

## ABSTRACT

Scholarly Paper: **COMPUTATIONAL MODELING FOR  
CITY SYSTEM-OF-SYSTEMS (SoS)  
SIMULATION AND MANAGEMENT**

Alan Nguyen, Master of Science, 2014

Directed by: Associate Professor Mark Austin  
Department of Civil and Environmental Engineering  
and ISR

**Abstract.** With 70% of the world's population expected to live in cities by 2050, there is an increasing need to model the inter-system behavior of these cities since outcomes in one system can strongly affect other systems, for better or worse. The purpose of this paper is to create a higraph-based framework for modeling the relationships and behavior between different city systems. A systems engineering approach was used to design the structure and behavior for the framework. The framework is implemented in Java using several software design patterns, and its capabilities are exercised by simulating the effect of a weather event on a city. Finally, the paper explores the use of ontologies and rule-based modeling to represent cities. Overall, the model allows for different viewpoints and levels of abstraction, as well as identification of cross-cutting issues and cascading effects through the city.

**Last Modified:** May 3, 2014

COMPUTATIONAL MODELING FOR CITY  
SYSTEM-OF-SYSTEMS (SoS)  
SIMULATION AND MANAGEMENT

by

Alan Nguyen

Scholarly Paper  
Master of Science in Systems Engineering  
2014

Advisory Committee:

Associate Professor Mark Austin, Chair/Advisor

Second Reader: Associate Professor David Lovell

## Acknowledgments

This research was completed with the collaboration of Dr. Mark Austin, James Vaughn, Parastoo Delgoshaei, and Leonard Petnga.

# Table of Contents

List of Figures	v
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	1
1.2 Scope and Objectives . . . . .	4
1.3 Background and Previous Work . . . . .	4
1.3.1 System-of-Systems (SoS) Modeling . . . . .	4
1.3.2 Modeling with Higraphs . . . . .	9
1.4 Scope and Objectives . . . . .	12
<b>2 Framework for Behavior Modeling of Cities</b>	<b>13</b>
2.1 Framework . . . . .	13
2.2 Framework Behavior . . . . .	16
2.3 High-Level Example Model of a City . . . . .	18
<b>3 Implementation of Framework in Java</b>	<b>28</b>
3.1 Framework Design Patterns . . . . .	28
3.2 Model Architecture . . . . .	35
3.2.1 Message Passing . . . . .	35
3.2.2 Components . . . . .	35
3.2.3 Rule Processing . . . . .	37
3.3 Implementing the City Model in Java . . . . .	38
3.4 Message Passing using JGraphT . . . . .	45
3.4.1 Graph Structure . . . . .	45
3.4.2 Traversing and Processing the Graph . . . . .	47
3.4.2.1 Traversing a Graph . . . . .	47
3.4.2.2 Processing the Graph . . . . .	48
3.5 Example: Revisiting the City . . . . .	48
3.6 Summary . . . . .	57
<b>4 Implementation of Framework using Ontology and Rule-Based Modeling</b>	<b>58</b>
4.1 Description of Ontologies and RDF Graphs . . . . .	58
4.2 Refactoring Framework to Jena . . . . .	60
4.2.1 Mapping Framework Concepts to Ontology-Based Concepts . . . . .	60
4.2.2 Creating Ontologies . . . . .	61
4.2.3 Defining Rules . . . . .	62
4.3 Implementing the City Model using Jena . . . . .	63
<b>5 Conclusions and Future Work</b>	<b>66</b>
5.1 Conclusions . . . . .	66
5.2 Future Work . . . . .	68
Bibliography	69

Appendix A: Java Framework Model Setup	71
Appendix B: Transportation Ontology Rules	75
Appendix C: City Network Modeling	77

## List of Figures

1.1	Network-of-Networks Model (Source: Tonn [21]). . . . .	7
1.2	An example mathematical graph. . . . .	8
1.3	An example of a Venn Diagram. . . . .	8
1.4	An example higraph. . . . .	11
2.1	Framework Metamodel. . . . .	14
2.2	A Message passing Activity Diagram. . . . .	17
2.3	Example of Framework: A Simple City. . . . .	19
2.4	Cascading failure, Step 1. . . . .	23
2.5	Cascading failure, Step 2. . . . .	24
2.6	Cascading failure, Step 3. . . . .	25
2.7	Cascading failure, Step 4. . . . .	26
2.8	Cascading failure, Step 5. . . . .	27
3.1	Class diagram for the component hierarchy design pattern. . . . .	29
3.2	Port Architecture. . . . .	30
3.3	Wire Implementation . . . . .	30
3.4	Higraph Version 1 - One MetaComponent, One BaseComponent . . .	31
3.5	External/Internal Ports. . . . .	32
3.6	Version 2 - MetaComponent feeding data to mcSource. . . . .	33
3.7	Version 3 - MetaComponent wrapping around Version 2. . . . .	34
3.8	SoS Framework Architecture. . . . .	36
3.9	Northeast section of the Washington D.C. Metro System. . . . .	39
3.10	Visual Representation of City model in Java. . . . .	40
3.11	Visual Representation of Simplified City Model. . . . .	50
3.12	High-level city graph. . . . .	50
3.13	Transportation-level graph (embedded inside the city-level graph). . .	51
3.14	Cluster-level graph (embedded inside the transportation-level graph). .	53
3.15	Controller-level graph. . . . .	54
3.16	Three-station Cluster-Level Graph. . . . .	55
4.1	Example of RDF triple where node A is a subject, predicate is a verb, and node B is an object. . . . .	59
4.2	Transportation Ontology, created using Protégé [16] . . . . .	63

