 */
public class EquationInterpreterException extends ValueComputationException {

    /** Constructs a new exception with null as its detail message. */
    public EquationInterpreterException() {
```

```
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public EquationInterpreterException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public EquationInterpreterException(String message, Throwable cause) {
        super(message, cause);
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public EquationInterpreterException(Throwable cause) {
        super(cause);
    }

}
```

EquationsLexer.java
EquationsParser.java

These two files are omitted since they are autogenerated from the Equations.g file in Appendix E.

Interpreter.java

```
/**
 * File: Interpreter.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.Equations;

// Import statements
import DynSysMod.ModelElements.ModelElement;
import DynSysMod.ModelElements.ValueNotFoundException;

import java.math.BigDecimal;
import java.math.MathContext;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
import org.antlr.runtime.debug.*;

import org.apache.log4j.*;

/**
 * An interpreter for mathematical equations.  This interpreter merely calculates
 * values - it does not assign the value to a variable.  The caller is responsible
 * for saving or assigning the calculated value.
 *
 * If the reference to an Interpreter instance is held, the equation can be evaluated
 * repeatedly using the Evaluate() method without requiring the string to be re-parsed,
 * potentially increasing performance.
 *
 * The syntax of equations is defined in the Equations.g file.
 * The explanation of what mathematical constructs are supported is
 * defined in the Technical Specification document.
 * @author Jennifer Leaf
```

```
 */
public class Interpreter {

    /** The equation being interpreted. */
    private String equation;
    /** The parse tree of the equation. */
    private Tree parseTree;
    /** The Model element that owns the equation.  This object is used to get values
     * of other model elements, if the equation contains variables. */
    private ModelElement modelElement;
    /** The current time of the simulation.  It is possible for equations to use the
current
     * time as a variable. */
    private double time;

    private static Logger logger = Logger.getLogger(Interpreter.class.getName());

    /**
     * Creates a new instance of the Interpreter class. This constructor should be used
     * only if the equation has no variables.  This is primarily used for unit testing
     * the interpreter.
     * @param equation The equation to interpret - it should not contain an equals sign
or any variables besides t.
     */
    public Interpreter(String equation) {
        this.equation = equation;
        this.Initialize();
    }

    /**
     * Creates a new instance of the Interpreter class. This constructor should be used
     * during normal DynSysMod operations, as it allows the lookup of variable values
     * during its execution.
     * @param equation The equation to interpret - it should not contain an equals sign.
     * @param element The Model element that owns the equation (most likely an Outflow or
Reservoir).
     * @param time The current simulation time.
     */
    public Interpreter(String equation, ModelElement element, double time) {
        this.equation = equation;
        this.modelElement = element;
        this.time = time;
        this.Initialize();
    }

    /** Initializes the parse tree. */
    private void Initialize() {
        EquationsLexer lex = new EquationsLexer(new ANTLRStringStream(this.equation));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        ParseTreeBuilder builder = new ParseTreeBuilder("startRule");

        EquationsParser parser = new EquationsParser(tokens, builder);

        try {
            // The start rule of the grammar.
            parser.startRule();
            this.parseTree = builder.getTree();
        } catch (RecognitionException e)  {
            e.printStackTrace();
        }
    }

    /**
     * Evaluates the value of the mathematical expression that this interpreter holds.
     * @return The current value of the mathematical expression.
     */
    public BigDecimal Evaluate() throws EquationInterpreterException,
ValueNotFoundException {
        // The root node is "<grammar [grammarname]>" and is not part of the parse tree.
        // Just eat it and move on.
```

```
        return this.Evaluate(this.parseTree.getChild(0));
    }

    private BigDecimal Evaluate(Tree tree) throws EquationInterpreterException,
ValueNotFoundException {
        if (tree.getText() == "startRule") {
            // The startRule production generates a single child node.
            return this.Evaluate(tree.getChild(0));
        }
        else if (tree.getText() == "expr" || tree.getText() == "term") {
           return this.EvaluateArithmeticRule(tree);
        }
        else if (tree.getText() == "unary") {
            return this.EvaluateUnaryRule(tree);
        }
        else if (tree.getText() == "factor") {
            return this.EvaluateFactorRule(tree);
        }
        else {
            throw new EquationInterpreterException ("Unknown tree node " + tree.getText()
+ " found.");
        }
    }

    private BigDecimal EvaluateArithmeticRule(Tree tree) throws
EquationInterpreterException, ValueNotFoundException {
        // There will be either 1, or some other odd number of child nodes.
        if (tree.getChildCount() == 1) {
            // 1 child node - just evaluate it.
            return this.Evaluate(tree.getChild(0));
        }
        else if (tree.getChildCount() % 2 == 1) {
            // other odd # of child nodes - evaluate the odd numbered children.  the even
numbered children are operators.
            // Walk the children left to right and evaluate in order.  If one or more of
the children are in their own
            // parenthesized block to override operator precedence, they will be in their
own child node and the recursive
            // nature of this parser/interpreter will honor the parentheses.

            // The first term will also be used to accumulate the final value.
            BigDecimal firstTerm = this.Evaluate(tree.getChild(0));
            for (int i = 1; i < tree.getChildCount(); i += 2) {
                String operation = tree.getChild(i).getText();
                BigDecimal secondTerm = this.Evaluate(tree.getChild(i + 1));
                firstTerm = this.Compute(firstTerm, secondTerm, operation);
            }

            // We've evaluated all the children.
            return firstTerm;
        }
        else {
            throw new EquationInterpreterException("Unexpected number of children (" +
tree.getChildCount() + ") found at " + tree.getCharPositionInLine());
        }
    }

    private BigDecimal EvaluateFactorRule(Tree tree) throws EquationInterpreterException,
ValueNotFoundException {
        if (tree.getChildCount() == 1) {
            Tree child = tree.getChild(0);
            if (child.getText() == "number") {
                return
BigDecimal.valueOf(Double.parseDouble(child.getChild(0).getText()));
            }
            else if (child.getText() == "dt") {
                // Stop parsing the rest of this subtree - return the model element's
timestep.
                return BigDecimal.valueOf(this.modelElement.GetTimeStep());
            }
            else if (child.getText() == "t") {
```

```
                return BigDecimal.valueOf(this.time);
            }
            else if (child.getText() == "identifier") {
                // Simply, get the value of the identifer at the previous time step.
                // The default, most common use case of referencing another model element
is
                // to get its value at the previous time step.
                return this.modelElement.GetPreviousValue(child.getChild(0).getText(),
this.time);
            }
            else {
                throw new EquationInterpreterException("Unexpected child " +
child.getText() + " of factor node found.");
            }
        }
        else if (tree.getChildCount() == 3) {
            // The only production of "factor" that generates 3 children (two terminals
and a non-terminal)
            // is ( expr ).
            // For now, until proven otherwise, just eat the parens and return the middle
child's result.
            return this.Evaluate(tree.getChild(1));
        }
        else {
            throw new EquationInterpreterException("Found " + tree.getChildCount() + "
children of a factor rule - unexpected number.");
        }
    }

    private BigDecimal EvaluateUnaryRule(Tree tree) throws EquationInterpreterException,
ValueNotFoundException {
        if (tree.getChildCount() == 1) {
            return this.Evaluate(tree.getChild(0));
        }
        else if (tree.getChildCount() == 2) {
            // The first child node is the unary operator, and the second child node
            // is the operand that needs to be evaluated.
            String unaryOperator = tree.getChild(0).getText();
            BigDecimal value = this.Evaluate(tree.getChild(1));

            if (unaryOperator.equals("-")) {
                return value.negate();
            }
            else {
                // We found an unexpected value for the unary operator.
                throw new EquationInterpreterException("Unexpected unary operator " +
unaryOperator + " found.");
            }
        }
        else {
            // This isn't a legal sentence in the grammar.
            throw new EquationInterpreterException("Found " + tree.getChildCount() + "
children of a unary rule - unexpected number.");
        }
    }

