A Semantic Framework for Modeling and Simulation of Cyber-Physical Systems

Parastoo Delgoshaei
Department of Civil and
Environmental Engineering
University of Maryland
College Park, MD 20742, USA
Email: parastoo@umd.edu

Mark A. Austin
Department of Civil and
Environmental Engineering
University of Maryland
College Park, MD 20742, USA
Email: austin@isr.umd.edu

Amanda J. Pertzborn
Energy and Environment Division
National Institute of Standards and
Technology (NIST)
Gaithersburg, MD 20899, USA
Email: amanda.pertzborn@nist.gov

Abstract—**This paper describes a new semantic framework for model-based systems engineering, requirements traceability, and system simulation and assessment of cyber-physical systems (CPSs). When fully developed this environment will support the organization and integration of hierarchies of physical and software components, and perform analysis on their discrete and continuous behavior. Results of computational analysis will work alongside domain ontologies for decision making and rule checking procedures. To support the modeling and simulation of physical system behavior, and integration of the physical and cyber domains, we introduce Whistle, a new scripting language where physical units are embedded within the basic data types, matrices, and method interfaces to external object-oriented software packages. The capabilities of Whistle are demonstrated through a series of progressively complicated applications.**

***Keywords*-Cyber-Physical System; Semantic Modeling; Simulation Environment; Software Design Pattern; Rule Checking.**

I. Introduction

Problem Statement. This paper is concerned with the development of procedures and software for the model-based systems engineering, integration, simulation and performance-assessment of cyber-physical systems (CPS). It builds upon our previous work [1] on semantic platforms for requirements traceability and system assessment. As illustrated in Figure 1, the distinguishing feature of CPS is a coupling of physical and cyber systems, with the cyber affecting the physical and vice versa. In a typical CPS application, embedded computers and networks will monitor and control physical processes, usually with feedback. The basic design requirement is that software and communications technologies will work together to deliver functionality that is correct and works with no errors. Unfortunately, present-day design procedures are inadequate for the design of modern CPS systems. A key problem is that today we do not have a mature science to support systems engineering of high-confidence cyber-physical systems assembled from subsystems having multiple physics (e.g., chemical, mechanical, electrical) [2], [3]. Design space exploration and trade studies are also difficult to conduct because decision variables span parametric, logical, and dependency relationship types. Components are often required to serve multiple functions – as such, cause-and-effect mechanisms are no longer localized and obvious. System relationships can reach laterally across systems hierarchies and/or intertwined network structures.

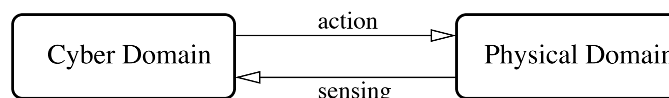
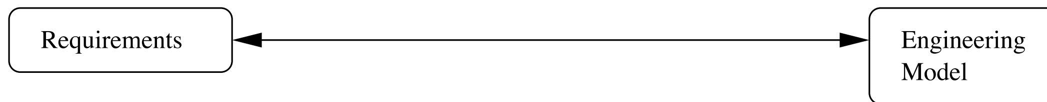


Figure 1. Interaction of cyber and physical domains in CPS.

In order for cyber-physical design procedures to proceed in a rational way we need mechanisms to easily combine abstractions from multiple physics and field equations (e.g., solids, fluids, heat, electromagnetics, chemistry) into sets of coupled equations that model the system. Components may be discrete (e.g., rigid body elements, control actuation elements, software logic), or continuous (e.g., differential equations for fluid flow). The challenge in developing accurate models of CPS behavior is complicated by differences in the underlying operation and data-stream flows associated with cyber and physical components. Whereas physical systems tend to have behavior that is continuous and associated with flows having physical quantities, cyber operates on discrete logic. To address these limitations, new computer programs and languages are required to address the challenges of distributed, complex CPSs. Their capabilities need to include establishing feedback loops between physical processes and computational units involving robust analysis, decision making mechanisms, dynamic modeling, knowledge of sensors and actuators, and computer networks. In a step toward creating this long-term goal, we are working on the development of a computational infrastructure where domain specific ontologies and rule checking routines operate hand-in-hand with a new scripting language introduced here as Whistle. This new language employs object-oriented design principles and software design patterns as a pathway to addressing challenging design questions.

Model-based Systems Engineering. Model-based systems engineering (MBSE) development is an approach to systems-level development in which the focus and primary artifacts of development are models, as opposed to documents. Our research methodology is driven by a need to achieve high levels of productivity in system development. We believe that high levels of productivity in system development can be achieved through the use of high-level visual abstractions coupled with lower-level (mathematical) abstractions suitable for formal systems analysis. The high-level abstractions provide

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org



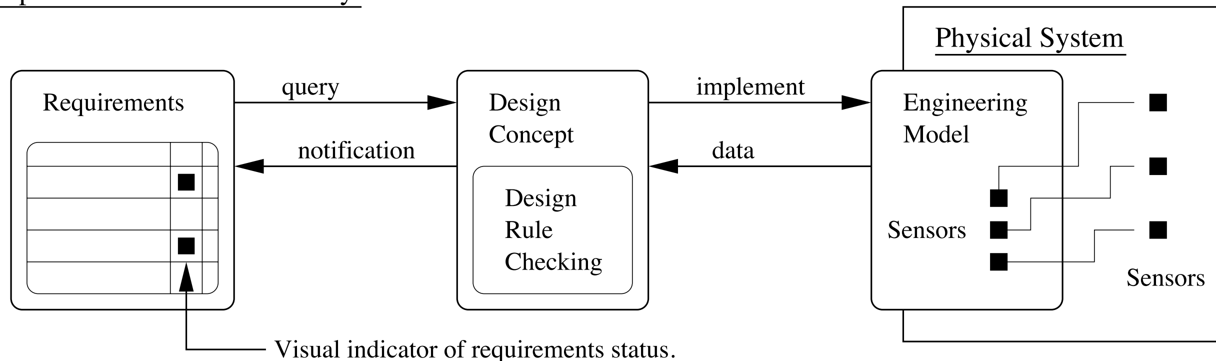


Figure 2. Schematics for: (top) state-of-the-art traceability, and (bottom) proposed model for ontology-enabled traceability for systems design and management.

a "big picture" summary of the system under development and highlight the major components, their connectivity, and performance. The lower-level abstractions are suitable for formal systems analysis – for example, verification of component interface compatibilities and/or assessment of system performance through the use of simulation methods. The former one is achieved through semantic web technologies, i.e., with domain specific ontologies. On the other hand, detailed simulation analysis can be performed by scripting language, or other analysis packages that are compatible with scripting language.

A tenet of our work is that methodologies for strategic approaches to design will employ semantic descriptions of application domains, and use ontologies and rule-based reasoning to enable validation of requirements, automated synthesis of potentially good design solutions, and communication (or mappings) among multiple disciplines [4][5][6]. A key element of required capability is an ability to identify and manage requirements during the early phases of the system design process, where errors are cheapest and easiest to correct. The systems architecture for state-of-the-art requirements traceability and the proposed platform model is shown in the upper and lower sections of Figure 2. In state-of-the-art traceability mechanisms, design requirements are connected directly to design solutions (i.e., objects in the engineering model). Our contention is that an alternative and potentially better approach is to satisfy a requirement by asking the basic question: What design concept (or group of design concepts) should I apply to satisfy a requirement? Design solutions are the instantiation/implementation of these concepts. The proposed architecture is a platform because it contains collections of domain-specific ontologies and design rules that will be reusable across applications. In the lower half of Figure 2, the textual requirements, ontology, and engineering models provide distinct views of a design: (1) Requirements are a statement of "what is required." (2) Engineering models are a statement of "how the required functionality and performance might be achieved," and (3) Ontologies are a statement of "concepts justifying a tentative design solution." During design, mathematical and logical rules are derived from textual requirements which, in turn, are connected to elements in an engineering model. Evaluation of requirements can include checks for satisfaction of system functionality and performance, as well as identification of conflicts in requirements themselves. A key benefit of our approach is that design rule checking can be applied at the earliest stage possible – as long as sufficient data is available for the evaluation of rules, rule checking can commence; the textual requirements and engineering models need not be complete. During the system operation, key questions to be answered are: What other concepts are involved when a change occurs in the sensing model? What requirement(s) might be violated when those concepts are involved in the change? To understand the inevitable conflicts and opportunities to conduct trade space studies, it is important to be able to trace back and understand cause-and-effect relationships between changes at system-component level and their affect on stakeholder requirements. Present-day systems engineering methodologies and tools, including those associated with SysML [7] are not designed to handle projects in this way.

Scope and Objectives. This paper describes a new approach to requirements traceability, simulation, and system assessment through the use of semantic platforms coupled with a component-based language where physical quantities (not just numbers) are deeply embedded in the language design and execution. The rationale for providing cyber with this capability is simple: if the cyber has an enhanced ability to represent the physical world in which it is embedded, then it will be in a better position to make decisions that are appropriate and correct.

Our test-bed application area and driver for this research is performance-based modeling and design of energy-efficient building environments. Modern buildings contain a variety of intertwined networks for the hierarchical arrangement of spaces (e.g., buildings have floors, floors contain rooms, and so forth), for fixed circulatory systems, e.g., power and heating, ventilation, and air conditioning (HVAC), for dynamic circulatory systems, e.g., air and water flows, and for wired and wire-

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

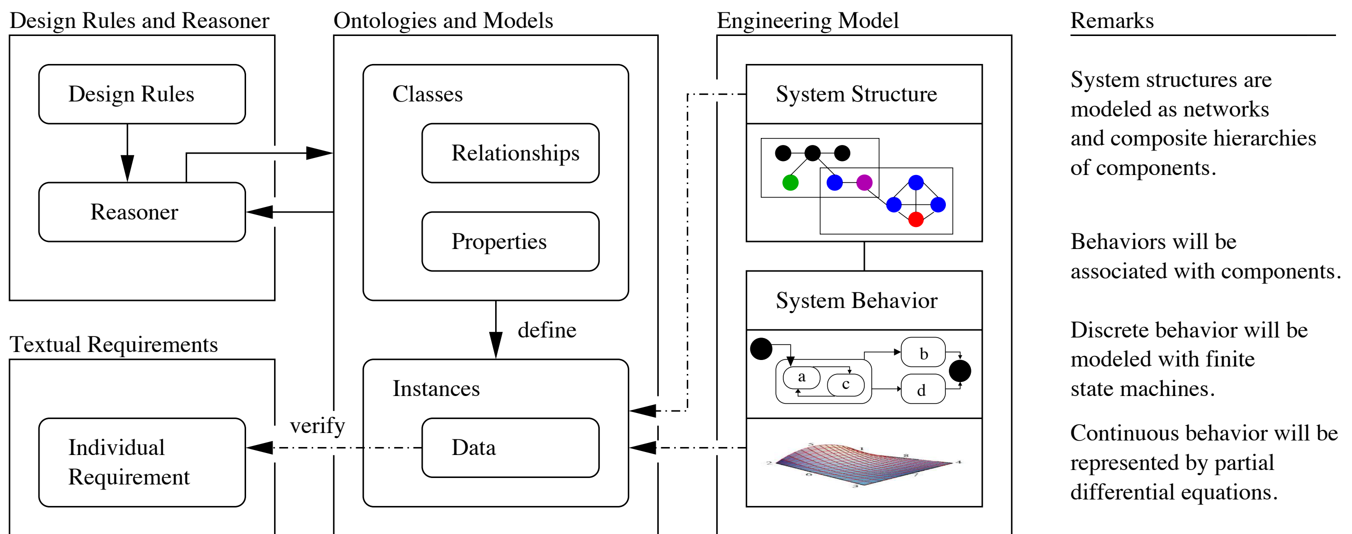


Figure 3. Framework for implementation of ontology-enabled traceability and design assessment.