## Chapter 1

### **Introduction**

#### 1.1 Problem Statement

During the past 200 years the World's population has sky-rocketed! Current projections are for a steady increase in population growth from a little more than 7 billion people today to 9.5 – 10.5 billion by 2050 [25]. These increases will be accompanied by a continuation of rural-to-urban drift, a trend where people migrate to cities to because of the opportunities they afford for education, work and long-term prosperity. Today, 82 percent of the US population lives in urban areas; by 2050, this is expected to increase to 90 percent. Not only are cities destined to become larger and more diversified, but the increased demand for limited resources – sometimes very limited resources – will cause cities to become more complex and perhaps less resilient. Today cities are responsible for two thirds of the energy used, 60 percent of all water consumed and 70 percent of all greenhouse gases produced. For many of the world's large cities, a major obstacle to long-term sustainability is a lack of funds to pay for improvements to aging urban infrastructure [6].

To avoid a long-term decline in the standard of living for city residents, future city managers will need to be smart in their utilization of resources and in “decision making” to support the management of city systems. Present-day thinking, partic-

ularly at technology companies who have a vested interest in seeing this happen, is that wireless communication and geographic information systems technologies can provide a backbone infrastructure upon which new decision making tools can be developed for managing urban congestion, maximizing energy efficiency, enhancing public safety and providing access to information. This stove-piped view may be over-simplified. A key problem is that urban areas are complex networks of intertwined networks, with many entities having a multiplicity of functions and network memberships. A simple example of the latter is the Fort Totten Metro Station, which is on the red and green lines – recently, it was also added to the yellow line – in the Washington DC Metro System. The behavior modeling problem is complicated by mixtures of autonomous and dependent subsystem-level behaviors. For example, the behavior of a train operating on the Washington DC Metro system is partially autonomous and partially controlled, subject to operational and scheduling constraints. At a higher level, transportation networks are dependent on electrical networks which, in turn, can be affected by environmental systems. Finally, cities do not have the luxury of being designed from scratch. This means that nearly all improvements need to occur within the structural and behavioral context of an existing city infrastructure and its many concurrent behaviors. Understanding and managing these interactions requires a system-of-systems perspective. If connections among the networks can be modeled and monitored, then patterns of usage can lead to the design of strategies for short- and long-term management.

The challenge in understanding how cities work is complicated by the many



types of subsystem and network behaviors within a city. Some of these behaviors interact on a regular basis and have been engineered to occur in a pre-defined and predictable manner. Others interact only as the consequence of extenuating circumstances (e.g., extreme weather events) and in ways that are not always easily identifiable. An event in one system can cause cascading effects throughout the city, affecting systems that may not be directly related to the original system. For example, Hurricane Katrina caused massive damage to the New Orleans levee system, which had predictable effects on the transportation, electrical, as well as many other systems. However, many social, political, and economic issues were overlooked due to Katrina. With the transportation and electrical system being down, citizens were not able to evacuate and supplies were unable to be brought in to the city. This caused widespread civil disobedience, which in turn caused political unrest [2, 5, 20].

Another difficulty of modeling cities is predicting the emergent properties that occur due to the integration of multiple systems. Emergence is the way in which complex systems and patterns arise out of a multiplicity of relatively simple interactions. An emergent property of a system is one that can only be assessed by looking at the system as a whole. From a city planning perspective, we would like to engineer positive emergent properties. For example, the introduction of a single bank to a neighborhood might spawn new business development. Conversely, allowing a neighborhood to become run-down can quickly lead to negative social effects.

## 1.2 Scope and Objectives

Solving complex issues such as the integration of different city systems requires an understanding of the behavior and relationships between each system. The purpose of this paper is to create a framework for modeling the relationships and behaviors of a system of systems. The framework will be based on system-of-systems concepts as well as concepts from higraphs. The framework is designed to allow users to create groups of components with similar characteristics. Therefore, multiple viewpoints of the system can be created and cross-cutting issues can easily be identified. Another goal of the model is to simulate the effect of an event on a system and capture the cascading effects of the event throughout every aspect of the system-of-systems. In order to facilitate simulation, the framework will be implemented in Java using a variety of software design patterns. Finally, the framework will be implemented using ontologies, resource description framework (RDF) graphs, and rule-based modeling. In each implementation of the framework, a simplified model of a city will be used as a proof-of-concept to demonstrate how data flows throughout the model.

## 1.3 Background and Previous Work

### 1.3.1 System-of-Systems (SoS) Modeling

A system of systems (SoS) is a collection of independently operational systems that work together and achieve further functionality and behavior [3]. The

component systems operate for their own purposes, however they also act collaboratively with other component systems to fulfill the purposes of the entire system. Because of these characteristics, SoS have two unique properties [18]:

- **Emergence:** Properties that belong to the entire system, but are not prevalent in any component system.
- **Evolution:** The overall system will change over time as component systems malfunction and are replaced.