    /**
     * Performs a binary mathematical operation.
     * @param firstTerm The first term of the operation.
     * @param secondTerm The second term of the operation.
     * @param operator The operator.  Currently supported: "+" "-" "*" "/"
     * @return The result of the operation.
     */
    private BigDecimal Compute(BigDecimal firstTerm, BigDecimal secondTerm, String
operator) throws EquationInterpreterException {

        if (operator.equals("+")) {
            return firstTerm.add(secondTerm, new MathContext(7));
        }
        else if (operator.equals("-")) {
            return firstTerm.subtract(secondTerm, new MathContext(7));
```

```
        }
        else if (operator.equals("*")) {
            return firstTerm.multiply(secondTerm, new MathContext(7));
        }
        else if (operator.equals("/")) {
            return firstTerm.divide(secondTerm, new MathContext(7));
        }
        else {
            throw new EquationInterpreterException("Unknown operator " + operator + ".");
        }
    }
}
```

SimpleTextLogger.java

```
/**
 * File: SimpleTextLogger.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.Logging;

// Import statements
import DynSysMod.ModelElements.ModelValueDatabase;

import java.io.*;
import java.util.*;

/**
 * Writes the contents of a ModelValueDatabase to a text file.
 * @author Jennifer Leaf
 */
public class SimpleTextLogger {

    /** Creates a new instance of the SimpleTextLogger class. */
    public SimpleTextLogger() {
    }

    /**
     * Saves a ModelValueDatabase to the given text file.  Uses a CSV (comma-separated
values) format.
     *
     * @param outputFileName The full path and filename of the file to write the contents
of the ModelValueDatabase to.
     * @param valueDatabase The ModelValueDatabase instance to save.
     */
    public void Save(String outputFileName, ModelValueDatabase valueDatabase) throws
IOException  {
        PrintWriter out = new PrintWriter(new BufferedWriter(new
FileWriter(outputFileName)));
        out.println(valueDatabase.ToCsvString());

        out.flush();
        out.close();
    }
}
```

CapacityOverflowException.java

```
/**
 * File: CapacityOverflowException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
```

```
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

/**
 * Indicates a Reservoir was filled above its capacity.
 * @author Jennifer Leaf
 */
public class CapacityOverflowException extends ValueComputationException {

    /** Constructs a new exception with null as its detail message. */
    public CapacityOverflowException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public CapacityOverflowException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public CapacityOverflowException(String message, Throwable cause) {
        super(message, cause);
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public CapacityOverflowException(Throwable cause) {
        super(cause);
    }

}
```

ColumnHeaderNotFound.java

```
/**
 * File: ColumnHeaderNotFoundException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

/**
 * Indicates a column header was not found in a file when one was expected.
 * @author Jennifer Leaf
 */
public class ColumnHeaderNotFoundException extends ModelInitializationException {

    /** Constructs a new exception with null as its detail message. */
    public ColumnHeaderNotFoundException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public ColumnHeaderNotFoundException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public ColumnHeaderNotFoundException(String message, Throwable cause) {
        super(message, cause);
```

```java
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public ColumnHeaderNotFoundException(Throwable cause) {
        super(cause);
    }

}
```

CsvFileOutflow.java

```java
/**
 * File: CsvFileOutflow.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import au.com.bytecode.opencsv.CSVReader;

import DynSysMod.Constants;
import DynSysMod.Equations.EquationInterpreterException;
import DynSysMod.ValueComputationException;

import java.io.*;
import java.math.BigDecimal;

/**
 * An Outflow whose value is determined by reading values from a CSV file.
 * Uses the 3rd party tool opencsv (http://opencsv.sourceforge.net/) for CSV
 * file operations.
 * A single CSV file can contain multiple values, where each column represents
 * a different flow.  The first line in the CSV file may contain column
 * headers, but it is not required to.  If the first line in the file contains
 * headers, then the column whose name matches the fully qualified name of
 * this flow is used.  If the first line in the file contains the first data
 * point, then the model must specify which column index to use.  If the model
 * specifies a column index but the CSV file contains a header row, the model's
 * column index is the setting that is used.
 * @author Jennifer Leaf
 */
public class CsvFileOutflow extends Outflow {

    /** The CSV file reader. */
    private CSVReader fileReader;
    /**
     * The column number from the CSV file that contains values for this flow.
     * The column index starts at 1.
     */
    private int columnNumber;

    /**
     * Creates a new instance of the CsvFileOutflow class.
     * @param fullyQualifiedName The fully qualified name of this flow.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param owner The Model element that owns this flow (where the flow
     *        originates from.
     * @param fileName The full path and filename of the CSV file to use.
     * @param columnNumber The column index to use.  If the model does not
     *        specify a column index, this class will assume the first line
     *        contains the column header names and this parameter should be set
     *        to Constants.UnspecifiedCsvColumnNumber.
     */
```

```
    public CsvFileOutflow(String fullyQualifiedName, ModelValueDatabase valueDatabase,
ProcessElement owner, String fileName, int columnNumber) throws
            java.io.FileNotFoundException, java.io.IOException,
ColumnHeaderNotFoundException {
        super(fullyQualifiedName, valueDatabase, owner);
        this.fileReader = new CSVReader(new FileReader(fileName));

        if (columnNumber == Constants.UnspecifiedCsvColumnNumber) {
            // This means the first row contains column header strings.  Find the index
of the column
            // that matches the fully qualified name of this flow, and store it for
later.
            String [] nextLine = this.fileReader.readNext();
            boolean isColumnFound = false;
            for (int i = 0; i < nextLine.length; i++) {
                if (!isColumnFound &&
nextLine[i].equals(this.GetFullyQualifiedName(this.GetName()))) {
                    this.columnNumber = i + 1;
                    isColumnFound = true;
                }
                else {
                    // Continue on to the next element, we haven't found the right
                    // column header yet.
                }
            }

            if (!isColumnFound) {
                // If we get here, we didn't find this flow's name in the column header
listing.
                throw new ColumnHeaderNotFoundException("Can't find column header for " +
this.GetFullyQualifiedName(this.GetName()) + ".");
            }
            else {
                // We had found a valid column number.
            }
        }
        else {
            this.columnNumber = columnNumber;
        }
    }

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public void Initialize() throws ValueComputationException {
        this.UpdateOutflows(0);
    }

    /**
     * Calculates and saves the value of the outflows in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateOutflows(double time) throws ValueComputationException {
         this.SetValue(this.GetFullyQualifiedName(this.GetName()), time,
this.ReadNextValue());
    }

    /**
     * Reads the next line from the CSV file and returns the value at the
     * correct index.
     * @return The next value in the CSV file.
     */
    private BigDecimal ReadNextValue() throws ValueComputationException {
         try {

             String [] nextLine = this.fileReader.readNext();
             if (nextLine != null) {
                 // the column number variable uses 1 as the first index, while the
                 // array's indexing starts at 0.
```

```
                return BigDecimal.valueOf(Double.parseDouble(nextLine[this.columnNumber
- 1]));
            } else {
                throw new ValueComputationException("No more values to read from the CSV
file.");
            }
        } catch (java.io.FileNotFoundException fnfEx) {
            throw new ValueComputationException(fnfEx);
        }
        catch (java.io.IOException ioEx) {
            throw new ValueComputationException(ioEx);
        }
    }
}
```

DuplicateElementException.java

```
/**
 * File: DuplicateElementException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

/**
 * Indicates a duplicate element is being added to the model.
 * @author Jennifer Leaf
 */
public class DuplicateElementException extends ModelInitializationException {

    /** Constructs a new exception with null as its detail message. */
    public DuplicateElementException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public DuplicateElementException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public DuplicateElementException(String message, Throwable cause) {
        super(message, cause);
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public DuplicateElementException(Throwable cause) {
        super(cause);
    }

}
```

DuplicateValueException.java

```
/**
 * File: DuplicateValueException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
```

```java
package DynSysMod.ModelElements;

import DynSysMod.ValueComputationException;

/**
 * Indicates a duplicate value is being added to the database.
 * @author Jennifer Leaf
 */
public class DuplicateValueException extends ValueComputationException {

    /** Constructs a new exception with null as its detail message. */
    public DuplicateValueException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public DuplicateValueException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public DuplicateValueException(String message, Throwable cause) {
        super(message, cause);
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public DuplicateValueException(Throwable cause) {
        super(cause);
    }