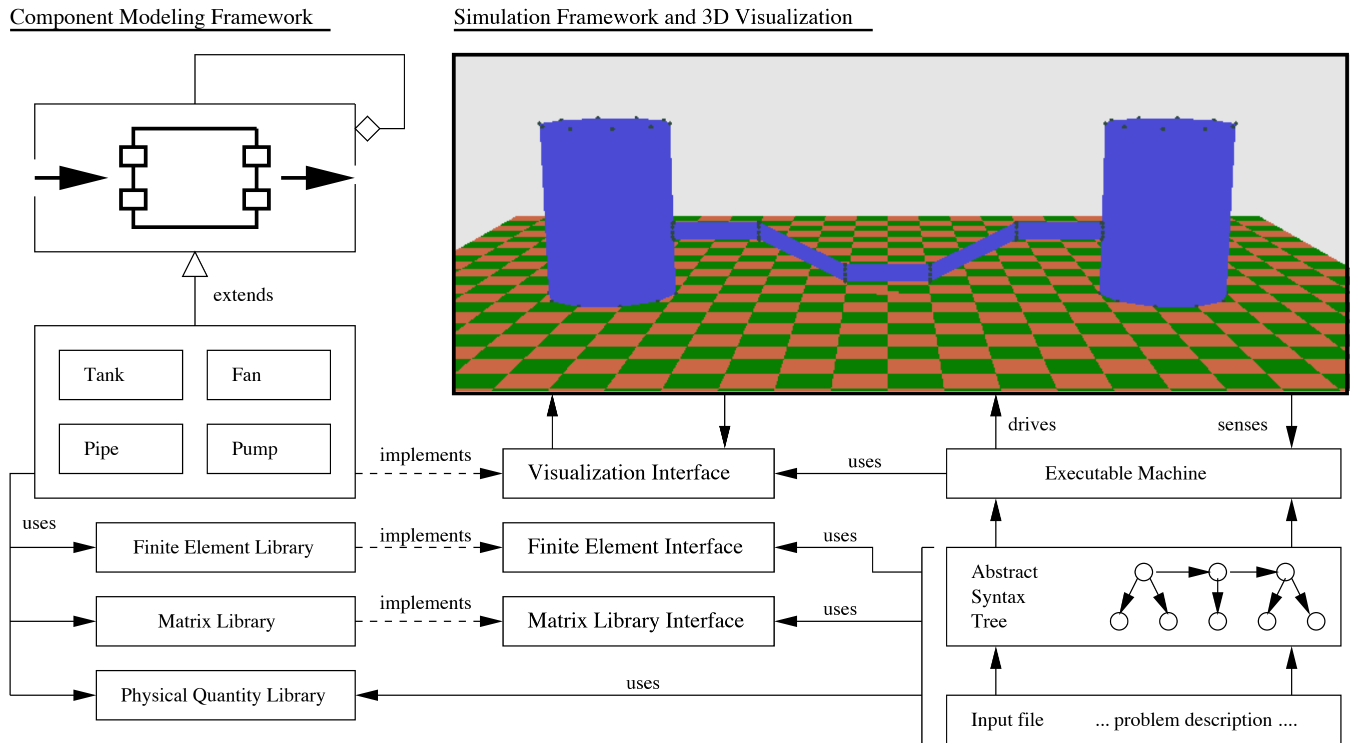


Figure 4. Architecture for modeling HVAC systems as networks of connected components, and using finite element solution procedures for computing and visualizing time-history behavior.

less communications. While there is a desire for each network to operate as independently as possible, in practice the need for new forms of functionality will drive components from different network types to connect in a variety of ways. Within the building simulation community state-of-the-art dynamic simulation is defined by Modelica, and steady-state simulation by DOE-2 and eQuest. From a CPS perspective, the time-history analysis and control of building system performance is complicated by the need to model combinations of discrete (e.g., control) and continuous behaviors (e.g., the physics of fluid dynamics). Predictions of dynamic behavior correspond to the solution of nonlinear differential algebraic equations (e.g., for water, air, and thermal flow) coupled to discrete equations (e.g., resulting from cyber decisions).

To facilitate and support this vision, we are currently working toward the platform infrastructure proposed by Figures 3 and 4. Figure 3 pulls together the different pieces of the proposed architecture shown in Figure 2. On the left-hand side the textual requirements are defined in terms of mathematical and logical rule expressions for design rule checking. Figure 4 highlights the software infrastructure for modeling systems that are part cyber and part physical. To deal with the complexity of

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

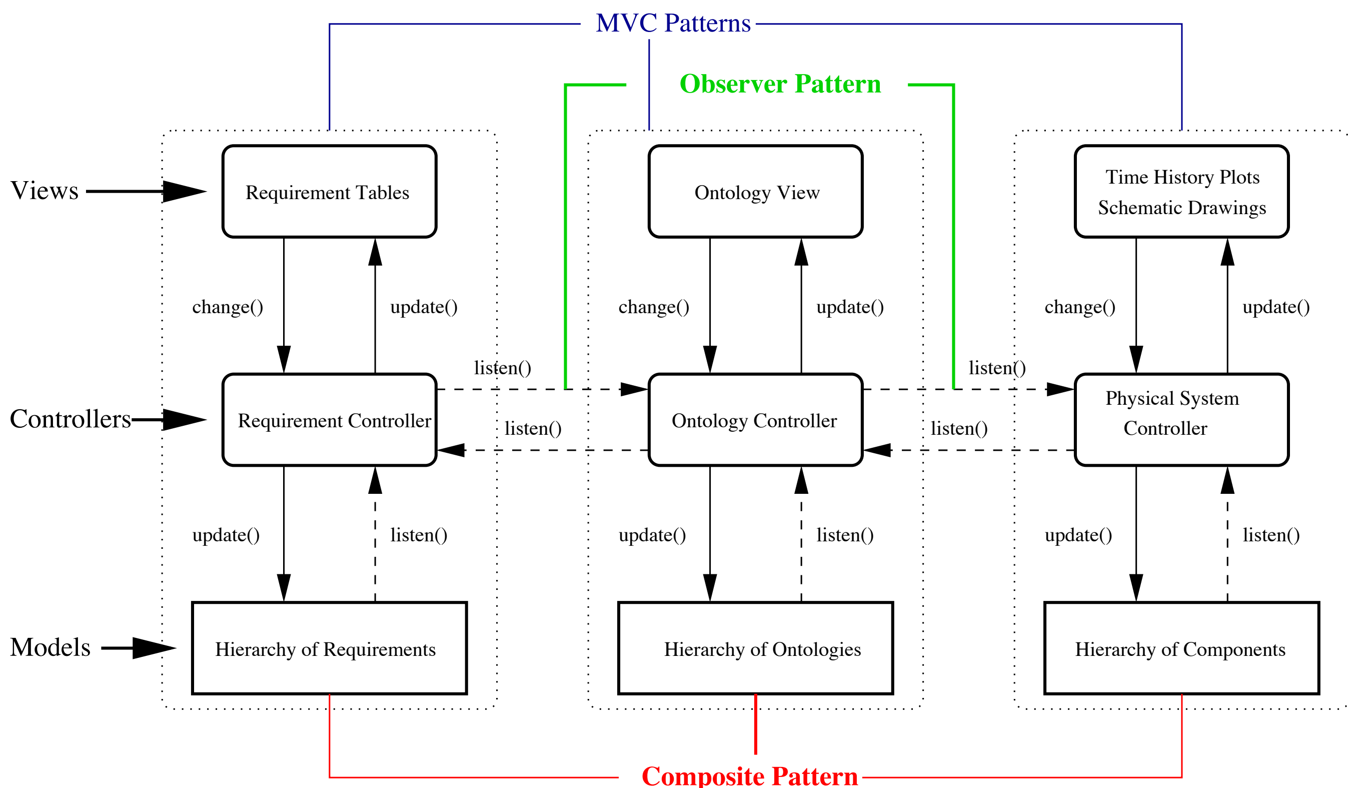


Figure 5. Software architecture for ontology-enabled traceability, annotated with model-view-controller, observer and composite-hierarchy design patterns.

building systems, which are defined by large numbers of physical and abstract components, we are proposing that models be organized into composite hierarchies, as shown on the top left-hand side of Figure 4. Specific component types will simply implement the composite hierarchy interface. To accommodate mixtures of discrete and continuous behavior, we are proposing that the software architecture implement a series of generic interfaces to software libraries for matrix computations, finite element analysis, and two- and three-dimensional visualization. This element is shown along the bottom of Figure 4. Finally, we need a capability for components to communicate across hierarchies, and we are proposing this be accomplished with listener mechanisms (e.g., a controller component might listen for data from a collection of sensor components). This is a work in progress. Looking ahead, our plans are to build a series of progressively capable software prototypes, with each iteration of development employing a combination of executable statecharts for the behavior modeling of HVAC components, and eventually finite element procedures for the computation of behaviors over continuous physical domains (e.g., fluid flow in a pipe network) [8][9][10].

This paper begins with a description of the semantic platform infrastructure and our use of software design patterns [11] (e.g., networks of model, view, controllers), software libraries and languages for semantic applications development using OWL [12] and Jena [13]. Section III describes related work. Section IV describes the design and features of Whistle, a scripting language we are developing to support the implementation of abstractions shown in Figures 3 and 4. A series of progressively complicated case study problems is presented in Section V.