Many problems today can be cast in a system of systems framework. For example, managing cascading network failures, understanding the mechanisms that control the spread of disease, and homeland security can all be thought of as a SoS. Many SoS problems can be defined as a network of networks, which can then be mathematically analyzed [4]. Each component system can be described using a simple network model of nodes and arcs. Nodes can represent a component or a process that passes information or resources to other components. Arcs represent the data flow between nodes as well as between different component systems. An example of an SoS application is sustainable systems development. Sustainability does not only consider the environmental impacts, but also the social and economic impacts of the system. In this sense, it is important to understand the behavior between the social, economic, and environmental dimensions of sustainability. For systems having behavior that is part discrete and part continuous (i.e., defined by solutions to families of ordinary and partial differential equations), behavior can be mathe-

matically analyzed using hybrid system formulations. A standard technique is to embed continuous behavior inside a network node.

As technology becomes more complex and more prevalent in society, the need to model and understand system of systems is increasingly important in order to pinpoint the fundamental cause of issues and the effects of a failure in one component system. As an example, a broken piece of foam was blamed for the Challenger shuttle disaster. However, after a deeper investigation found that the organizational and safety culture as well as budget and schedule constraints also played major roles in the failure. Although these deeper issues are not apparent as the physical cause of the accident, it is important to identify and address them, since a failure in the safety culture at NASA can cascade to a physical failure of a space shuttle.

Tonns research was based on SoS modeling and its application to city modeling [21]. She defines a city as a network of networks, where cities are comprised of layers. Components within each layer are connected together with an arc, which transfer “physical or virtual processes or interdependencies and allow physical, information, or influence flows.” Components from one layer can also be connected to components of another layer. Figure 1.1 represents a visualization of Tonns model. The strength of this paper lies in the definitions for modeling the behavior of a city. Nodes and arcs are well defined, and the transfer of information between nodes is clearly stated. However, city systems are defined at a high level of abstraction, which makes it difficult to pinpoint exactly where failures occur. Another issue is that the city is modeled from a single point of view. As such, it is difficult to see

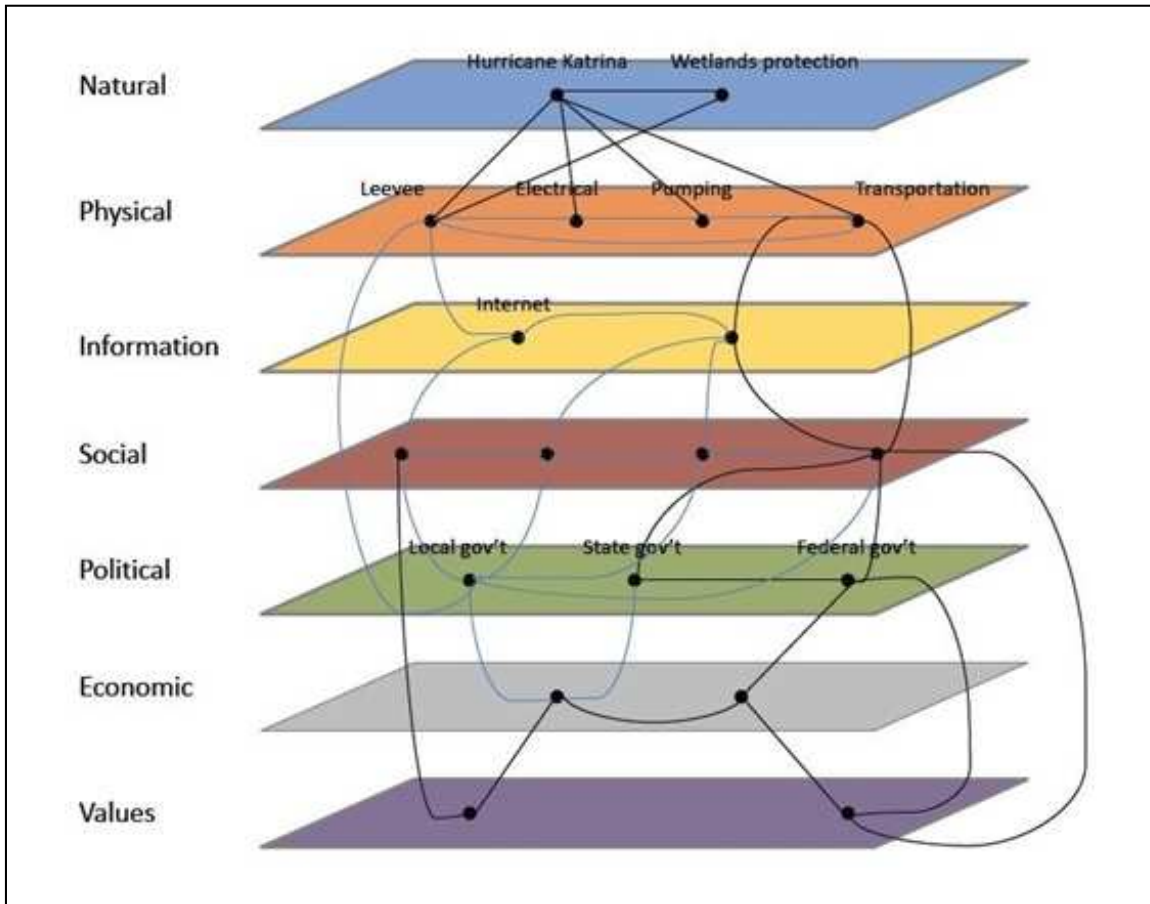


Figure 1.1: Network-of-Networks Model (Source: Tonn [21]).

how changes to one aspect of the city structure or behavior will affect the same components viewed from a different perspective. Finally, the model does not detail how a node changes states and how a state change will affect the rest of the city. In an effort to overcome these limitations, our model looks to build upon Tonn's work by allowing for user-defined groups and defining a formal method for passing information between components.