}
```

EquationOutflow.java

```java
/**
 * File: EquationOutflow.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.Equations.Interpreter;
import DynSysMod.ValueComputationException;

import java.math.BigDecimal;

/**
 * An Outflow whose value is determined by evaluating a mathematical function.
 * @author Jennifer Leaf
 */
public class EquationOutflow extends Outflow implements IEquationElement {

    /** The initial value. */
    private BigDecimal initialValue;
    /** The equation the element uses to update its state. */
    private String equation;

    /**
     * Creates a new instance of the EquationOutflow class.
     * @param fullyQualifiedName The fully qualified name of this flow.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param owner The Model element that owns this flow (where the flow
     *        originates from.
```

```
     * @param initialValue The initial value for this flow.
     */
    public EquationOutflow(String fullyQualifiedName, ModelValueDatabase valueDatabase,
ProcessElement owner, double initialValue) {
        super(fullyQualifiedName, valueDatabase, owner);
        this.initialValue = BigDecimal.valueOf(initialValue);
    }

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public void Initialize() throws ValueComputationException {
        this.SetValue(this.GetFullyQualifiedName(this.GetName()), 0, this.initialValue);
    }

    /**
     * Calculates and saves the value of the outflows in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateOutflows(double time) throws ValueComputationException {
        if (this.equation.indexOf('=') != -1) {
            // The part that actually computes the value starts immediately
            // after the equals sign.
            String calculation = equation.substring(equation.indexOf('=') + 1);
            // The part that contains the variable to set ends immediately
            // before the equals sign.
            String variableName = equation.substring(0, equation.indexOf('=') -
1).trim();

            this.SetValue(this.GetFullyQualifiedName(variableName), time, new
Interpreter(calculation, this, time).Evaluate());
        }
    }
    /**
     * Sets the equation the element uses to update its state.
     * @param equation The equation used by the element.
     */
    public void SetProcessEquation(String equation) {
        this.equation = equation;
    }
}
```

IEquationElement.java

```
/**
 * File: IEquationElement.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

/**
 * Defines elements that use equations to update their state.
 * @author Jennifer Leaf
 */
public interface IEquationElement {

    /**
     * Sets the equation the element uses to update its state.
     * @param equation The equation used by the element.
     */
    public void SetProcessEquation(String equation);
}
```

Model.java

```
/**
 * File: Model.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.Constants;
import DynSysMod.ValueComputationException;

import java.lang.*;
import java.util.*;

import org.apache.log4j.*;

/**
 * Represents a Model to be simulated.
 * @author Jennifer Leaf
 */
public class Model {

    /** The Sources of the Model. */
    private Hashtable<String, Source> sources;
    /** The Systems of the Model. */
    private Hashtable<String, System> systems;
    /** The Sinks of the Model. */
    private Hashtable<String, Sink> sinks;
    /** The name of the model. */
    private String name;
    /** The ModelValueDatabase instance for the model. */
    private ModelValueDatabase valueDatabase;
    /** The time interval to update the Model at. */
    private TimeSpan timeClock;

    private static Logger logger = Logger.getLogger(Model.class.getName());

    /**
     * Creates a new instance of the Model class.
     * @param name The fully qualified name of this element.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param timeClock The time interval to update this element at.
     */
    public Model(String name, ModelValueDatabase valueDatabase, TimeSpan timeClock) {
        this.name = name;
        this.sources = new Hashtable<String, Source>();
        this.systems = new Hashtable<String, System>();
        this.sinks = new Hashtable<String, Sink>();
        this.valueDatabase = valueDatabase;
        this.timeClock = timeClock;
    }

    /**
     * Adds a Source to this Model.
     * @param sourceName The name of the Source.
     * @param sourceObject The Source instance.
     */
    public void AddSource(String sourceName, Source sourceObject) throws
DuplicateElementException {
        if (!this.sources.containsKey(sourceName)) {
            this.sources.put(sourceName, sourceObject);
        }
        else {
            String errorMessage = "Cannot add duplicate Source " + sourceName + ".";
            logger.error(errorMessage);
```

```
            throw new DuplicateElementException(errorMessage);
        }
    }

    /**
     * Adds a System to this Model.
     * @param systemName The name of the System.
     * @param systemObject The System instance.
     */
    public void AddSystem(String systemName, System systemObject) throws
DuplicateElementException {
        if (!this.systems.containsKey(systemName)) {
            this.systems.put(systemName, systemObject);
        }
        else {
            String errorMessage = "Cannot add duplicate System " + systemName + ".";
            logger.error(errorMessage);
            throw new DuplicateElementException(errorMessage);
        }
    }

    /**
     * Adds a Sink to this Model.
     * @param sinkName The name of the Sink.
     * @param sinkObject The Sink instance.
     */
    public void AddSink(String sinkName, Sink sinkObject) throws
DuplicateElementException {
        if (!this.sinks.containsKey(sinkName)) {
            this.sinks.put(sinkName, sinkObject);
        }
        else {
            String errorMessage = "Cannot add duplicate Sink " + sinkName + ".";
            logger.error(errorMessage);
            throw new DuplicateElementException(errorMessage);
        }
    }

    /**
     * Gets the Model elements that are within this Model instance.  This
     * includes Sources, Systems, and Sinks.
     * @return The Model elements that are within this Model instance.
     */
    public Hashtable<String, ModelElement> GetModelElements() {
        Hashtable<String, ModelElement> elements = new Hashtable<String, ModelElement>();
        elements.putAll(this.sources);
        elements.putAll(this.systems);
        elements.putAll(this.sinks);
        return elements;
    }

    /**
     * Gets the ModelValueDatabase instance for the model.
     * @return The ModelValueDatabase instance for the model.
     */
    public ModelValueDatabase GetValueDatabase() {
        return this.valueDatabase;
    }

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public void Initialize() throws ValueComputationException {
        this.valueDatabase.Reset();

        for (Source source : this.sources.values()) {
            source.Initialize();
        }

        for (System system : this.systems.values()) {
            system.Initialize();
```

```
        }

        for (Sink sink : this.sinks.values()) {
            sink.Initialize();
        }
    }

    /**
     * Gets the time interval value for the model.
     * @return The time interval value for the model.
     */
    public TimeSpan GetTimeClock() {
        return this.timeClock;
    }
}
```

ModelElement.java

```
/**
 * File: ModelElement.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 * - What if there is no value for the element at the given time?  In other words, the
 * caller asks for a value at a previous time, but that time exactly doesn't exist.
 * For example, we have values for t = 3 and t = 4, but the caller asked for t = 3.5.
 * Do we round up (to the future) to the nearest time where we do have a value?  Do
 * we throw an exception?  What if the caller requests a time in the future thus there
 * is no data for the given time?
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

import java.math.BigDecimal;
import java.util.*;

import org.apache.log4j.*;

/**
 * Represents an element that can be added to a Model.
 * @author Jennifer Leaf
 */
public abstract class ModelElement {

    /** The name of the element. */
    private String name;
    /** The ModelValueDatabase instance for the model. */
    private ModelValueDatabase valueDatabase;

    private static Logger logger = Logger.getLogger(ModelElement.class.getName());

    /**
     * Creates a new instance of the ModelElement class.
     * @param name The fully qualified name of this element.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     */
    protected ModelElement(String name, ModelValueDatabase valueDatabase) {
        this.name = name;
        this.valueDatabase = valueDatabase;
    }

    /**
     * Gets the name of this element.  This may or may not be the same as the
     * fully qualified name.
     * @return The name of this element.
```

```
     */
    public String GetName() {
        return this.name;
    }

    /**
     * Gets the short name of this element.  This strips any system prefixes off
     * of the name and returns just the last section of the name.
     * @param fullyQualifiedName The fully qualified name of this element.
     * @return The short name of this element.
     */
    public String GetShortName(String fullyQualifiedName) {
        int indexOfLastSeparator =
fullyQualifiedName.lastIndexOf(DynSysMod.Constants.NameSeparator);
        return fullyQualifiedName.substring(indexOfLastSeparator + 1);
    }

    /**
     * Gets the fully qualified name of this Model element.  Refer to the technical
     * specifications document for a definition of "fully qualified name". This
     * method can determine whether the given name is already fully qualified
     * and adjust the name appropriately.
     * @param name The short or fully qualified name of this element.
     * @return The fully qualified name.
     */
    public abstract String GetFullyQualifiedName(String name);