II. Semantic Platform Infrastructure

Software Systems Architecture. Figure 5 represents the software architecture for ontology-enabled traceability and physical systems simulation, annotated with our use of model-view-controller, observer, and composite hierarchy software design patterns. Software design patterns are defined as general repeatable solutions to common software design problems; designers customize these templates to suit the design requirements. The model-view-controller (MVC) pattern is an architectural pattern with three components of model, view, and controller. This pattern is widely used in graphical user interface (GUI) applications. The observer design pattern defines a one-to-many relationship between objects. An observer component registers itself to a subject of interest and will be notified when an event occurs. An observer can register to different observable components or be removed when the interest no longer exists. The composite design pattern is used to describe groups of objects whose natural organizational structure is a hierarchy (e.g., a building contains floors; floors contain rooms; rooms contain desks and chairs). For composite hierarchies that represent spatial systems, algorithms can be developed to systematically traverse the hierarchy and process it according to a pre-defined purpose (e.g., display the contents of a hierarchy of coordinate systems; query to see if a point is inside a particular object). Another key benefit is model flexibility. Suppose, for example, that an engineer is working with a simple model of a building consisting of an air-handling unit and rooms defined by walls and doors and windows inside walls. If the room model is adjusted to a different orientation, then all of the subsystem elements (i.e., the walls, doors and windows) will be automatically re-positioned.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

We employ a combination of MVC, observer, and composite hierarchy design patterns to synthesize dependency and data flow relationships between the requirements, ontology and engineering model work spaces, and a modified version of MVC where the controller serves as a mediator between multiple models and views. The latter can also be found in Apple Cocoa [14]. The requirements, ontology, and the physical system models are each represented as MVC nodes. Inside a MVC node, messages are distributed between the controller, views and models. Then, the observer design pattern is used to connect controller elements at each MVC node to other points of interest, thereby enabling traceability and flows of data across the system architecture. The controller registers with the model to be notified of a change in a property, and then updates views following a change in a model property. In practical terms, an end-user interacts with the views and makes changes to the model by passing data through the controller. Views pass the change queries to the controller and the controller updates the relevant models.

The composite hierarchy design pattern is used to organize the entities within each workspace. For the requirements model, this implies definition of compound requirements containing other sub-requirements. For the ontology models this implies that far-reaching ontology might be assembled from collections of ontologies describing specific domains. For example, an ontology for building systems might contain a mechanical systems ontology, among others. Finally, physical system models are created as hierarchies of components. Notice that the ontology controller is listening to the physical system controller and vice versa. This mechanism means that as a system is operating or is being simulated, changes in the system state will be reported to the ontology controller and will be updated in the data stored (individuals) in the ontology model. Looking the other way, an update to the value of a component attribute in the physical system model will trigger rule checking in the ontology workspace and possibly a change in the satisfaction of system requirements. For both scenarios, views will be updated upon a change in their models. The requirement controller listens to the ontology controller. This connection is the traceability thread back to the requirements. The requirements view will highlight the relevant requirement when the associated rule in the ontology is triggered.

Modeling and Reasoning with Ontologies. Textual requirements are connected to the ontology model and logical and mathematical design rules, and from there to the engineering model. Ontology models encompass the design concepts (ontology classes) in a domain, as well as the relationships among them. Classes are qualified with properties (c.f., attributes in classes) to represent the consequence of constraint and design rule evaluations. Examples of valid relationships are: containment, composition, uses, and "Is Kind of". These classes are place holders for the data extracted from the engineering model. Individuals are the object counterpart of classes, with data and object property relationships leading to the resource description framework -(RDF) graph infrastructure. Each instance of an individual holds a specific set of values obtained from the engineering model.

Rules serve the purpose of constraining the system operation and/or system design. They provide the mechanisms for early design verification, and ensure the intended behavior is achieved at all times during system operation. We are currently working with reasoners provided in the Jena API. A reasoner works with the RDF graph infrastructure and sets of user-defined rules to evaluate and further refine the RDF graph. Rule engines are triggered in response to any changes to the ontological model. This process assures that the model is consistent with respect to the existing rules. Traceability from ontologies to requirements is captured via implementation of the listeners that are notified as a result of change in the semantic model.

In a departure from past work, we are exploring the feasibility of creating built-in functions to capture and evaluate performance criteria, i.e., energy efficiency of the HVAC system. A second potential use of built-in functions is as an interface to packages that provide system improvements through optimization and performance related queries. We note that a rule-based approach to problem solving is particularly beneficial when the application logic is dynamic (i.e., where a change in a policy needs to be immediately reflected throughout the application) and rules are imposed on the system by external entities [15][16]. Both of these conditions apply to the design and management of engineering systems.

III. Related Work

An important facet of our work is use of Semantic Web technologies [17] as both system models and mechanisms to derive system behavior. While the vast majority of Semantic Web literature has used ontologies to define system structure alone, this is slowly changing. Derler and co-workers [18] explain, for example, how ontologies along with hybrid system modeling and simulation and concurrent models of computation can help us better address the challenges of modeling cyber-physical systems (CPSs). These challenges emerge from the inherited heterogeneity, concurrency, and sensitivity to timing of such systems. Domain specific ontologies are used to strengthen modularity, and to combine the model of system functionality with system architecture. As a case in point, the Building Service Performance Project proposes use of ontologies and rules sets to enhance modularity and perform cross-domain information exchange and representation [19]. Koelle and Strijland are investigating the design and implementation of a software tool to support semantic-driven architecture with application of rules for security assurance of large systems in air navigation [20].

For the cyber side of the CPS problem, visual modeling languages such as the Unified Modeling Language (UML) and SysML provide weak semantic support for MBSE. This leads us to consider languages and tools for MBSE that have stronger semantics. Consider, for example, the possibility of conceptual modeling through the use of ontologies and constraints represented as rules. In the physical domain, some modeling languages and modeling frameworks are developed to address the physical modeling and analysis of complex physical systems. Two well known examples are Modelica [21] and Ptolemy II [22]. Modelica offers strong physical modeling capabilities and features to be utilized in component based modeling. Physical equations are embedded inside components and components are connected together via ports. Some frameworks such as Open Modelica have been developed

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

to support graphical block diagram modeling with Modelica. Ptolemy studies modeling and simulation of concurrent real-time systems with actor-based designs. Actors are software components that communicate via message sending. A model is a network of interconnected actors. Moreover, directors implement a model of computation in this framework and can be attached to different layers of the model. For example, discrete-events (DE), data-flow (SDF), and 3-D visualization are some of the directions supported in Ptolemy [23]. The challenges for CPS design are greater because we need both the cyber and physical models to interact with each other, and at this time the bi-directional link connecting physical (continuous) operations to computational (discrete) operations is missing. Ongoing work is trying not only to cover this gap, but also take a step toward tieing the governing rules in the domain-specific ontologies to the textual requirements [24]. The work by Simko [25] uses CyPhyML, Hybrid Bond Graphs and ESMoL to formally describe the structure and behavior of CPSs. However, deductive reasoning is lacking in this work.

IV. WHISTLE SCRIPTING LANGUAGE

This section introduces Whistle, a new scripting language where physical units are deeply embedded within the basic data types, matrices, branching and looping constructs, and method interfaces to external object-oriented software packages. Whistle builds upon ideas prototyped in Aladdin [26][27][28] a scripting environment for the matrix and finite element analysis of engineering systems.

Language Design and Implementation. Scripting languages [29][30][31] are designed for rapid, high-level solutions to software problems, ease of use, and flexibility in gluing application components together. They facilitate this process by being weakly typed and interpreted at run time. Weakly typed means that few restrictions are placed on how information can be used a priori – the meaning and correctness of information is largely determined by the program at run time. And since much of the code needed to solve a problem using a system programming language is due to the language being typed, broadly speaking, weakly typed scripting languages require less code to accomplish a task [32]. Whistle is tiny in the sense that it uses only a small number of data types (e.g., physical quantities, matrices of physical quantities, booleans and strings). Features of the language that facilitate the specification of problem solutions include: (1) liberal use of comment statements (as with C and Java, c-style and in-line comment statements are supported), (2) consistent use of function names and function arguments, (3) use of physical units in the problem description, and (4) consistent use of variables, matrices, and looping and branching structures to control the flow of program logic.

Whistle is implemented entirely in Java. We use the tools JFlex (the Fast Scanner Generator for Java) [33] and BYACC/J (an extension of Berkeley YACC for Java) [34] to handle the parsing and lexical analysis of tokens and statements, Java Collections for the symbol table, and a variety of tree structure representations of the abstract syntax tree. A good introduction to symbol tables and abstract syntax tree representations can be found in the compilers and interpreters text by Mak [35].

Definition and Management of Physical Quantities. A physical quantity is a measure of some quantifiable aspect of the modeled world. In Whistle, basic engineering quantities such as length, mass, and force, are defined by a numerical value (number itself) plus physical units. Figure 6 is a subset of units presented in the Unit Conversion Guide [36], and shows the primary base units, supplementary units, and derived units that occur in engineering mechanics and structural analysis. The four basic units needed for engineering analysis are: length unit L; mass unit M; time unit t; and temperature unit T. Planar angles are represented by the supplementary base unit rad. Derived units are expressed algebraically in terms of base and supplementary units by means of multiplication and division, namely:

$$\text{units} = k \cdot L^{\alpha} M^{\beta} t^{\gamma} T^{\delta} \cdot \text{rad}^{\epsilon} \tag{1}$$

where $\alpha, \beta, \gamma, \delta$ and ϵ are exponents, and k is the scale factor. Numbers are simply non-dimensional quantities represented by the family of zero exponents $[\alpha, \beta, \gamma, \delta, \epsilon] = [0, 0, 0, 0, 0]$. The four basic units play the primary role in determining dimensional consistency of units in physical quantity and matrix operations. Because a radian represents the ratio of two distances (i.e., distance around the perimeter of a circle divided by its radius), most software implementations deal with radians as if they were dimensionless entities. Whistle departs from this trend by explicitly representing radians, and employing a special set of rules for their manipulation during physical quantity and matrix operations.

The scripting language libraries provide facilities for dynamic allocation of units (in both the US and SI systems), units copying, consistency checking and simplification, and units printing. Operations for units conversion are provided. In an effort to keep the scripting language usage and implementation as simple as possible, all physical quantities are stored as floating point numbers with double precision accuracy, plus units. Floating point numbers are viewed as physical quantities without units. There are no integer data types in Whistle.