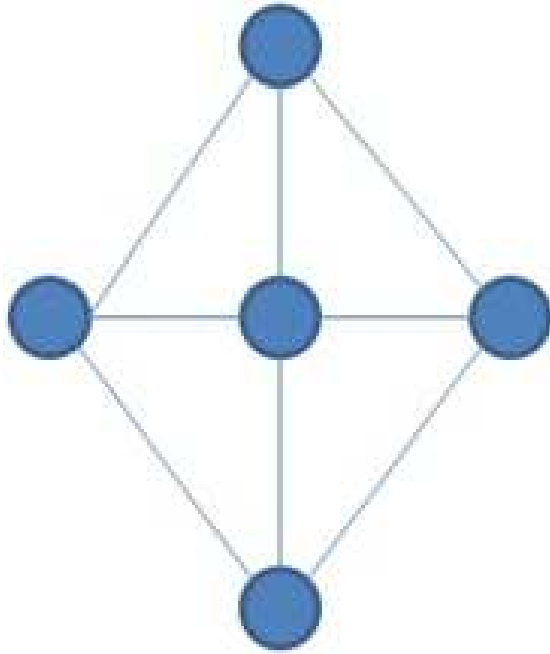


Figure 1.2: An example mathematical graph.

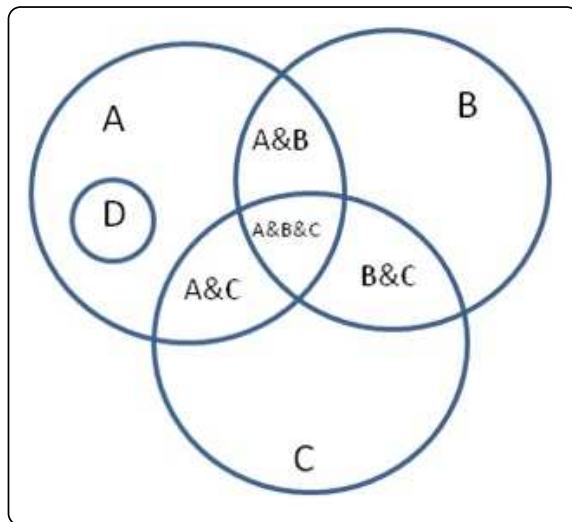


Figure 1.3: An example of a Venn Diagram.

### 1.3.2 Modeling with Higraphs

A higraph is defined as a mathematical graph that combines elements of a standard mathematical graph with elements from Euler/Venn diagrams [11]. To date, the higraph formalism has been applied to a wide range of applications including statecharts in UML (Unified Modeling Language) [22], expression of relationships in drawings [19] and urban forms [7], formal specifications in software development [15, 17], component-based development of web applications [24], and verification procedures in rule-based expert systems [17]. Higraphs have also made their way into Headway Software’s reView, a tool for management of large software code-bases (the source code, libraries, packages, etc..) [12].

**Understanding Higraphs.** To begin understanding the structure of higraphs, we define a standard mathematical graph as a collection of nodes and edges. Figure 1.2 is an example of a graph. This graph is generic in the sense that it has no real meaning until someone defines what the nodes and edges represent. However, we are able to infer that there is a relationship between two nodes that are connected by an edge. Once the relationship is defined, graphs can be an important tool in Systems Engineering.

Venn diagrams show the logical relations between a finite number of sets. More importantly, they define elements that belong to a certain set, elements that are excluded from a set, and elements that belong to multiple sets [23]. Figure 1.3 is an example of a Venn diagram.

The circles in a Venn diagram define what elements belong and do not belong in a certain set. Intersections between circles show which elements belong to each of the intersecting circles. For example, elements found in the region "A" & "B" belong to sets A and B, but not set C. In addition, sets such as set D can be fully enclosed in another set, creating a natural hierarchy. Elements in set D also belong in set A.

Higraphs combine the notion of Venn diagrams and graphs to extend a mathematical graph and allows for depth and orthogonality. The most basic element of a higraph is what Harel refers to as a blob [11]. A blob is essentially a node in the graph and can contain other blobs. From this point forward, we will define a blob with no sub-elements as a node and a blob that contains sub-elements as a cluster. Edges in a higraph can connect any node or cluster together to represent a relationship between them. Edges can be directed implying a one-way relationship or undirected, implying a symbiotic relationship between elements.

Figure 1.4 shows a visual representation of a higraph and highlights interesting facts about them:

1. Nodes can belong to multiple clusters (e.g., a metro station can belong to multiple transportation lines).
2. Single nodes can affect clusters. For example, node l has an arc pointing to cluster c, meaning it will affect all nodes within cluster c.
3. Clusters can affect other clusters. For example, any node in cluster b will affect all nodes in cluster j.