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public abstract void Initialize() throws ValueComputationException;

    /**
     * Gets the time interval that this Model element should be recalculated at.
     * @return The time interval for this Model element.
     */
    public abstract double GetTimeStep();

    /**
     * Calculates and saves the value of the outflows in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public abstract void UpdateOutflows(double time) throws ValueComputationException;

    /**
     * Calculates and saves the value of the reservoirs in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public abstract void UpdateReservoirs(double time) throws ValueComputationException;

    /**
     * Gets the value of the element with the given name at the given time.
     * @param fullyQualifiedName The fully qualified name of the element.
     * @param time The simulation time to get the value for.
     * @return The value of the element at the given time.
     */
    public BigDecimal GetValue(String fullyQualifiedName, double time) throws
ValueNotFoundException {
        // There is a chance that the name passed in doesn't exist in the symbol
        // table.  If this is the case, figure out whether prepending the enclosing
        // system name will give us a valid value.
        if (this.valueDatabase.Contains(fullyQualifiedName)) {
            return this.valueDatabase.GetValue(fullyQualifiedName, time);
        } else if
(this.valueDatabase.Contains(this.GetFullyQualifiedName(fullyQualifiedName))) {
            return
this.valueDatabase.GetValue(this.GetFullyQualifiedName(fullyQualifiedName), time);
        } else {
```

```
            String errorMessage = "Cannot get the value of " + fullyQualifiedName + " at
time = " + time + " - not found in the database.";
            logger.error(errorMessage);
            throw new ValueNotFoundException(errorMessage);
        }
    }

    /**
     * Gets the value of the element with the given name at the previous time interval.
     * @param fullyQualifiedName The fully qualified name of the element.
     * @param time The simulation time to get the value for.
     * @return The value of the element at the previous time interval.
     */
    public BigDecimal GetPreviousValue(String fullyQualifiedName, double time) throws
ValueNotFoundException {
        // TODO - what if I don't have data for every interval?  Q6 in the
specifications.
        return this.GetValue(fullyQualifiedName, time - this.GetTimeStep());
    }

    /**
     * Sets the value of the element with the given name at the given time.
     * @param fullyQualifiedName The fully qualified name of the element.
     * @param time The simulation time to set the value for.
     * @param value The value of the element.
     */
    public void SetValue(String fullyQualifiedName, double time, BigDecimal value) throws
DuplicateValueException, CapacityOverflowException {
        this.valueDatabase.SetValue(fullyQualifiedName, time, value);
    }
}
```

ModelFormatException.java

```
/**
 * File: ModelFormatException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

/**
 * Indicates a syntax error in the model definition.
 * @author Jennifer Leaf
 */
public class ModelFormatException extends ModelInitializationException {

    /** Constructs a new exception with null as its detail message. */
    public ModelFormatException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public ModelFormatException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public ModelFormatException(String message, Throwable cause) {
        super(message, cause);
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public ModelFormatException(Throwable cause) {
```

```
        super(cause);
    }

}
```

ModelInitializationException.java

```
/**
 * File: ModelInitializationException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

/**
 * Indicates a error when initializing the model to its initial state during a simulation
run.
 * @author Jennifer Leaf
 */
public class ModelInitializationException extends ValueComputationException {

    /** Constructs a new exception with null as its detail message. */
    public ModelInitializationException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public ModelInitializationException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public ModelInitializationException(String message, Throwable cause) {
        super(message, cause);
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public ModelInitializationException(Throwable cause) {
        super(cause);
    }

}
```

ModelValueDatabase.java

```
/**
 * File: ModelValueDatabase.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 * - The simple implementation of this class was to dump the values into an in-
 * memory data structure, and serialize it to a CSV file after the simulation
 * run.  For scalability and flexibility in implementing the data analysis
 * components, it may be prudent to replace the internal implementation with a
 * connection to an actual database, or at least stream the values to file at
 * some interval during runtime.  Otherwise the potential exists for an instance
 * of this class to consume a considerable amount of memory during runtime.  The
 * other classes in DynSysMod will not be impacted by this internal change as
 * long as the public interface for this class is left unchanged.
```

```
 * - What if there is no value for the element at the given time?  In other words, the
 * caller asks for a value at a previous time, but that time exactly doesn't exist.
 * For example, we have values for t = 3 and t = 4, but the caller asked for t = 3.5.
 * Do we round up (to the future) to the nearest time where we do have a value?  Do
 * we throw an exception?  What if the caller requests a time in the future thus there
 * is no data for the given time?
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import java.math.BigDecimal;
import java.util.*;

import org.apache.log4j.*;

/**
 * Represents a database of values for all Model elements, for a single Model
 * during a single simulation run.  The Model instance "owns" the database, and
 * this database instance should be shared across all Model elements that are
 * part of the Model instance.
 * @author Jennifer Leaf
 */
public class ModelValueDatabase {

    /**
     * Holds the values of each element.
     * key = name of model element
     * value = dictionary of time/value pairs
     *         key = time t
     *         value = value at time t
     */
    private LinkedHashMap<String, LinkedHashMap<Double, BigDecimal>> dataTable;

    private static Logger logger = Logger.getLogger(ModelValueDatabase.class.getName());

    /**
     * Creates a new instance of the ModelValueDatabase class.
     */
    public ModelValueDatabase() {
        this.dataTable = new LinkedHashMap<String, LinkedHashMap<Double, BigDecimal>>();
    }

    /**
     * Gets the value of the element with the given name at the given time.
     * @param fullyQualifiedName The fully qualified name of the element.
     * @param time The simulation time to get the value for.
     * @return The value of the element at the given time.
     */
    public BigDecimal GetValue(String fullyQualifiedName, double time) {
        // See TODO section above for a note about what to do if the requested
        // value doesn't exist in the database.  Most relevant when different
        // systems can have different time clocks.
        return this.dataTable.get(fullyQualifiedName).get(time);
    }

    /**
     * Sets the value of the element with the given name at the given time.
     * @param fullyQualifiedName The fully qualified name of the element.
     * @param time The simulation time to set the value for.
     * @param value The value of the element.
     */
    public void SetValue(String fullyQualifiedName, double time, BigDecimal value) throws
DuplicateValueException {
        // Assume we will never be asked to enter multiple values for any given
        // fullyQualifiedName/time pair.  Can't think of a situation where we would
        // want or need to recalculate a value we already calculated.
        if (!this.dataTable.containsKey(fullyQualifiedName)) {
            this.dataTable.put(fullyQualifiedName, new LinkedHashMap<Double,
BigDecimal>());
```

```
        }
        // else the fullyQualifiedName is already in the data table.

        if (this.dataTable.get(fullyQualifiedName).containsKey(time)) {
            throw new DuplicateValueException("Cannot enter more than one value for " +
fullyQualifiedName + " at time = " + time + ".");
        }

        logger.debug("Setting model element " + fullyQualifiedName + " to value " +
value.toString() + " at time = " + time + ".");
        this.dataTable.get(fullyQualifiedName).put(time, value);
    }

    /**
     * Resets the database to its initial state, i.e. with no saved values.
     */
    public void Reset() {
        this.dataTable.clear();
    }

    /**
     * Serializes the contents of this database to a CSV formatted string.
     * Each csv line is formatted:   elementName,time,value
     * followed by a newline.
     * @return The contents of this database as a CSV formatted string.
     */
    public String ToCsvString() {
        StringBuilder csvString = new StringBuilder();
        for(Map.Entry<String, LinkedHashMap<Double, BigDecimal>> element :
this.dataTable.entrySet()) {
            for (Map.Entry<Double, BigDecimal> entry : element.getValue().entrySet()) {
                csvString.append(element.getKey() + "," + entry.getKey() + "," +
entry.getValue() + java.lang.System.getProperty("line.separator"));
            }
        }
        return csvString.toString();
    }

    /**
     * Determines whether this database contains at least 1 entry for the
     * Model element with the given name.
     * @param name The Model element to lookup.
     * @return true if there is at least 1 entry for the Model element with the
     *         given name, false otherwise.
     */
    public boolean Contains(String name) {
        return this.dataTable.containsKey(name);
    }
}
```

Outflow.java