Physical Quantity Arithmetic. Whistle supports the construction and evaluation of physical quantity expressions involving arithmetic, relational, and logical operators. The integration of units into the scripting language provides a powerful check for the dimensional consistency of formulas. A detailed summary may be found in Tables I and II. Suppose, for example, that we want to compute the force needed to move 1 kg over a distance of 10 m in 2 seconds. The fragment of code:

```
mass     = 1 kg;
distance = 10 m;
dt       = 2 sec;

force01 = mass*distance/dt^2;
print "*** Required force = ", force01;
```

demonstrates the procedure for defining the physical quantity variables mass (kg), distance (m) and dt (sec), and computing the required force. The output is:

```
*** Required force = [ 2.500, N]
```

Whistle provides a small library of built-in constants (e.g., Pi) and functions (e.g., Max(), Min(), Sqrt()) for the evaluation

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

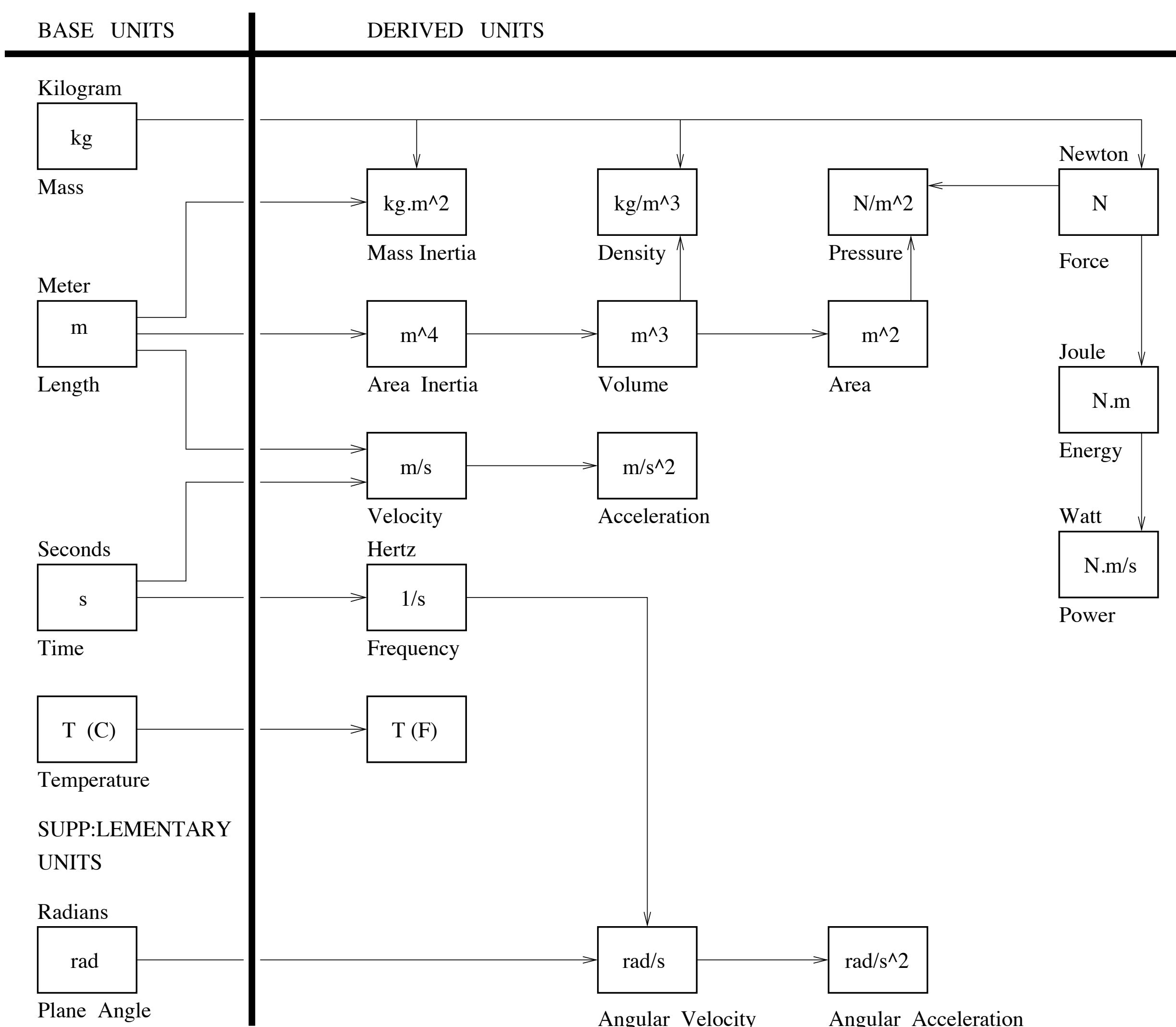


Figure 6. Primary base and derived units commonly found in engineering mechanics.

TABLE I. UNITS ARITHMETIC IN ARITHMETIC OPERATIONS

Description	Expression	Scale Factor	Unit Exponents
Addition	$q_1 + q_2$	k_1	$[\alpha_1, \beta_1, \gamma_1, \delta_1, \epsilon_1]$
Subtraction	$q_1 - q_2$	k_1	$[\alpha_1, \beta_1, \gamma_1, \delta_1, \epsilon_1]$
Multiplication	$q_1 * q_2$	$k_1 \cdot k_2$	$[\alpha_1 + \alpha_2, \beta_1 + \beta_2, \gamma_1 + \gamma_2, \delta_1 + \delta_2, \epsilon_1 + \epsilon_2]$
Division	q_1 / q_2	k_1 / k_2	$[\alpha_1 - \alpha_2, \beta_1 - \beta_2, \gamma_1 - \gamma_2, \delta_1 - \delta_2, \epsilon_1 - \epsilon_2]$
Exponential	$q_1 \,\hat{}\, q_2$	$k_1^{N\dagger}$	$[N\alpha_1, N\beta_1, N\gamma_1, N\delta_1, N\epsilon_1]^\dagger$

TABLE II. EXPRESSIONS INVOLVING RELATIONAL AND LOGICAL OPERATORS. A UNITS CONSISTENCY CHECK IS MADE BEFORE THE OPERATION PROCEEDS, AND THE RESULT OF THE OPERATION IS EITHER TRUE (1) OR FALSE (0). HERE WE ASSUME `x = 2 in` AND `y = 2 ft`.

Operator	Description	Example	Result
$<$	less than	$x < y$	true
$>$	greater than	$x > y$	false
$<=$	less than or equal to	$x <= y$	true
$>=$	greater than or equal to	$x >= y$	false
$==$	identically equal to	$x == y$	false
$!=$	not equal to	$x != y$	true
&&	logical and	$(x < y)$ && $(x <= y)$	true
$\|\|$	logical or	$(y < x) \,\|\|\, (x <= y)$	true
!	logical not	$!y$	false

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

of arithmetic expressions involving physical quantities. For example, the expressions:

```
print "Compute: Abs ( -2 cm ) --> ",
                Abs ( -2 cm );
print "Compute: Min ( 2 cm,   3 cm ) --> ",
                Min ( 2 cm,  3 cm );
print "Compute: Max ( 2 cm,   3 cm ) --> ",
                Max ( 2 cm,  3 cm );
```

generate the output:

```
Compute: Abs ( -2 cm ) --> [ 0.02000, m]
Compute: Min ( 2 cm,   3 cm ) --> [ 2.000, cm]
Compute: Max ( 2 cm,   3 cm ) --> [ 3.000, cm]
```

Relational and Logical Expressions. Whistle provides support for the representation and evaluation of relational expressions involving the “and operator” (&&), the “or operator” (||), and physical quantities. Consider, for example, the pair of lengths:

```
x = 10 cm; y = 20 cm;
```

The ensemble of expressions:

```
print "z01 = x <= 15 cm && y > x --> ",
             x <= 15 cm && y > x;
print "z02 = x <= 15 cm && y < x --> ",
             x <= 15 cm && y < x;
print "z03 = x <= 15 cm || y > x --> ",
             x <= 15 cm || y < x;
```

generates the output:

```
z01 = x <= 15 cm && y > x --> true
z02 = x <= 15 cm && y < x --> false
z03 = x <= 15 cm ||y > x --> true
```

Program Control. Program control is the basic mechanism in programming languages for using the outcome of logical and relational expressions to guide the pathway of a program execution. Whistle supports the “if” and “if-else” branching constructs, and the “while” and “for” looping constructs, with logical and relational operations being computed on physical quantities. The fragment of code:

```
x = 0 cm;
while ( x <= 10 cm ) {
   print "*** x = ", x;
   if ( x <= 5 cm ) {
      x = x + 1 cm;
   } else {
      x = x + 2 cm;
   }
}
```

generates the output:

```
*** x = [ 0.000, cm]
*** x = [ 1.000, cm]
*** x = [ 2.000, cm]
*** x = [ 3.000, cm]
*** x = [ 4.000, cm]
*** x = [ 5.000, cm]
*** x = [ 6.000, cm]
*** x = [ 8.000, cm]
*** x = [ 10.00, cm]
```

and demonstrates the basic functionality of a while loop and if-else branching construct working together.

Matrix Data Structure. Figure 7 shows the high-level layout of memory for the matrix data structure.

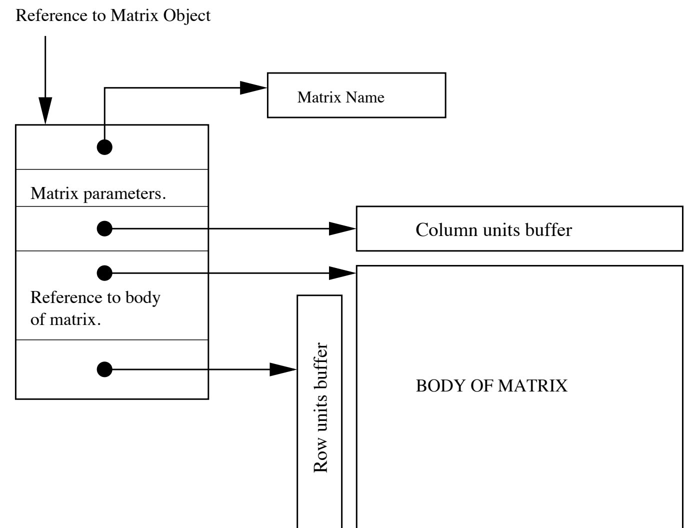


Figure 7. Layout of memory in matrix data structure.