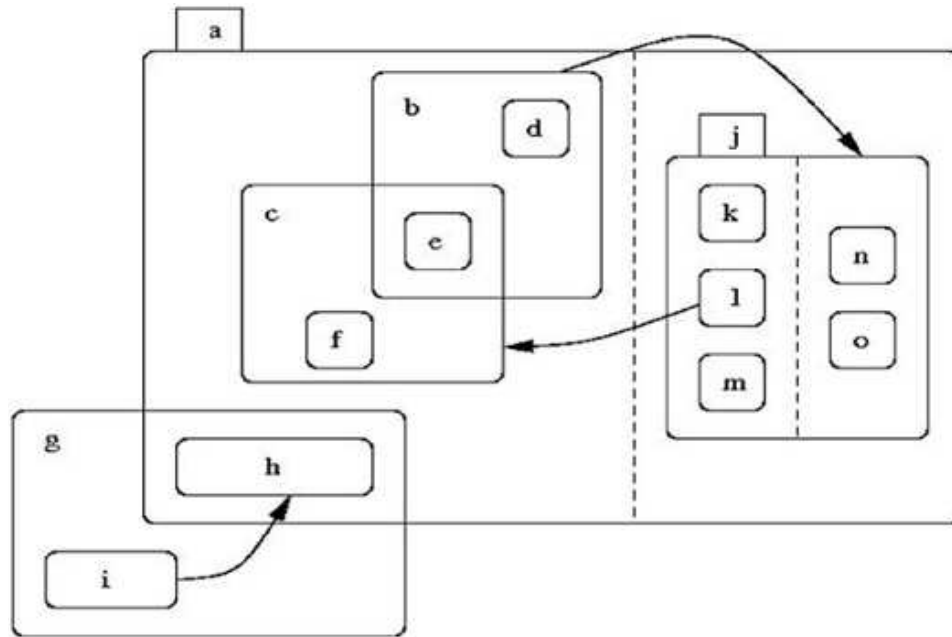


Figure 1.4: An example higraph.

Higraphs up to this point have been discussed from a theoretical standpoint and not really used for practical purposes. However, there are a few examples of higraphs used as a systems modeling tool. Forgarty's thesis [8] explores the use of Higraphs for systems modeling and traceability. Fogarty replaces traditional UML diagrams and artifacts with higraphs, and uses edges to connect different higraph artifacts as a means to transfer information between artifacts. Higraphs are an excellent tool for visually representing complicated relationships in static models; however, they cannot be used to model dynamic real-time systems due to a lack of support for rule processing [9].

## 1.4 Scope and Objectives

The purpose of this paper is to combine aspects of higraph modeling and system of systems engineering techniques to create a framework for modeling and simulating system of systems. The grouping of nodes to clusters naturally creates a hierarchy of components, which serves two purposes systems of systems modeling. First, it allows for systems to be represented more accurately. Nodes (which represent components) can belong to multiple clusters (which represent systems), which is more representative of real-life cities. The other purpose of higraphs is to allow for different viewpoints. Users can create their own clusters and group components how they please. Cities, however, are a real-time system, so there must be a mechanism that allows for communication between city systems and state changes. The proposed framework attempts to solve this issue with the introduction of an abstract concept called controllers.

## Chapter 2

### Framework for Behavior Modeling of Cities

#### 2.1 Framework

The goal of this paper is to create a prototype framework for behavior modeling of cities, and to do so in a way that mirrors the network and hierarchy organizational structure of cities. The model will allow for multiple viewpoints of the city and allow users to identify cross-cutting issues. The model will also accurately show the effect of an event on the city and capture the cascading effects throughout the rest of the city.

Figure 2.1 shows a class diagram of the metamodel for the proposed framework. According to the framework, a city is a collection of one or more systems. Systems are sets of components designed to achieve a task. Examples of engineered systems include transportation systems and electrical systems. Of course, the city will also be affected by natural systems, such as weather.

There are two types of components that make up a system. The first is called a *node*, which represents a single object of a system and contributes to the expected behavior of the system. For instance, train stations may be represented as nodes of a subway system. Each node is defined by attributes that determine its state. A change in the state of a node causes information to be passed to the rest of the