```
/**
 * File: Outflow.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.Equations.EquationInterpreterException;

/**
 * Represents a flow between two Model elements.  An outflow corresponds to
 * "rates" or "flows" in systems dynamics terminology.  It defines the rate
 * of change of a value in a simulated model.
```

```
 * @author Jennifer Leaf
 */
public abstract class Outflow extends ModelElement {

    /**
     * The Model element that owns this flow (where the flow originates from.
     * It can either be a Source or System.
     */
    protected ProcessElement owner;

    /**
     * Creates a new instance of the Outflow class.
     * @param fullyQualifiedName The fully qualified name of this flow.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param owner The Model element that owns this flow (where the flow
     *        originates from.
     */
    protected Outflow(String fullyQualifiedName, ModelValueDatabase valueDatabase,
ProcessElement owner) {
        super(fullyQualifiedName, valueDatabase);
        this.owner = owner;
    }

    /**
     * Gets the time interval that this Model element should be recalculated at.
     * @return The time interval for this Model element.
     */
    public double GetTimeStep() {
        return this.owner.GetTimeStep();
    }

    /**
     * Gets the fully qualified name of this Model element.  Refer to the technical
     * specifications document for a definition of "fully qualified name". This
     * method can determine whether the given name is already fully qualified
     * and adjust the name appropriately.
     * @param name The short or fully qualified name of this element.
     * @return The fully qualified name.
     */
    public String GetFullyQualifiedName(String name) {
        if (name.indexOf(DynSysMod.Constants.NameSeparator) == -1) {
            return this.owner.GetName() + DynSysMod.Constants.NameSeparator + name;
        }
        else {
            // The name is already fully qualified.
            return name;
        }
    }

    /**
     * Calculates and saves the value of the reservoirs in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateReservoirs(double time) throws EquationInterpreterException,
DuplicateValueException {
        // Outflows have no reservoirs, so do nothing.
        // Defining this method here prevents the subclasses from needing to
        // create this empty method.
    }
}
```

ProcessElement.java

```
/**
 * File: ProcessElement.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
```

```
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

import java.util.*;

import org.apache.log4j.*;

/**
 * Represents a Model element that contains a Process.  That is, a Model element
 * that calculates and updates one or more flows through the Model.
 * @author Jennifer Leaf
 */
public abstract class ProcessElement extends ModelElement {

    /** The flows out of this element. */
    private Hashtable<String, Outflow> flows;
    /** The time interval to update this element at. */
    private TimeSpan timeClock;

    private static Logger logger = Logger.getLogger(ProcessElement.class.getName());

    /**
     * Creates a new instance of the ProcessElement class.
     *
     * @param name The name of the element.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param timeSpan The time interval to update this element at.
     */
    protected ProcessElement(String name, ModelValueDatabase valueDatabase, TimeSpan
timeSpan) {
        super(name, valueDatabase);
        this.flows = new Hashtable<String, Outflow>();
        this.timeClock = timeSpan;
    }

    /**
     * Gets the time interval that this Model element should be recalculated at.
     * @return The time interval for this Model element.
     */
    public double GetTimeStep() {
        return this.timeClock.GetValue();
    }

    /**
     * Adds an Outflow to this element.
     * @param flowName The name of the Outflow.
     * @param flowObject The Outflow instance.
     */
    public void AddFlow(String flowName, Outflow flowObject) throws
DuplicateElementException {
        if (!this.flows.containsKey(flowName)) {
            flows.put(flowName, flowObject);
        }
        else {
            String errorMessage = "Cannot add duplicate Outflow " + flowName + ".";
            logger.error(errorMessage);
            throw new DuplicateElementException(errorMessage);
        }
    }

    /**
     * Calculates and saves the value of the outflows in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateOutflows(double time) throws ValueComputationException {
```

```java
        for (Map.Entry<String, Outflow> entry : this.flows.entrySet()) {
            logger.debug("Updating model element " + entry.getKey() + " at time = " +
time + ".");
            entry.getValue().UpdateOutflows(time);
        }
    }

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public void Initialize() throws ValueComputationException {
        for (Map.Entry<String, Outflow> flow : this.flows.entrySet()) {
            flow.getValue().Initialize();
        }
    }
}
```

Reservoir.java

```java
/**
 * File: Reservoir.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.Equations.Interpreter;
import DynSysMod.ValueComputationException;

import java.math.BigDecimal;

/**
 * Represents a Reservoir in a Model.  A reservoir corresponds to "stocks" or
 * "levels" in systems dynamics terminology.
 * @author Jennifer Leaf
 */
public class Reservoir extends ModelElement implements IEquationElement {

    /**
     * The action that a Reservoir should take, if any, if the value of the
     * Reservoir overflows its capacity during a simulation.
     */
    public enum OverflowBehavior {
        /**
         * Raise an exception when the capacity is overflowed and do not
         * continue processing.
         */
        Error,
        /**
         * Quantities over the capacity are discarded, and the value of the reservoir is
set
         * to be equal to the capacity.
         **/
        DiscardOverflow,
        /**
         * Ignore the Capacity constraint and allow the value of the Reservoir to be set
         * to a value greater than its capacity.
         */
        Ignore
    }

    /** The initial value of the Reservoir. */
    private BigDecimal initialValue;
    /** The unit of measurement for the Reservoir.  Currently unused. */
    private String unit;
```

```
    /** The maximum capacity of the Reservoir. */
    private double capacity;
    /** The action to take if the Reservoir overflows its capacity. */
    private OverflowBehavior overflowBehavior;
    /** The equation the element uses to update its state. */
    private String equation;
    /**
     * The Model element that owns this reservoir.  It can either be a System
     * or Sink.
     */
    private ModelElement owner;

    /**
     * Creates a new instance of the Reservoir class.
     * @param name The name of this Reservoir.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param owner The Model element that owns this Reservoir.
     * @param initialValue The initial value of this Reservoir.
     * @param capacity The maximum capacity of this Reservoir.
     * @param overflowBehavior The action to take if the Reservoir overflows
     *        its capacity.
     */
    public Reservoir(String name, ModelValueDatabase valueDatabase, ModelElement owner,
double initialValue,
            double capacity, OverflowBehavior overflowBehavior) {
        super(name, valueDatabase);
        this.initialValue = BigDecimal.valueOf(initialValue);
        this.capacity = capacity;
        this.overflowBehavior = overflowBehavior;
        this.owner = owner;
    }

    /**
     * Sets the equation the element uses to update its state.
     * @param equation The equation used by the element.
     */
    public void SetProcessEquation(String equation) {
        this.equation = equation;
    }

    /**
     * Calculates and saves the value of the reservoirs in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateReservoirs(double time) throws ValueComputationException {
        if (this.equation.indexOf('=') != -1) {
            // The part that actually computes the value starts immediately
            // after the equals sign.
            String calculation = equation.substring(equation.indexOf('=') + 1);
            // The part that contains the variable to set ends immediately
            // before the equals sign.
            String variableName = equation.substring(0, equation.indexOf('=') -
1).trim();

            BigDecimal value = new Interpreter(calculation, this, time).Evaluate();

            if (this.capacity < value.doubleValue()) {
                switch (this.overflowBehavior) {
                    case DiscardOverflow:
                        this.SetValue(this.GetFullyQualifiedName(variableName), time,
BigDecimal.valueOf(this.capacity));
                        break;
                    case Error:
                        throw new CapacityOverflowException ("Reservoir capacity overflow
- capacity of reservoir " +
                                this.GetFullyQualifiedName(variableName) + " is " +
this.capacity +
                                ", but trying to set the value to " + value + ".");
                    case Ignore:
```

```
                        this.SetValue(this.GetFullyQualifiedName(variableName), time,
value);
                        break;
                }
                return;
            }
            else {
                // The Reservoir's new value is less than its capacity,
                // so just continue on.
                this.SetValue(this.GetFullyQualifiedName(variableName), time, value);
            }
        }
    }

    /**
     * Calculates and saves the value of the outflows in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateOutflows(double time) {
        // Reservoirs have no outflows, so do nothing.
    }