Memory is provided for a character string containing the matrix name, two integers for the number of matrix rows and columns, as well as the matrix body. The matrix element units are stored in two one-dimensional arrays of type `Dimension`. One array stores the column units, and a second array the row units. The units for matrix element at row `i` and column `j` are simply the product of the `i-th` element of the row units buffer and the `j-th` element of column units buffer. Our use of row and column units matrices means that this model does not support the representation of matrices of quantities having arbitrary units. For most engineering applications, however, matrices are simply a compact and efficient way of describing families of equations of motion and equilibrium, and collections of data.

Engineering considerations dictate that the terms within an equation be dimensionally consistent. Similarly, consistency of dimensions in large collections of engineering data also must hold. In practical terms, the assumptions made by this model not only have minimal impact on our ability to solve engineering problems with matrices, but requires much less memory than individual storage of units for all matrix elements. Whistle performs dimensional consistency checks (and possible conversion of units types) before proceeding with all matrix operations. All that is required is examination of the row and column matrix units – there is no need to examine consistency of units at the matrix element level.

Matrix Operations. We are building computational support for standard matrix operations (e.g., addition, subtraction, multiplication, solution of linear equations) on physical quantities. For example, the fragment of code:

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
Force    = [ 2 N, 3 N, 4 N ];
Distance = [ 1 m; 2 m; 3 m ];
Work     = Force*Distance;
```

is a simple calculation for the work done by a force moving through a prescribed distance. The output is as follows:

```
Matrix: Force
row/col                  1            2            3
       units             N            N            N
   1           2.00000e+00  3.00000e+00  4.00000e+00

Matrix: Distance
row/col                  1
       units             m
   1           1.00000e+00
   2           2.00000e+00
   3           3.00000e+00

Matrix: Work
row/col                  1
       units           Jou
   1           2.00000e+01
```

Notice that the computation of units for the work done is automatically handled.

Java Bytecode Components. Early versions of the scripting environment [27] were essentially closed and came with a small set of built-in functions (e.g., Max(x,y), Abs (x), Sqrt (x)). Now, users can import references to compiled Java classes accessible in the JVM (Java Virtual Machine), and under certain restrictions, the methods of those classes can become part of the scripting environment. As we will soon see in the case study examples in section V, scripting statements of the form:

```
import className;
```

will dynamically load `className` into the scripting environment at runtime. When a class is loaded, all of the classes it references are loaded too. This class loading pattern happens recursively, until all classes needed are loaded.

This capability means that end-users can use the scripting language to glue computation components together and export heavy-duty computations to external mechanisms, such as Java libraries, or any other libraries to which Java can interface. Because our work has been driven by the simulation needs of energy efficient buildings, we initially had in mind that these classes would represent physical components in the building. However, from a scripting language perspective, whether or not the component represents a physical entity is irrelevant. As such, and as we will see in the case study examples below, components can also be defined for plotting, data modeling, executable statechart behaviors or, in fact, any modeling abstraction that uses physical quantity interfaces.

V. CASE STUDY PROBLEMS

We now demonstrate the capabilities of Whistle by working step by step through five progressively complicated case study problems.

Case Study 1: Parsing a Simple Assignment Statement. The computational platform parses problem specifications into an abstract syntax tree, and then executes the statements by traversing the syntax tree in a well-defined manner. To see how this process works in practice, let's begin by working step by step through the details of processing the assignment statement:

```
x = 2 in;
```

Figure 8 shows the parse tree for this statement.

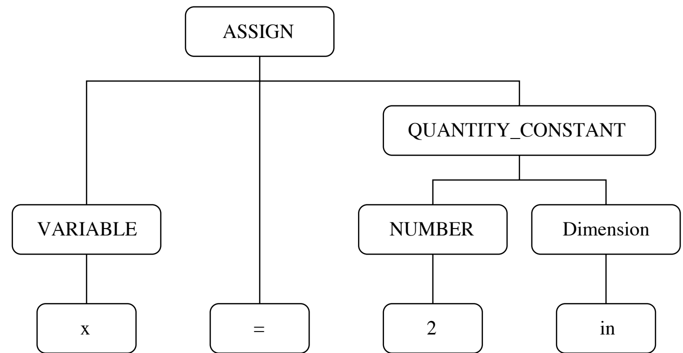


Figure 8. Parse tree for x = 2 in.

The interpreter parses and stores the character sequence "2 in" as the physical quantity two inches. Notice how `2` juxtaposed with `in` implies multiplication; we have hard-coded this interpretation into the scripting language because `2 in` is more customary and easier to read than `2 * in`. This quantity is discarded once the statement has finished executing.

The abstract syntax tree is as follows:

```
Starting PrintAbstractSyntaxTree() ...
========================================== ...

<COMPOUND>
   <ASSIGN>
       <VARIABLE id="x" level="0" />
       <QUANTITY_CONSTANT value="[ 2.000, in]" />
   </ASSIGN>
</COMPOUND>

========================================== ...
Finishing PrintAbstractSyntaxTree() ...
```

Compound statements allow for the modeling of sequences of individual statements. The assignment is defined by two parts, a variable having an identification "x" and a quantity constant having the value 2.0 in.

Internally, the quantity constant is automatically converted to its metric counterpart. Table III shows the name and value of variable "x" as well as details of the units type, scale factor and exponent values.

Case Study 2: Hierarchy of Water Tank Models. The purpose of this example is to see how modules of Java code

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
---------------------------
QUANTITY NAME AND VALUE
---------------------------
Quantity Name  : x
Quantity Value : 0.0508 (m)
----------------------------------------------
UNITS
----------------------------------------------
Units Name     : "in"     Length Exponent : 1
Units Type     : US       Mass Exponent   : 0
Scale Factor   : 0.0254   Time Exponent   : 0
                          Temp Exponent   : 0
                          Radian Exponent : 0
----------------------------------------------
```

TABLE III. SYMBOL TABLE STORAGE FOR QUANTITY x = 2 IN.

can be imported into the scripting language environment and become part of the admissible syntax.

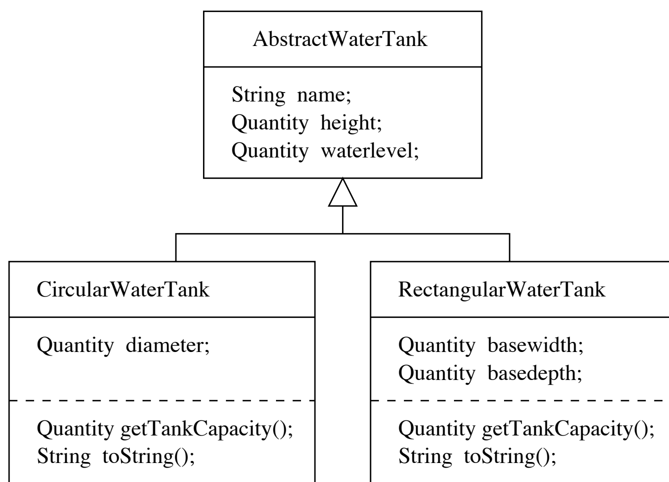


Figure 9. Water tank class hierarchy, annotated with a partial list of variables and methods.

Figure 9 shows a simple class hierarchy for the modeling of water tank components. The AbstractWaterTank class defines concepts and parameters common to all water tanks (e.g., name, waterlevel, height of the tank). The classes RectangularWaterTank and CircularWaterTank add details relevant to tanks with rectangular and circular base areas, respectively. For example, circular water tanks are defined by their diameter. Rectangular water tanks are defined by the parameters basewidth and basedepth. Geometry specific methods are written to compute tank capacities, and so forth.

Now let us assume that the source code for these classes has been compiled in a Java bytecode and references to their specifications are accessible in the JVM (Java Virtual Machine). The fragment of code:

```
import whistle.component.hvac.CircularWaterTank;
```

makes all of the public methods in `CircularWaterTank` and `AbstractWaterTank` available to the library of terms acceptable to the scripting language environment. A circular water tank component with diameter `2 m` and height `2 m` is created by writing:

```
tank01 = CircularWaterTank();
tank01.setDiameter( 2.0 m );
tank01.setHeight( 2 m );
```

The variable `tank01` references an object of type CircularWaterTank stored within the JVM. In a departure from standard programming and scripting languages, which support exchange of basic data types (e.g., float, double) and references to objects in method calls, our philosophy is that participating java classes will work with quantities, matrices of quantities, booleans and strings. Thus, in order to compute and see the tank capacity, we can write:

```
capacity = tank01.getTankCapacity();
print "*** Capacity is: ", capacity;
```

The output is as follows:

```
*** Capacity is: [ 6.283, m^3]
```

Case Study 3: Visualization of Pump Model Data. Pumps (a fan is a pump that moves a gas) are a type of turbomachinery that are generally modeled using empirical data because models based deductively upon first principles of physics can only represent generalized, idealized behavior, not actual specific behavior. Pump performance is difficult to predict because it requires understanding the complex interaction between the pump and the fluid: the shape of the impeller blades, the friction between the blades and the fluid at different temperatures, pressures, and impeller speeds, the details of the pipes and valves upstream and downstream of the pump all have an effect on the pump performance. Manufacturers of pumps create performance curves based on measurements of pumps. The curves show head (pressure), brake horse power, and efficiency as a function of flow rate for a given impeller diameter. The performance of the same pump design with a different impeller diameter, different rotational speed, or different fluid can be calculated from a set of performance curves using the similarity laws. These curves can be used to produce a curve of dimensionless head versus dimensionless flow rate that is more generally useful for incorporation into a modeling program [37], [38].