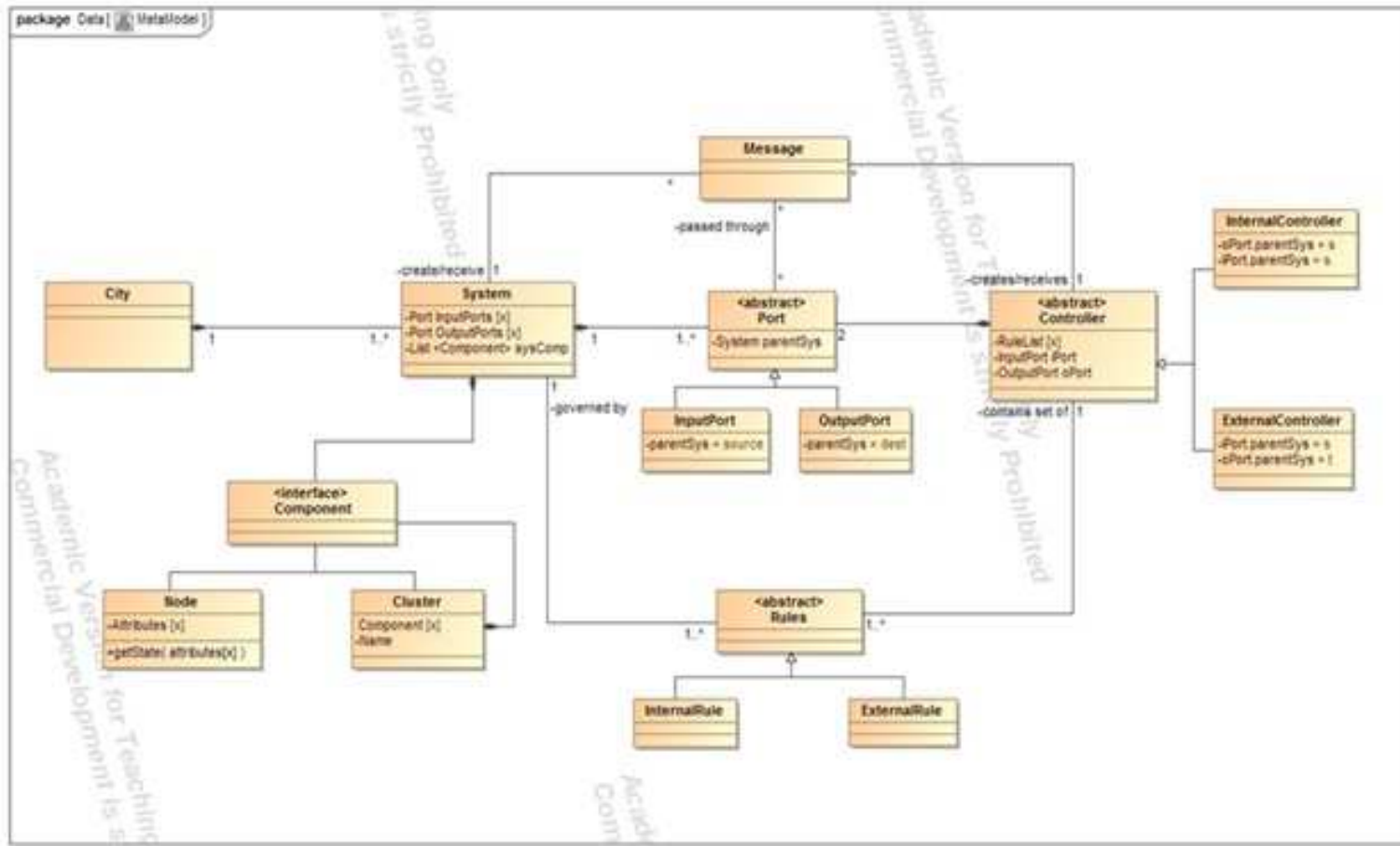


Figure 2.1: Framework Metamodel.

city. The second type of component is called a *cluster*. A cluster is a group of one or more components (i.e., nodes and clusters) with similar characteristics. Clusters can be used to show different views of a system, or can be used to represent different levels of abstraction. Note that the composite hierarchy design pattern was used to represent components of systems. This allows us to treat nodes and groups of nodes in a similar manner.

Systems are governed by a set of rules which are contained in an abstract object called a *controller*. *Rules* are logic that describes how a change in one system affects the state of components in another system. Controllers and rules can be classified as internal or external. Internal rules define how a state change in one system affects other components in the same system. External rules define how a state change in one system affects components of a different system. Controllers can only contain one type of rule, making them either internal controllers or external controllers. External controllers can only contain rules between 2 distinct systems. This means that a city having  $n$  systems, there may be as many as  $n(n + 1)/2$  controllers,  $n$  of them being internal controllers.

When a component in a system changes states, it creates a message describing the change that occurred. Messages may also originate from controllers, and will represent an effect on a particular node or cluster due to a state change. Messages are passed to other parts of the model via an abstract mechanism called ports. Ports are used to send and receive messages between systems and controllers. Ports are also responsible for disseminating the information from messages to nodes

and clusters.

## 2.2 Framework Behavior

The framework provides an infrastructure for the passing of information between different components of the system. Information is passed in the form of messages between different sub systems. When all components are in a steady state, no information is transmitted. Once a component changes its state, the system will begin the process of sending messages to determine if any other systems are affected by this change.

The first thing that occurs when the state of a component changed is a message is generated. This message is generated in the city system and contains the name of the message, the attribute of the component that changed, the new state of that component, and the event that caused the component to change. The message is sent to all output ports that are connected to the system, which is then propagated to all the input ports of the controllers that the system is connected to.

From the input ports of the controllers, the message is sent through a pre-defined set of rules that are specific to each controller. These conditions determine if the new condition will have an effect on any other components. If the controller has a condition that is relevant to the message, a new message is generated. This new message contains different types of information that the former message contained: name of the message, which components that are affected by this message, the attribute that is affected by this message, and the new value of this attribute.

Every new message that is generated by the controllers is sent to the output port of the controllers.

From the output ports of the controllers the message is sent to the input port of city systems. From the input port the system determines which components the message affects and modifies the attribute that is affected with the new value or state. This process continues until no new messages are generated and the system reaches a steady state again. Figure 2.2 is an activity diagram that visually shows this process of messages propagating through the system.