    /**
     * Gets the time interval that this Model element should be recalculated at.
     * @return The time interval for this Model element.
     */
    public double GetTimeStep() {
        return this.owner.GetTimeStep();
    }

    /**
     * Gets the fully qualified name of this Model element.  Refer to the technical
     * specifications document for a definition of "fully qualified name". This
     * method can determine whether the given name is already fully qualified
     * and adjust the name appropriately.
     * @param name The short or fully qualified name of this element.
     * @return The fully qualified name.
     */
    public String GetFullyQualifiedName(String name) {
        if (name.indexOf(DynSysMod.Constants.NameSeparator) == -1) {
            return this.owner.GetName() + DynSysMod.Constants.NameSeparator + name;
        }
        else {
            // The name is already fully qualified.
            return name;
        }
    }

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public void Initialize() throws ValueComputationException {
        this.SetValue(this.GetFullyQualifiedName(this.GetName()), 0, this.initialValue);
    }
}
```

Sink.java

```
/**
 * File: Sink.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 * - The current implementation of Sink is to act as an infinite capacity
 * Reservoir.  Once data flows are implemented, the Sink could be modified to
 * send a notification to the Model elements that flow into it that the Sink is
 * full, causing a "backup" of the flows within the Model for more realistic
 * behavior.
```

```java
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

import java.math.BigDecimal;
import java.util.*;

/**
 * Represents a Model Sink.  The values that flow into the Sinks are not cycled
 * back into another Model element as an input.
 * @author Jennifer Leaf
 */
public class Sink extends ModelElement {

    /**
     * The Reservoir for the Sink value.  Sinks are treated as a System with a
     * single, unnamed Reservoir of infinite capacity.
     */
    private Reservoir reservoir;
    /** The list of named Outflows that flow into this Sink. */
    private ArrayList<String> receiverList;
    /** The time interval to update the Sink's values on. */
    private TimeSpan timeClock;

    /**
     * Creates a new instance of the Sink class.
     * @param fullyQualifiedName The fully qualified name of this Sink.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param initialValue The initial value of this Sink.
     * @param timeSpan The time interval to update this element at.
     */
    public Sink(String fullyQualifiedName, ModelValueDatabase valueDatabase, double
initialValue, TimeSpan timeSpan) {
        super(fullyQualifiedName, valueDatabase);
        this.reservoir = new Reservoir(fullyQualifiedName, valueDatabase, this,
initialValue, Double.POSITIVE_INFINITY, Reservoir.OverflowBehavior.Ignore);
        this.receiverList = new ArrayList<String>();
        this.timeClock = timeSpan;
    }

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public void Initialize() throws ValueComputationException {
        this.reservoir.Initialize();
    }

    /**
     * Gets the fully qualified name of this Model element.  Refer to the technical
     * specifications document for a definition of "fully qualified name". This
     * method can determine whether the given name is already fully qualified
     * and adjust the name appropriately.
     * @param name The short or fully qualified name of this element.
     * @return The fully qualified name.
     */
    public String GetFullyQualifiedName(String name) {
        // Model Sinks require no qualification.
        return name;
    }

    /**
     * Gets the time interval that this Model element should be recalculated at.
     * @return The time interval for this Model element.
     */
    public double GetTimeStep() {
        return this.timeClock.GetValue();
    }
```

```java
    /**
     * Calculates and saves the value of the reservoirs in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateReservoirs(double time) throws ValueComputationException {
        BigDecimal currentSum = new BigDecimal(0.0);
        for(String receiverName : this.receiverList) {
            currentSum = currentSum.add(this.GetPreviousValue(receiverName, time));
        }

        this.reservoir.SetValue(reservoir.GetFullyQualifiedName(reservoir.GetName()),
time, currentSum);
    }

    /**
     * Calculates and saves the value of the outflows in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateOutflows(double time) throws ValueComputationException {
        // Sinks have no outflows, so do nothing.
    }

    /**
     * Adds an named Outflow as an input to this Sink.
     * @param fullyQualifiedName The fully qualified name of the Outflow that
     *        flows into this Sink.
     */
    public void AddReceiver(String fullyQualifiedName) {
        this.receiverList.add(fullyQualifiedName);
    }
}
```

Source.java

```java
/**
 * File: Source.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 * - Not sure how to handle time steps in exogenous sources.  Are they updated in
 * sync with the Model time clock? Q2 in the specifications.
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

/**
 * Represents an exogenous source to a Model.
 * @author Jennifer Leaf
 */
public class Source extends ProcessElement {

    /**
     * Creates a new instance of the Source class.
     * @param fullyQualifiedName The fully qualified name of this Source.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param timeSpan The time interval to update this element at.
     */
    public Source(String fullyQualifiedName, ModelValueDatabase valueDatabase, TimeSpan
timeSpan) {
        super(fullyQualifiedName, valueDatabase, timeSpan);
    }
```

```
    /**
     * Gets the fully qualified name of this Model element.  Refer to the technical
     * specifications document for a definition of "fully qualified name". This
     * method can determine whether the given name is already fully qualified
     * and adjust the name appropriately.
     * @param name The short or fully qualified name of this element.
     * @return The fully qualified name.
     */
    public String GetFullyQualifiedName(String name) {
        // Exogenous Sources require no qualification.
        return name;
    }

    /**
     * Calculates and saves the value of the reservoirs in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateReservoirs(double time) throws ValueComputationException {
        // Nothing to do here, so override the base class behavior.
    }
}
```

System.java

```
/**
 * File: System.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

import java.util.*;

import org.apache.log4j.*;

/**
 * Represents a System in the model.  Systems aggregate flows, reservoirs, and
 * the processes used to update them into a single element.
 * @author Jennifer Leaf
 */
public class System extends ProcessElement {

    private static Logger logger = Logger.getLogger(System.class.getName());

    /** The Reservoirs that are within this System. */
    private Hashtable<String, Reservoir> reservoirs;

    /**
     * Creates a new instance of the System class.
     * @param fullyQualifiedName The fully qualified name of this System.
     * @param valueDatabase The ModelValueDatabase instance for the model.
     * @param timeSpan The time interval to update this element at.
     */
    public System(String fullyQualifiedName, ModelValueDatabase valueDatabase, TimeSpan
timeSpan) {
        super(fullyQualifiedName, valueDatabase, timeSpan);
        this.reservoirs = new Hashtable<String, Reservoir>();
    }

    /**
     * Adds a Reservoir to this System.
     * @param reservoirName The name of the Reservoir.
```

```
     * @param reservoirObject The Reservoir instance.
     */
    public void AddReservoir(String reservoirName, Reservoir reservoirObject) throws
DuplicateElementException {
        if (!this.reservoirs.containsKey(reservoirName)) {
            reservoirs.put(reservoirName, reservoirObject);
        }
        else {
            String errorMessage = "Cannot add duplicate Reservoir " + reservoirName +
".";
            logger.error(errorMessage);
            throw new DuplicateElementException(errorMessage);
        }
    }

    /**
     * Initializes this model element for a simulation time t = 0.
     */
    public void Initialize() throws ValueComputationException {
        // force model elements managed by base class to be initialized.
        super.Initialize();

        for (Map.Entry<String, Reservoir> reservoir : this.reservoirs.entrySet()) {
            reservoir.getValue().Initialize();
        }
    }

    /**
     * Gets the fully qualified name of this Model element.  Refer to the technical
     * specifications document for a definition of "fully qualified name". This
     * method can determine whether the given name is already fully qualified
     * and adjust the name appropriately.
     * @param name The short or fully qualified name of this element.
     * @return The fully qualified name.
     */
    public String GetFullyQualifiedName(String name) {
        if (name.indexOf(DynSysMod.Constants.NameSeparator) == -1) {
            return this.GetName() + DynSysMod.Constants.NameSeparator + name;
        }
        else {
            // The name is already fully qualified.
            return name;
        }
    }

    /**
     * Calculates and saves the value of the reservoirs in this Model element for
     * the given time.
     * @param time The current simulation time.
     */
    public void UpdateReservoirs(double time) throws ValueComputationException {
        for (Map.Entry<String, Reservoir> entry : this.reservoirs.entrySet()) {
            logger.debug("In System " + this.GetName() + " - updating Reservoir " +
entry.getKey() + " at time = " + time + ".");
            entry.getValue().UpdateReservoirs(time);
        }
    }
}
```

TimeSpan.java