While the principal purpose of component modeling is for the representation of entities in the physical world, from a scripting perspective, the concept of components extends to services designed to support the analysis and visualization of CPS. To this end, we are in the process of developing data model and visualization components. Figure 10 shows a plot of pump performance data for a size 3, drawthrough 9 inch, BCMpress Fan. Note that the y-axis is dimensionless pressure, where the pressure head is normalized by $\rho * D^2 * N^2$, where ρ is density, D is impeller diameter, and N is rotational speed (rpm). The x-axis is dimensionless flow, where the flow rate

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

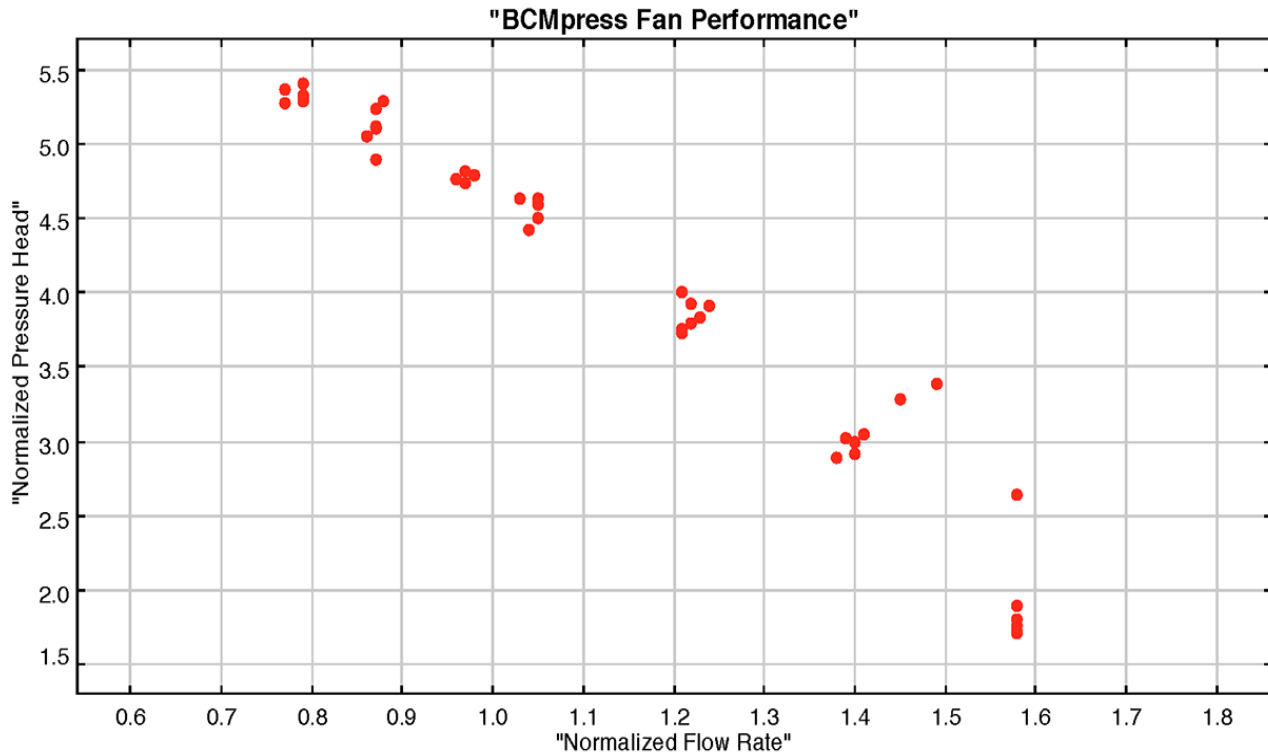


Figure 10. Dimensionless pressure as a function of dimensionless flow of a pump as calculated from standard manufacturer pump curves.

is normalized by $\rho * D^3 * N$. These normalizations are based on idealizations known as "fan laws".

The scripting language specification employs data model and plot components, i.e.,

```
// Import data model from XML file ....

data01 = DataModel();
data01.getData( "pumpModel.xml" );

// Plot pressure head vs discharge rate ...

plot01 = PtPlot();
plot01.setSize( 600, 700 );
plot01.setTitle( "BCMpress Fan Performance");
plot01.setXLabel("Dimensionless Flow Rate Q");
plot01.setYLabel("Dimensionless Pressure Head");

// Transfer data model to plot component ...

c01 = data01.getCurve ("level01");
nsteps = c01.getNoPoints();
for (i = 0; i < nsteps; i = i + 1) {
    plot01.addPoint( c01.getX(i), c01.getY(i) );
}

plot01.display();
```

DataModel() is an experimental component for the storage and management of data models, and their import/export in an xml format. The PtPlot() component is an interface to the PtPlot visualization package distributed with PtolemyII [23].

Case Study 4: Oscillatory Flow between Two Tanks. The language supports the representation of differential equations in their discrete form, and solution via numerical integration techniques.

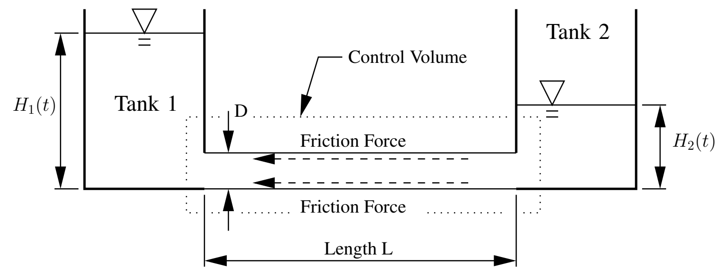


Figure 11. Summary of forces acting on a pipe element connecting two tanks.

Consider, for example, the problem of computing the oscillatory flow of fluid between two tanks as illustrated in Figure 11. Let v(t) and Q(t) be the velocity (m/sec) and flowrate (m^3/sec) in the pipe, measured positive when the flow is from Tank 1 to Tank 2. For a pipe cross section, A_p, and tank cross-section areas A_1 and A_2, conservation of mass implies:

$$Q(t) = A_p v(t) = -A_1 \frac{dH_1(t)}{dt} = A_2 \frac{dH_2(t)}{dt}. \quad (2)$$

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

When water depths $H_1(t) \neq H_2(t)$, this "head" differential will cause fluid to flow through the pipe. Transient behavior of the fluid flow is obtained from the equations of momentum balance in the horizontal direction of the control volume, i.e.,

$$\left[\frac{dv(t)}{dt}\right] + \left[\frac{f_1}{2D}\right] v(t)|v(t)| = \left[\frac{g}{L}\right] [H_1(t) - H_2(t)]. \quad (3)$$

Notice that each term in equation (3) has units of acceleration, and that damping forces work to reduce and overall amplitude of accelerations. Damping forces are proportional to pipe roughness and inversely proportional to pipe diameter. The time-history response is computed by creating discrete forms of equations (2) and (3), and systematically integrating the first-order equations of motion with Euler integration. First, the update for momentum balance is given by:

$$v(t + dt) = v(t) + \left[\frac{dv(t)}{dt}\right] dt. \quad (4)$$

Updates in the water depth for each tank are given by:

$$H_1(t + dt) = H_1(t) - \left[\frac{A_p}{A_1}\right] v(t)dt. \quad (5)$$

and

$$H_2(t + dt) = H_2(t) + \left[\frac{A_p}{A_2}\right] v(t)dt. \quad (6)$$

If the tank and pipe components are defined as follows:

```
// Define tank and pipe components ....

tank01 = RectangularWaterTank();
tank01.setName("Tank 01");
tank01.setHeight( 10 m );
tank01.setBaseWidth( 3 m );
tank01.setBaseDepth( 5 m );
tank01.setWaterLevel( 5 m );

tank02 = RectangularWaterTank();
tank02.setName("Tank 02");
tank02.setHeight( 5 m );
tank02.setBaseWidth( 2.0 m );
tank02.setBaseDepth( 2.5 m );
tank02.setWaterLevel( 1 m );

pipe01 = Pipe();
pipe01.setLength(  5.0 m );
pipe01.setRadius( 10.0 cm );
pipe01.setRoughness( 0.005 );
```

then the script:

```
velFluid  = pRough/(4*pRadius)*velOld*Abs(velOld)*dt;
velUpdate = g/pLength*( h01Old - h02Old )*dt;
velNew    = velOld + velUpdate - velFluid;
```

shows the essential details of computing the fluid velocity update with Euler integration. During the executable phases of simulation (right-hand side of Figure 4), the runtime interpreter checks for dimensional consistency of terms in statements before proceeding with their evaluation. Figures 13 and 14 are plots of the tank water levels (m) versus time (sec), and volumetric flow rate (m^3/sec) versus time (sec), respectively.

Case Study 5: Tank with Water Supply and Shut-off Valve. This example, adapted from Turns [39], illustrates the steady and transient states of mass conservation and control volume of a tank with a shut-off valve and water supply system.

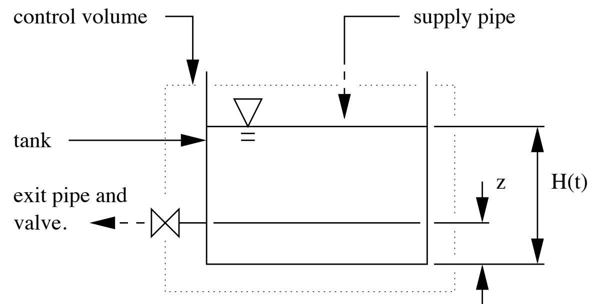


Figure 12. Front elevation of tank, supply pipe, and exit pipe and valve.

The system behavior corresponds to four states as follows: (I) The tank is empty, (II) The tank is being filled to a depth of 1 m, (III) The shut-off valve is opened and the water level is decreasing, (IV) The water level in the tank reaches a steady state and does not change. Based on conservation of mass for an unsteady filling process, we obtain the change in water level from equation (7),

$$\left[\frac{dH(t)}{dt}\right] \rho A_t = \rho v_1 A_1, \quad (7)$$

where $H(t)$ is water height in the tank in (m), ρ is water density and is equal to 997 (kg/m^3), A_t is cross-section area of the tank in (m^2), A_1 is cross-section area of supply pipe in (m^2), v_1 is average velocity of inlet water in (m/sec). When the water height is 1 m, the shut-off valve opens and the height of water in the tank will be updated based on equations:

$$\left[\frac{dH(t)}{dt}\right] \rho A_t = \dot{m}_1 - \dot{m}_2, \quad (8)$$

where $\dot{m}_1$ and $\dot{m}_2$ are the instantaneous mass flow of inlet and outlet pipes in (kg/s):

$$\dot{m}_2 = \rho v_2 A_2, \quad (9)$$

where A_2 is the cross-section area of the outlet pipe in (m^2):