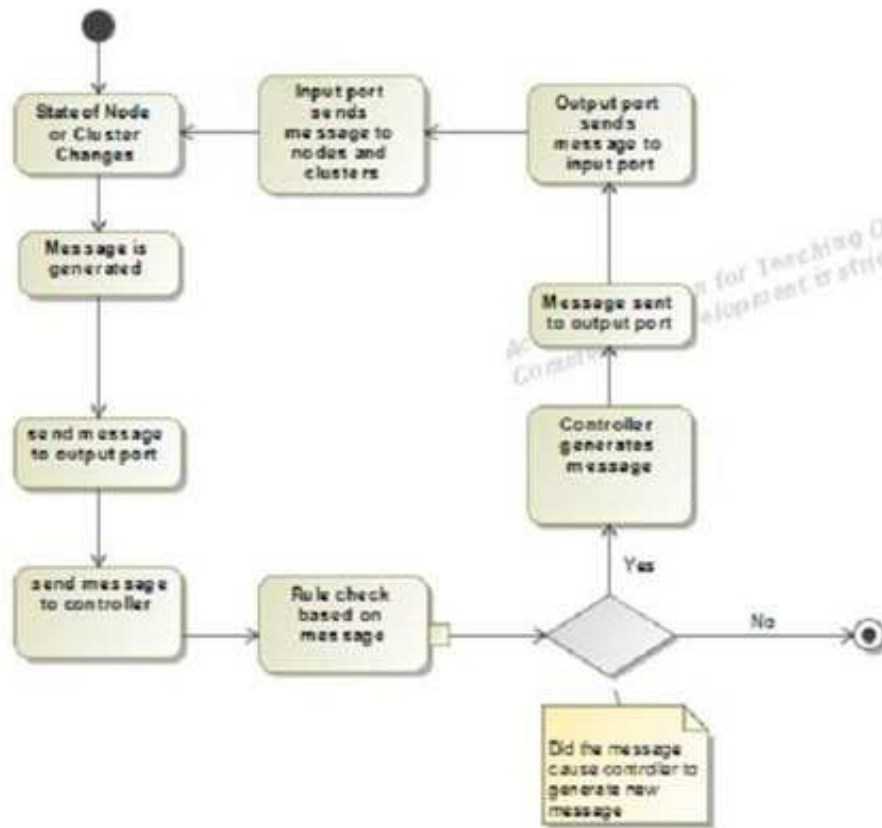


Figure 2.2: A Message passing Activity Diagram.

## 2.3 High-Level Example Model of a City

A simplified example of Washington, D.C. is modeled using the proposed framework in Figure 2.3. In this model we examine a transportation system (DC Metro) and an electrical system and how they are affected by a weather event. Clusters are used inside the transportation system in order to group individual nodes by specific characteristics. Each cluster must contain at least one node; however a particular node may belong to multiple clusters. For example, a train station such as Fort Totten belongs to both the green and red line cluster.

Nodes have specific attributes that make each node unique. Each system has a specific set of node types and each node type contains different attributes. The attributes contribute to the “state” of a node and the attributes of a node can change depending on the state of other nodes. Table 2.1 shows an example of different node types and their attributes.

<b>Node Type</b>	<b>Attributes</b>
Transportation	Station Name, Location , Open , Parking Lot size
Electric	Name, Location, Power transmitted, Component service, Component Serviced by
Weather	Rain Rate, Snowfall Rate, Wind Speed, Visibility, Temperature, Surge, Lightning Frequency

Table 2.1: Node types and attributes.

The example contains five controllers containing rule sets that control how states of components in one system affect the states of components in another system. The controllers have a very specific naming convention that identifies what system