```
/**
 * File: TimeSpan.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 * - Represent times using a more sophisticated representation than a double
 * and a string to represent the unit.  I recommend researching a 3rd party
 * library such as JScience, rather than rolling our own time interval class.
```

```
 */

// Package declaration
package DynSysMod.ModelElements;

/**
 * Represents a time interval.
 * @author Jennifer Leaf
 */
public class TimeSpan {

    /** The value of the time interval. */
    private double value;
    /** The unit of measurement of the time interval. */
    private String unit;

    /**
     * Creates a new instance of the TimeSpan class.
     * @param value The value of the time interval.
     * @param unit The unit of measurement of the time interval.
     */
    public TimeSpan(double value, String unit) {
        this.value = value;
        this.unit = unit;
    }

    /**
     * Gets the value of the time interval.
     * @return The value of the time interval.
     */
    public double GetValue() {
        return this.value;
    }
}
```

ValueNotFoundException.java

```
/**
 * File: ValueNotFoundException.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.ModelElements;

// Import statements
import DynSysMod.ValueComputationException;

/**
 * Indicates a value was not found in the database when one was expected.
 * @author Jennifer Leaf
 */
public class ValueNotFoundException extends ValueComputationException {

    /** Constructs a new exception with null as its detail message. */
    public ValueNotFoundException() {
        super();
    }

    /** Constructs a new exception with the specified detail message. */
    public ValueNotFoundException(String message) {
        super(message);
    }

    /** Constructs a new exception with the specified detail message and cause. */
    public ValueNotFoundException(String message, Throwable cause) {
        super(message, cause);
```

```
    }

    /** Constructs a new exception with the specified cause and a detail message of
     * (cause==null ? null : cause.toString()) (which typically contains the class
     * and detail message of cause). */
    public ValueNotFoundException(Throwable cause) {
        super(cause);
    }
}
```

LinearScheduler.java

```
/**
 * File: LinearScheduler.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.SimulationEngine;

// Import statements
import DynSysMod.ModelElements.Model;
import DynSysMod.ModelElements.ModelElement;
import DynSysMod.ValueComputationException;

import java.util.*;

import org.apache.log4j.*;

/**
 * A simple scheduler that updates all Reservoirs and Outflows in the Model at
 * every time step.  This class can be used if all time clocks in the model are
 * the same - that is, no System overrides the Model-level clock.
 * @author Jennifer Leaf
 */
public class LinearScheduler extends Scheduler {

    private static Logger logger = Logger.getLogger(LinearScheduler.class.getName());

    /**
     * Creates a new instance of the LinearScheduler class.
     * @param model The model to use.
     */
    public LinearScheduler(Model model) {
        super(model);
    }

    /**
     * Calculates all elements that require updating at the given time.
     * @param time The current value of the simulation clock.
     **/
    public void CalculateAllElements(double time) throws ValueComputationException {
        this.UpdateReservoirs(time);
        this.UpdateOutflows(time);
    }

    /**
     * Updates all Reservoirs in the model.
     * @param time The current value of the simulation clock.
     */
    private void UpdateReservoirs(double time) throws ValueComputationException {
        logger.debug("Updating Model Reservoirs at time = " + time + ".");
        for (Map.Entry<String, ModelElement> entry : this.modelElements.entrySet()) {
            logger.debug("Updating model element " + entry.getKey() + " at time = " +
time + ".");
            entry.getValue().UpdateReservoirs(time);
        }
```

```java
    }

    /**
     * Updates all Outflows in the model.
     * @param time The current value of the simulation clock.
     */
    private void UpdateOutflows(double time) throws ValueComputationException {
        logger.debug("Updating Model Outflows at time = " + time + ".");
        for (Map.Entry<String, ModelElement> entry : this.modelElements.entrySet()) {
            logger.debug("Updating model element " + entry.getKey() + " at time = " +
time + ".");
            entry.getValue().UpdateOutflows(time);
        }
    }
}
```

Scheduler.java

```java
/**
 * File: Scheduler.java
 * Creation Date: June 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.SimulationEngine;

// Import statements
import DynSysMod.ModelElements.Model;
import DynSysMod.ModelElements.ModelElement;
import DynSysMod.ValueComputationException;

import java.util.*;

/**
 * Uses the Time Clocks defined in a Model's element to update the appropriate
 * Model elements at each time interval.  This class should be subclassed in
 * order to implement different scheduling algorithms, based on the relationships
 * of the model elements and time clocks.
 * @author Jennifer Leaf
 */
public abstract class Scheduler {

    /** The model elements that will be updated at each time interval.
     * This collection should include Sources, Systems, and Sinks - not the
     * flows and reservoirs directly.  The Sources, Systems, and Sinks will
     * be responsible for updating their internal objects.
     */
    protected Hashtable<String, ModelElement> modelElements;

    /** Creates a new instance of the Scheduler class. */
    protected Scheduler(Model model) {
        this.modelElements = model.GetModelElements();
    }

    /**
     * Calculates all elements that require updating at the given time.
     * @param time The current value of the simulation clock.
     **/
    public abstract void CalculateAllElements(double time) throws
ValueComputationException;

}
```

SimulationEngine.java

```java
/**
 * File: SimulationEngine.java
```

```
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 * - Once allowing Systems to have their own time clocks is supported, the time
 * step that the simulation should use needs to be calculated rather than
 * just using the Model's clock.  The simulation should run at the smallest
 * clock interval.
 * - Using the LinearScheduler is hardcoded.  Once allowing Systems to have
 * their own time clocks is supported, the Model should be analyzed to determine
 * the appropriate scheduler to use.
 */

// Package declaration
package DynSysMod.SimulationEngine;

// Import statements
import DynSysMod.ModelElements.Model;
import DynSysMod.ValueComputationException;

import org.apache.log4j.*;

/**
 * The main simulation engine.  This simply manages the simulation clock and
 * notifies other components to do the heavy lifting at each time interval.
 * @author Jennifer Leaf
 */
public class SimulationEngine {

    /** The model to simulate. */
    private Model model;
    /** The interval to increment the simulation clock on.  See TODO above. */
    private double timeStep;
    /** The current simulation time.  Starts at t = 0. */
    private double currentTime;
    /** How long to run the simulation for.*/
    private double duration;

    private Logger logger = Logger.getLogger(SimulationEngine.class.getName());

    /**
     * Creates a new instance of the SimulationEngine class.
     * @param model The model to simulate.
     * @param timeStep The time interval to increment the simulation clock on.
     * @param duration How long to run the simulation for.
     */
    public SimulationEngine(Model model, double timeStep, double duration) {
        this.model = model;
        this.timeStep = timeStep;
        this.currentTime = 0;
        this.duration = duration;
    }

    /** Initializes the simulation engine to begin a run. */
    private void Initialize() throws ValueComputationException {
        this.model.Initialize();
    }

    /**
     * Run the simulation.
     */
    public void Run() throws java.io.IOException, ValueComputationException {
        logger.info("Starting the simulation.");

        this.Initialize();

        Scheduler sch = new LinearScheduler(this.model);

        // Increment the clock - this.Initialize() handles the operations for time t = 0;
        this.currentTime += this.timeStep;
```

```
        // We assume the duration interval is inclusive - once we reach the
        // duration, let the simulation run one more cycle.
        while (this.currentTime <= this.duration) {
            logger.info("Running simulation for time t = " + this.currentTime);

            sch.CalculateAllElements(this.currentTime);

            // Increment the clock.
            this.currentTime += this.timeStep;
        }
        // We're done.
    }
}
```

InterpreterTests.java

```
/**
 * File: InterpreterTests.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.UnitTests;

// Import statements
import DynSysMod.Equations.Interpreter;

import java.util.*;

import org.junit.*;
import static org.junit.Assert.*;

/**
 * Unit tests for the equation interpreter.
 * @author Jennifer Leaf
 */
public class InterpreterTests {

    /**
     * Creates a new instance of the InterpreterTests class.
     */
    public InterpreterTests() {
    }