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

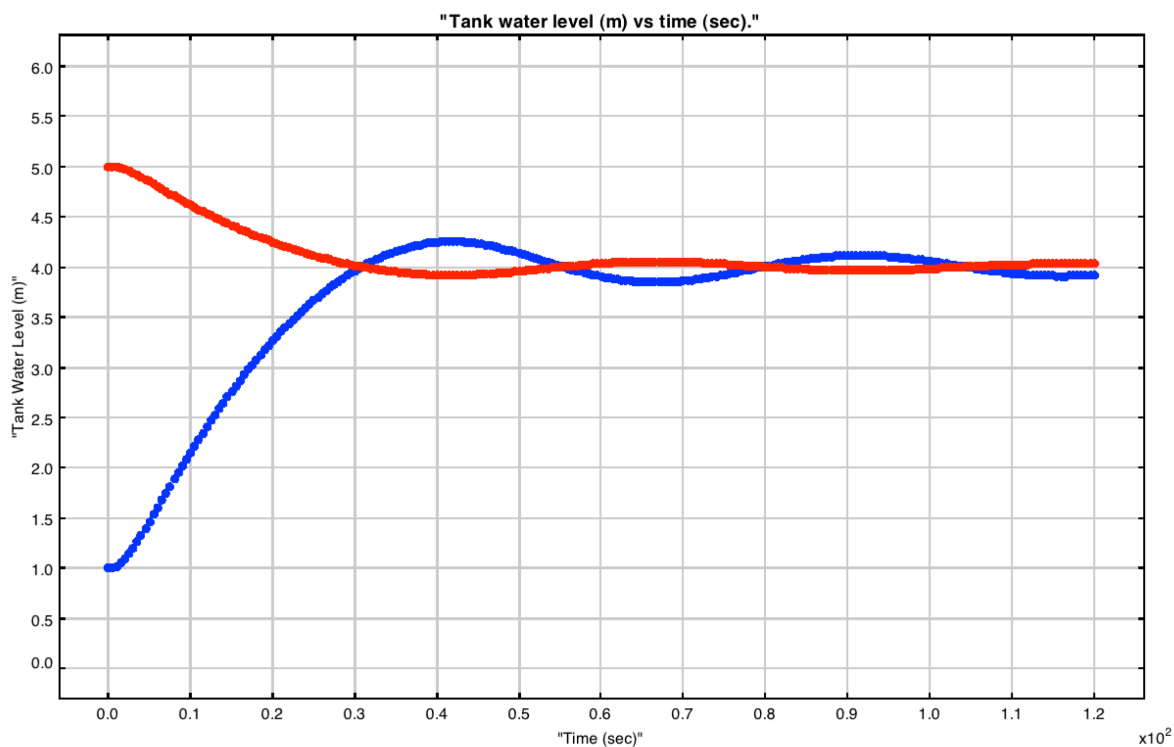


Figure 13. Tank water levels (m) versus time (sec).

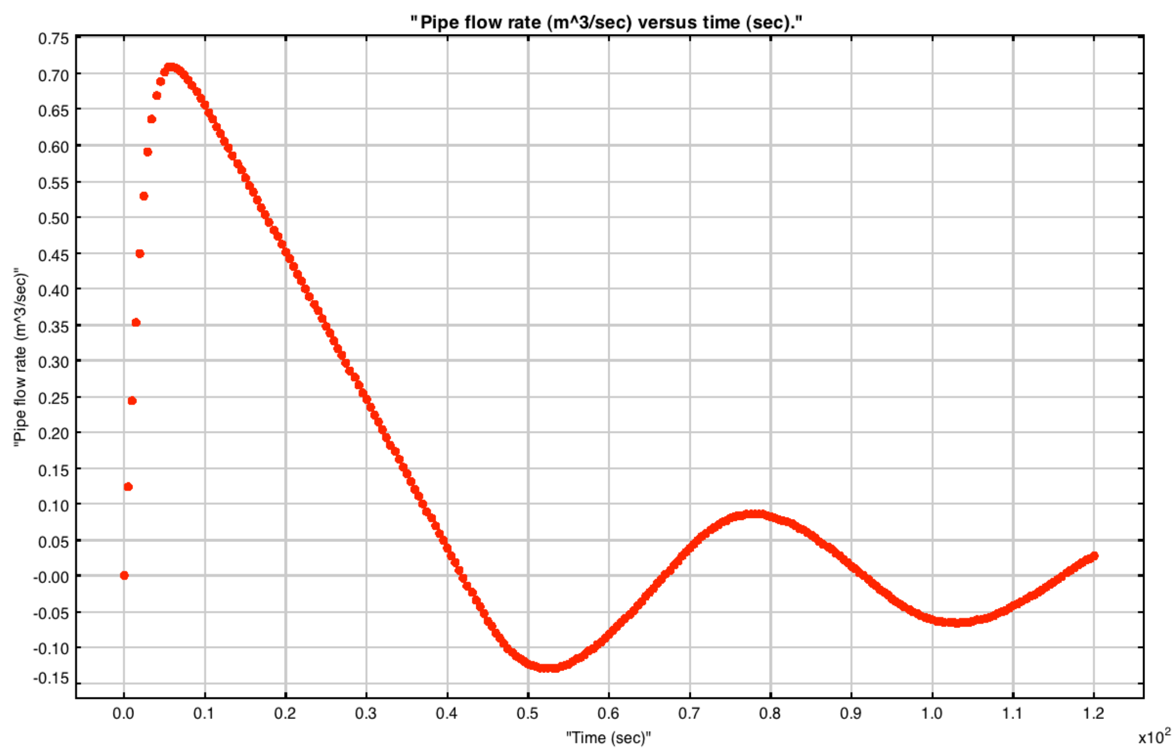


Figure 14. Volumetric flow rate (m^3/sec) versus time (sec).

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

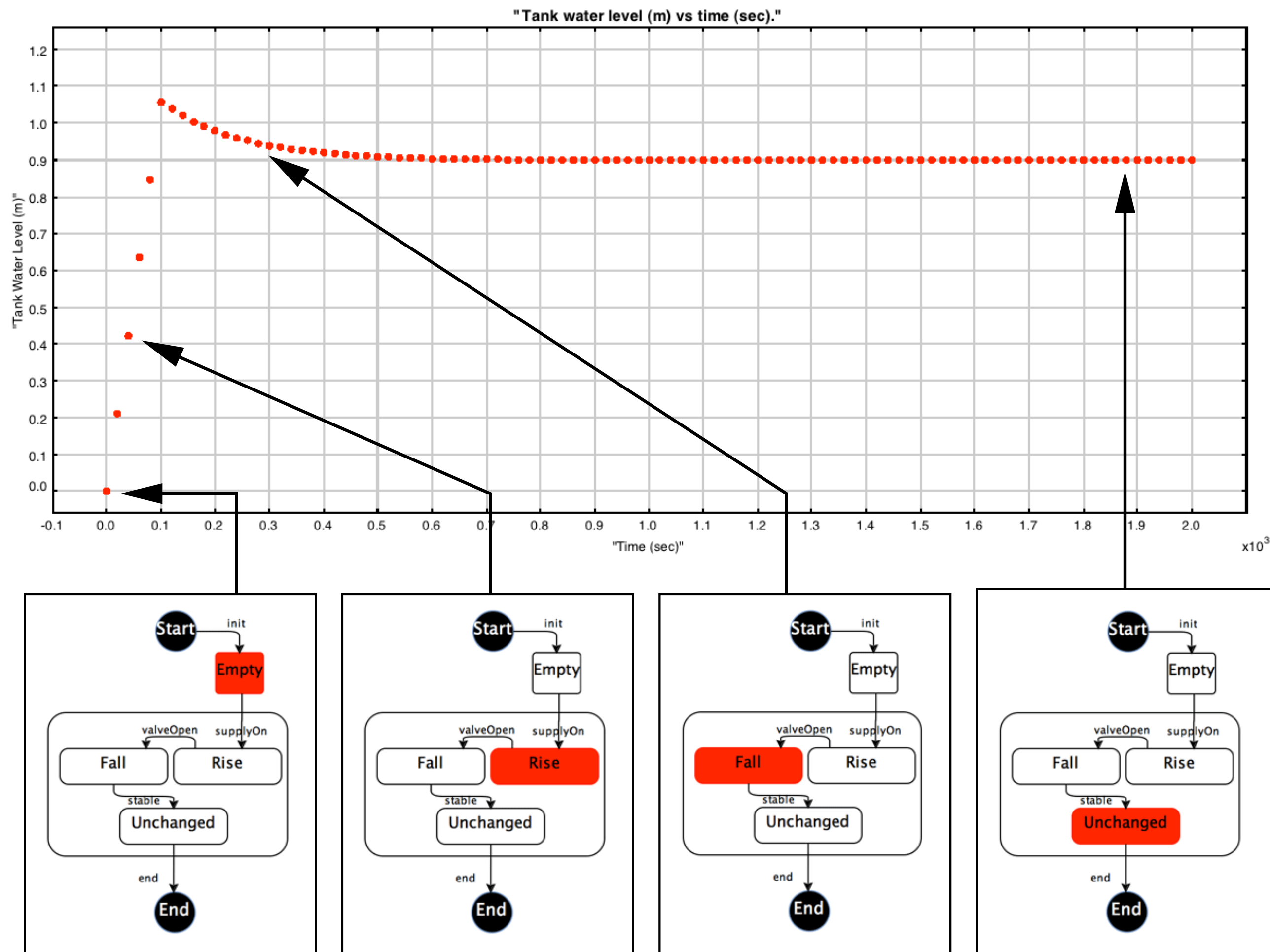


Figure 15. Time-history response for a tank having a water supply and shut-off valve. Upper plot: tank water level (m) versus time (sec). Lower plot: discrete statechart behaviors at various points in the time-history response.

$$\dot{m}_1 = \rho v_1 A_1, \quad (10)$$

$$v_2(t) = 0.85\sqrt{g\,(H(t) - z)}, \quad (11)$$

where $v(t)$ is outlet velocity in (m/s) and z is the location of the shut-off valve in (m). In order to mimic the physical equations, we used the scripting language to model components of the tank, supply, and exit pipes with their associated parameters.