    @Test
    public void TestSingleNumber() throws Exception {
        Interpreter inter = new Interpreter("1");
        Assert.assertEquals(1.0, inter.Evaluate());
    }

    @Test
    public void TestSingleNegativeNumber() throws Exception {
        Interpreter inter = new Interpreter("-1.6");
        Assert.assertEquals(-1.6, inter.Evaluate());
    }

    @Test
    public void TestSimpleAdditionIntegers() throws Exception {
        Interpreter inter = new Interpreter("2+3");
        Assert.assertEquals(5, inter.Evaluate());
    }

    @Test
    public void TestSimpleSubtractionIntegers() throws Exception {
        Interpreter inter = new Interpreter("25- 19");
        Assert.assertEquals(6, inter.Evaluate());
```

```
}

@Test
public void TestSimpleMultiplicationIntegers() throws Exception {
    Interpreter inter = new Interpreter("5 *16");
    Assert.assertEquals(80, inter.Evaluate());
}

@Test
public void TestSimpleDivisionIntegers() throws Exception {
    Interpreter inter = new Interpreter("12 / 3");
    Assert.assertEquals(4, inter.Evaluate());
}

@Test
public void TestSimpleAdditionPositiveDoubles() throws Exception {
    Interpreter inter = new Interpreter("2.45+3.198");
    Assert.assertEquals(5.648, inter.Evaluate());
}

@Test
public void TestSimpleSubtractionPositiveDoubles() throws Exception {
    Interpreter inter = new Interpreter("4819.18 - 429.173");
    Assert.assertEquals(4390.007, inter.Evaluate());
}

@Test
public void TestSimpleMultiplicationPositiveDoubles() throws Exception {
    Interpreter inter = new Interpreter("5.6 *16.7");
    Assert.assertEquals(93.52, inter.Evaluate());
}

@Test
public void TestSimpleDivisionPositiveDoubles() throws Exception {
    Interpreter inter = new Interpreter("6.26 / 1.6");
    Assert.assertEquals(3.9125, inter.Evaluate());
}

@Test
public void TestSimpleAdditionNegativeDoubles() throws Exception {
    Interpreter inter = new Interpreter("2.45+ -3.198");
    Assert.assertEquals(-0.748, inter.Evaluate());
}

@Test
public void TestSimpleSubtractionNegativeDoubles() throws Exception {
    Interpreter inter = new Interpreter("4819.18 - -429.173");
    Assert.assertEquals(5248.353, inter.Evaluate());
}

@Test
public void TestSimpleMultiplicationNegativeDoubles() throws Exception {
    Interpreter inter = new Interpreter("5.6 *-16.7");
    Assert.assertEquals(-93.52, inter.Evaluate());
}

@Test
public void TestSimpleDivisionNegativeDoubles() throws Exception {
    Interpreter inter = new Interpreter("6.26 / -1.6");
    Assert.assertEquals(-3.9125, inter.Evaluate());
}

@Test
public void TestSimpleAdditionParentheses() throws Exception {
    Interpreter inter = new Interpreter("(2.45+3.198)");
    Assert.assertEquals(5.648, inter.Evaluate());
}

@Test
public void TestSimpleSubtractionParentheses() throws Exception {
    Interpreter inter = new Interpreter("(4819.18 - 429.173)");
```

```
        Assert.assertEquals(4390.007, inter.Evaluate());
    }

    @Test
    public void TestSimpleMultiplicationParentheses() throws Exception {
        Interpreter inter = new Interpreter("(5.6 *16.7)");
        Assert.assertEquals(93.52, inter.Evaluate());
    }

    @Test
    public void TestSimpleDivisionPositiveDoublesParentheses() throws Exception {
        Interpreter inter = new Interpreter("(6.26 / 1.6)");
        Assert.assertEquals(3.9125, inter.Evaluate());
    }

    // 0.1 * 150 * (1+1)
    @Test
    public void TestRandomString1() throws Exception {
        Interpreter inter = new Interpreter("0.1 * 150 * (1+1) ");
        Assert.assertEquals(30, inter.Evaluate());
    }

    public static void main(String args[]) {
      org.junit.runner.JUnitCore.main("DynSysMod.UnitTests.InterpreterTests");
    }
}
```

ReservoirTests.java

```
/**
 * File: ReservoirTests.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.UnitTests;

// Import statements
import DynSysMod.ModelElements.*;
// Needed to disambiguate from java.util.System.
import DynSysMod.ModelElements.System;
import DynSysMod.ValueComputationException;

import org.junit.*;
import static org.junit.Assert.*;

/**
 * Unit tests for the Reservoir class.
 * @author Jennifer Leaf
 */
public class ReservoirTests {

    private Reservoir reservoir;
    private System owner;
    private String equation = "testReservoir = 1200 * t";


    public ReservoirTests() {
        this.owner = new System("system", new ModelValueDatabase(), new TimeSpan(1,
"day"));
        this.reservoir = new Reservoir("system.testReservoir", new ModelValueDatabase(),
this.owner, 0, 100, Reservoir.OverflowBehavior.Error);
    }

    @Test
    public void testSetProcessEquation() {
        this.reservoir.SetProcessEquation(this.equation);
```

```java
    }

    @Test
    public void testInitialize() throws Exception {
        this.reservoir.Initialize();
    }

    @Test(expected=DynSysMod.ModelElements.CapacityOverflowException.class)
    public void testUpdateReservoirsForceError() throws Exception {
        double time = 1.0;

        this.reservoir.SetProcessEquation(this.equation);
        this.reservoir.UpdateReservoirs(time);
    }

    @Test
    public void testUpdateReservoirsDiscardOverflow() throws Exception {
        double time = 1.0;

        this.reservoir = new Reservoir("system.testReservoir", new ModelValueDatabase(),
this.owner, 0, 100, Reservoir.OverflowBehavior.DiscardOverflow);
        this.reservoir.SetProcessEquation(this.equation);
        this.reservoir.UpdateReservoirs(time);
        Assert.assertEquals(100, this.reservoir.GetValue("system.testReservoir", time));
    }

    @Test
    public void testUpdateReservoirsIgnoreOverflow() throws Exception {
        double time = 1.0;

        this.reservoir = new Reservoir("system.testReservoir", new ModelValueDatabase(),
this.owner, 0, 100, Reservoir.OverflowBehavior.Ignore);
        this.reservoir.SetProcessEquation(this.equation);
        this.reservoir.UpdateReservoirs(time);
        Assert.assertEquals(1200, this.reservoir.GetValue("system.testReservoir", time));
    }

    @Test
    public void testUpdateOutflows() {
        double time = 1.0;
        this.reservoir.UpdateOutflows(time);
    }

    public static void main(String args[]) {
        org.junit.runner.JUnitCore.main("DynSysMod.UnitTests.ReservoirTests");
    }
}
```

SinkTests.java

```java
/**
 * File: SinkTests.java
 * Creation Date: July 2007
 *
 * Modifications:
 *
 * TODO:
 */

// Package declaration
package DynSysMod.UnitTests;

// Import statements
import DynSysMod.ModelElements.*;

import org.junit.*;
import static org.junit.Assert.*;

/**
 * Unit tests for the Sink class.
 * @author Jennifer Leaf
```

```
 */
public class SinkTests {

    private ModelValueDatabase valueDatabase;
    private Sink sinkInstance;

    private final double initialValue = 30.3;
    private final String sinkName = "Test Sink";

    /** Creates a new instance of SinkTests */
    public SinkTests() {
    }

    @Before
    public void SetUp() {
        this.valueDatabase = new ModelValueDatabase();
        sinkInstance = new Sink(this.sinkName, this.valueDatabase, this.initialValue, new
TimeSpan(1, ""));
    }

    @Test
    public void TestInitialize() throws Exception {
        sinkInstance.Initialize();
        Assert.assertTrue(this.valueDatabase.Contains(this.sinkName));
        Assert.assertEquals(this.initialValue, this.valueDatabase.GetValue(this.sinkName,
0));
    }

    public static void main(String args[]) {
        org.junit.runner.JUnitCore.main("DynSysMod.UnitTests.SinkTests");
    }
}
```