The fragment of script below illustrates the essential details of defining the circular water tank and pipe components:

```
// Define tank and pipe components ....

tank01 = CircularWaterTank();
tank01.setName("Tank 01");
tank01.setDiameter( 1*0.15 m);

// Define supply pipe ....

pipe01 = Pipe();
pipe01.setRadius( 10.0 mm );
```

The heart of the time-history simulation is a looping construct that contains two cases (or discrete states) for physical behavior:

```
// Case 1: Water level is below 1 m:

DepthUpdate =  pipe1Velocity * pipe1Area*dt / tankArea;
DepthNew    =  DepthOld + DepthUpdate;
response01 [i][0] = i * dt;
response01 [i][1] = DepthNew;
DepthOld = DepthNew;

// Case 2: Water level is above 1 m:

massFRSupplyPipe = rho*pipe1Velocity * pipe1Area;

velocityExit   = 0.85*Sqrt(g*(DepthOld - 0.1 m));
massFRExitPipe = rho* velocityExit*pipe02.getArea();

massFlowRateCV = massFRSupplyPipe - massFRExitPipe;

dHeight  = massFRCV/(rho*tankArea)*dt;
DepthNew = DepthOld + dHeight;
response01 [i][0] = i * dt;
```

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
response01 [i][1] = DepthNew;
DepthOld = DepthNew;
```

Figure 15 shows the time-history response of the water level in the tank as it transitions from an empty tank to steady state where the water level remains unchanged t height of 0.9 m. In order to visualize the discrete behavior of this system, we employ our previously developed executable statechart package [10]. This package is capable of modeling and implementation for event-driven behavior with finite state machines. It supports modeling for: (1) Simple, hierarchical and concurrent states, start and final states, (2) History and deep-history pseudostates in hierarchical states, (3) Fork and join pseudostates for concurrent states, (4) Segmented transitions using junction points, and (5) Events, guards and actions for transitions. Visualization of the statechart behaviors is supported through use of mxGraphics in our code. The MVC design pattern (see Section II) is used to make views come alive as models transition through a sequence of states. The abbreviated script:

```
import whistle.statechart.TankStatechart;

....
statechart = TankStatechart();
statechart.startStatechart();
statechart.TransitionEvent(init);

if( DepthOld >= 1 m ){
  statechart.TransitionEvent(valveOpen);
  ....
}
....
```

shows how a statechart element for the water tank is created in an input file developed by the scripting language, and how the language is capable of triggering an event to the statechart when the water level exceeds 1 m. The bottom level of Figure 15 shows how different regions of continuous behavior correspond to the discrete states in the tank statechart.

VI. Conclusion

The purposes of this paper have been two-fold: (1) to describe a semantic platform infrastructure for the model-based systems engineering of cyber-physical systems, and (2) to describe a new and novel scripting language called Whistle. Both efforts are a work in progress. The proposed semantic platform infrastructure will enhance systems engineering practice by lowering validation costs (through rule checking early in design) and providing support for performance assessment during system operation. Our focus in this paper has been to describe the basic features of Whistle, and to show how it can be used to simulate the behavior of a variety of systems characterized by fluid flows and simple control.

Our plans for the future are to conduct research in scripting language design and computational modeling so that Whistle provides the CPS modeling infrastructure and systems integration glue needed to implement the vision of Figures 3 through 5. We envision cyber-physical systems having behaviors that are both distributed and concurrent, and defined by mixtures of local- and global- rule-based control. For the time-history behavior modeling and control of energy-efficient buildings, the finite element method is attractive because problem solutions (e.g., spatial distributions of temperature and pressure in large enclosures) can be formulated from first principles of engineering such as momentum balance. Solution procedures need to be robust, scalable, and extensible to energy-balance calculations. We will design a family of component model interfaces (see the left-hand side of Figure 4), extend them for the implementation of a build components library (e.g., tanks, pipes, valves) and where needed, participate in finite element analysis, actuation, and control. In order for modeling procedures to be efficient we need mechanisms that take advantage of the natural hierarchy of physical systems. Engineers should be provided with the capability to position sensors inside water tanks, and then connect tanks together with networks of pipes and pumps. At the same time, we also need a capability for components to communicate across hierarchies, and we are proposing this be accomplished with listener mechanisms (e.g., a controller component might listen for data from a collection of sensor components and then depending on the water level reading, take an appropriate action). The keys to making this work are software interfaces designed to support a multitude of system viewpoints (e.g., a visualization interface for 2D- and 3D- visualization, a finite element interface for the description of element-level behaviors cast in a matrix format, a communications interface for sensor to controller communication) and Whistle's feature to import and work with references to compiled bytecodes in the Java Virtual Machine. Whistle will act as the glue for systems integration and access to procedures for simulation, visualization and system assessment.

Acknowledgment

The work reported here is part of a US National Institute of Standards and Technology (NIST) funded program dedicated to the development of standards for CPS design, modeling, construction, verification and validation.

References

[1] P. Delgoshaei, M.A. Austin, and D.A. Veronica, "A semantic platform infrastructure for requirements traceability and system assessment," The Ninth International Conference on Systems (ICONS 2014), February 2014, pp. 215-219, ISBN: 978-1-61208-319-3.

[2] NIST. "Strategic R&D opportunities for 21st Cyber-physical systems: connecting computer and information systems with the physical world," National Institute of Science and Technology(NIST), Gaithersburg, MD, USA, 2013.

[3] J. Wing, "Cyber-physical systems research challenges," National Workshop on High-Confidence Automotive Cyber-Physical Systems, Troy, MI, USA, 2008.

[4] M.A. Austin,V. Mayank, and N. Shmunis, "Ontology-based validation of connectivity relationships in a home theater system," The 21st International Journal of Intelligent Systems, Vol. 21, No. 10, pp. 1111–1125, October 2006.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[5] M.A. Austin, V. Mayank, and N. Shmunis, “PaladinRM: graph-based visualization of requirements organized for team-based design,” Systems Engineering: The Journal of the International Council on Systems Engineering, Vol. 9, No. 2, pp. 129–145, May 2006.

[6] N. Nassar and M.A. Austin, “Model-based systems engineering design and trade-off analysis with RDF graphs,” 11th Annual Conference on Systems Engineering Research (CSER 2013), Georgia Institute of Technology, Atlanta, GA, March 19-22, 2013, pp. 216–225, doi:10.1016/j.procs.

[7] S. Fridenthal, A. Moore, and R. Steiner, “A practical guide to SysML,” MK-OMG, 2008.

[8] P. Delgoshaei and M.A. Austin, “Software design patterns for ontology-enabled traceability,” Conference on Systems Engineering Research (CSER 2011), Redondo Beach, Los Angeles, April 15-16, 2011.

[9] P. Delgoshaei and M.A. Austin, “Software patterns for traceability of requirements to finite-state machine behavior: application to rail transit systems design and management,” The 22nd Annual International Symposium of The International Council on Systems Engineering (INCOSE 2012), Rome, Italy, 2012, pp. 2141-2155, DOI: 10.1002/j.2334-5837.2012.tb01463.x.

[10] P. Delgoshaei, “Software patterns for traceability of requirements to state machine behavior,” M.S. Thesis in Systems Engineering, University of Maryland, College Park, MD 20742, November 2012.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: elements of reusable object-oriented software,” Addison-Wesley Professional Computing Series, 1995.

[12] OWL w3 See http://www.w3.org/TR/owl-features/ (Accessed, February 2004).

[13] 2013. *Apache Jena*, accessible at: http://www.jena.apache.org.

[14] R. Eckstein, “Java SE application design with MVC,” Sun Microsystems, 2007. For more information, see http://www.oracle.com/technetwork/articles/javase/index-142890 (Accessed, November 2014).

[15] M.H. Qusay, “Getting started with the Java rule engine API (JSR 94): toward rule-based applications,” Sun Microsystems, 2005. For more information, see http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html (Accessed, March 10, 2008).

[16] G. Rudolf, “Some guidelines for deciding whether to use a rules engine,” 2003, Sandia National Labs. For more information see http://herzberg.ca.sandia.gov/guidelines.shtml (Accessed, March 10, 2008).

[17] T. Berners-Lee, J. Hendler, and O. Lassa, “The semantic web,” Scientific American, pp. 35–43, May 2001.

[18] P. Derler, E.A Lee, and A.S. Sangiovanni-Vincentelli, “Modeling cyberphysical systems,” Proceedings of the IEEE, 100, January 2012.

[19] D. Macpherson and M. Raymond, “Ontology across building, emergency, and energy standards,” The Building Service Performance Project, Ontology Summit, 2009.

[20] R. Koelle and W. Strijland, “Semantic driven security assurance for system engineering in SESAR/NextGen,” In Integrated Communications, Navigation and Surveillance Conference (ICNS), 2013, pp. k2-1–k2-12.

[21] P. Fritzson, “Principles of Object-Oriented modeling and simulation with Modelica 2.1,” Wiley-IEEE Press, 2003.

[22] E.A. Lee, “Finite state machines and models in Ptolemy II,” Technical report, EECS Department, University of California, Berkeley, 2009. For more information, see http://ptolemy.eecs.berkeley.edu/ptolemyII (Accessed, August 1, 2014).

[23] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, “Heterogeneous concurrent modeling and design in Java (volume 1: introduction to Ptolemy II),” Technical Report ECB/EECS-2008-28, Department Electrical Engineering and Computer Sciences, University of California, Berkeley, CA, April 2008.

[24] J. Lin, S. Sedigh, and A. Miller, “A semantic agent framework for cyber-physical systems,” Semantic agent systems studies in computational intelligence, Vol. 344, pp. 189-213, 2011.

[25] G. Simko, D. Lindecker, T. Levendovszky, S. Neema, and J. Sztipanovits, “Specification of cyber-physical components with formal semantics integration and composition,” The 16th ACM-IEEE International Conference on Model Driven Engineering Languages and Systems, 2013.

[26] M.A. Austin, X.G. Chen, and W.J. Lin, “ALADDIN: A computational toolkit for interactive engineering matrix and finite element analysis,” Technical Research Report TR 95-74, Institute for Systems Research, College Park, MD 20742, August 1995.

[27] M.A. Austin, W.J. Lin, and X.G. Chen, “Structural matrix computations with units,” Journal of Computing in Civil Engineering, ASCE, Vol.14, No. 3, pp. 174–182, July 2000.

[28] M.A. Austin, “Matrix and finite element stack machines for structural engineering computations with units,” Advances in Engineering Software, Vol. 37, No. 8, pp. 544–559, August 2006.

[29] J.K. Osterhout, “Tcl and the Tk Toolkit,” Addison-Wesley Professional Computing Series, Reading, MA 01867, 1994.

[30] L. Wall, T. Christiansen, and R. Schwartz, “Programming Perl,” O’Reilly and Associates, Sebastopol, CA 95472, 2nd edition, 1993.

[31] R.L. Schwartz, T. Phoenix, and B.D Foy, “Learning Perl,” O’Reilly and Associates, Sebastopol, CA 95472, 4th edition, July 2005.

[32] J. Ousterhout, “Scripting: higher level programming for the 21st century,” IEEE Computer Magazine, March 1998.

[33] JFlex -The fast scanner generator for Java: See http://jflex.de/, (Accessed: August 1, 2013).

[34] Berkeley Yacc: See http://invisible-island.net/byacc/, (Accessed: August 1, 2013).

[35] R. Mak, “Writing compilers and interpreters: a software engineering approach (Third Edition),” Wiley Publishing Inc, 2009.

[36] “Unit Conversion Guide,” “Fuels and petrochemical division of AICHE,” 1990.

[37] F.M. White, ”Fluid mechanics (4th Edition),” McGraw-Hill, 1999.

[38] J. Wright, “Building performance simulation for design and optimization,” chapter HVAC systems performance and prediction, pp. 312–340, Spon Press (an imprint of Taylor & Francis), London and New York, 2010.

[39] S.R. Turns, “Thermal-fluid sciences: an integrated approach,” Cambridge University Press, 2006.

2014, © Copyright by authors, Published under agreement with IARIA - www.iaria.org