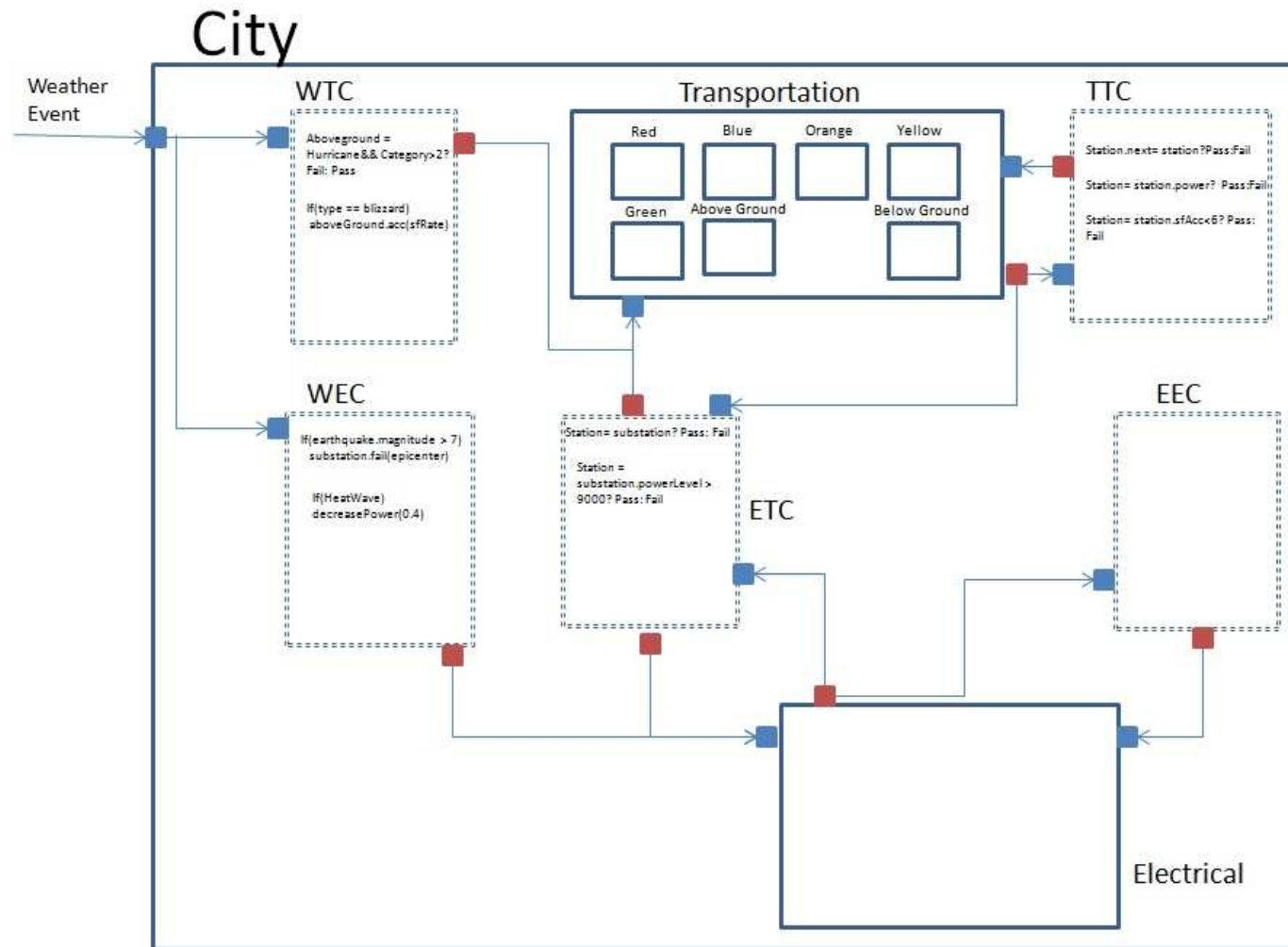


Figure 2.3: Example of Framework: A Simple City.

produces the incoming message to the system and which system the message that is generated in the controller will be sent to. A controller that is named TTC stands for the Transportation-Transportation Controller. The first word is the system in where the incoming message was generated in and the second is for where the message that is generated in the controller is going to be sent. This system contains a TTC,ETC,WTC,,WEC, and a EEC. Controllers that have repeating letters are internal controllers, while controllers with different letters correspond to external controllers. Below are example rule sets that are found within controllers.

```
WTC : Weather - Transportation Controller
```

```
If Snowfall Rate >= 8 inch per 8 hours
```

```
    Close AboveGround Cluster
```

```
If Snowfall Rate >= 1 inch per 4 hours
```

```
    Delay AboveGround Cluster
```

```
If Rain Rate >= 4 Inches per Hour
```

```
    Close Flood Zone Cluster
```

```
TTC: Transportation -Transportation Controller
```

```
If AboveGround Cluster is closed
```

```
    Put BelowGround Cluster into limited service
```

```
If BelowGround Cluster is closed
```

```
    Put AboveGround Cluster into limited service
```

Note how WTC makes changes to the transportation system based off of conditions in the weather system and TTC makes changes to the transportation system based off of conditions in the Transportation system. Information is propagated throughout the model by passing messages to other components via ports and wires.

From Figure 2.3 the blue squares indicate input ports receiving messages while red squares indicate output ports that disseminate messages. Note that it is

possible to send messages from an input port to another input port. This scenario must exist in order to allow rule propagation to clusters.

**Demonstration of a Cascading Failure.** Figures 2.4 through 2.8 illustrate a sequence of system states for a cascading failure propagating throughout the city. Affected portions of the model are highlighted in red. The key points to note are as follows:

1. In Figure 2.4 a hurricane is affecting the city. A message is created in the framework with details of the hurricane, such as wind speed and rainfall rate. The message is passed to the WTC and WEC controllers for rule processing.
2. Figure 2.5 illustrates the consequences of Step 1. Within the controllers, the rules in red are triggered. In the WTC, a message to close all above ground train stations is sent to the transportation system, while in the WEC, a message that affects power lines is generated and sent to the electrical system.
3. Figures 2.6 and 2.7 show how information is disseminated to the appropriate cluster after messages have been sent to their respective systems. The states of all nodes within the cluster are changed in accordance to the message. At this point, all above ground stations are closed due to the hurricane and overhead transmission lines are damaged because of the high winds. Since the states of the nodes have changed, new messages detailing the changes from the transportation and electrical systems are created and sent to the TTC and EEC controllers for additional rule checking.

4. Figure 2.8 completes the sequence of cascading failure mechanisms. Inside the TTC and EEC, only one rule is triggered. The damage done to the transmission lines means the substations are without power. A message is sent from the EEC back to the electrical system, closing all substations.

The changes to the substations creates a new message that is sent to the ETC. Without power coming from the substations, every station in the transportation system is closed. This example shows how this model can predict cascading failures throughout a city. It shows systematically how the effects of a hurricane affect the whole city. At the beginning of this example there was a fully functioning city before the hurricane occurred. However, by the end it showed major failures in the electrical system as well as many closures and limited service of stations in the transportation system.

Cities and SoS in general are comprised of much more than two component systems. As the number of components grow, the number of controllers and connections in the framework increases at an exponential rate. Therefore, it is necessary to create software capable of managing and simulating the elements of the framework. The next section will discuss the implementation of the proposed framework in Java.

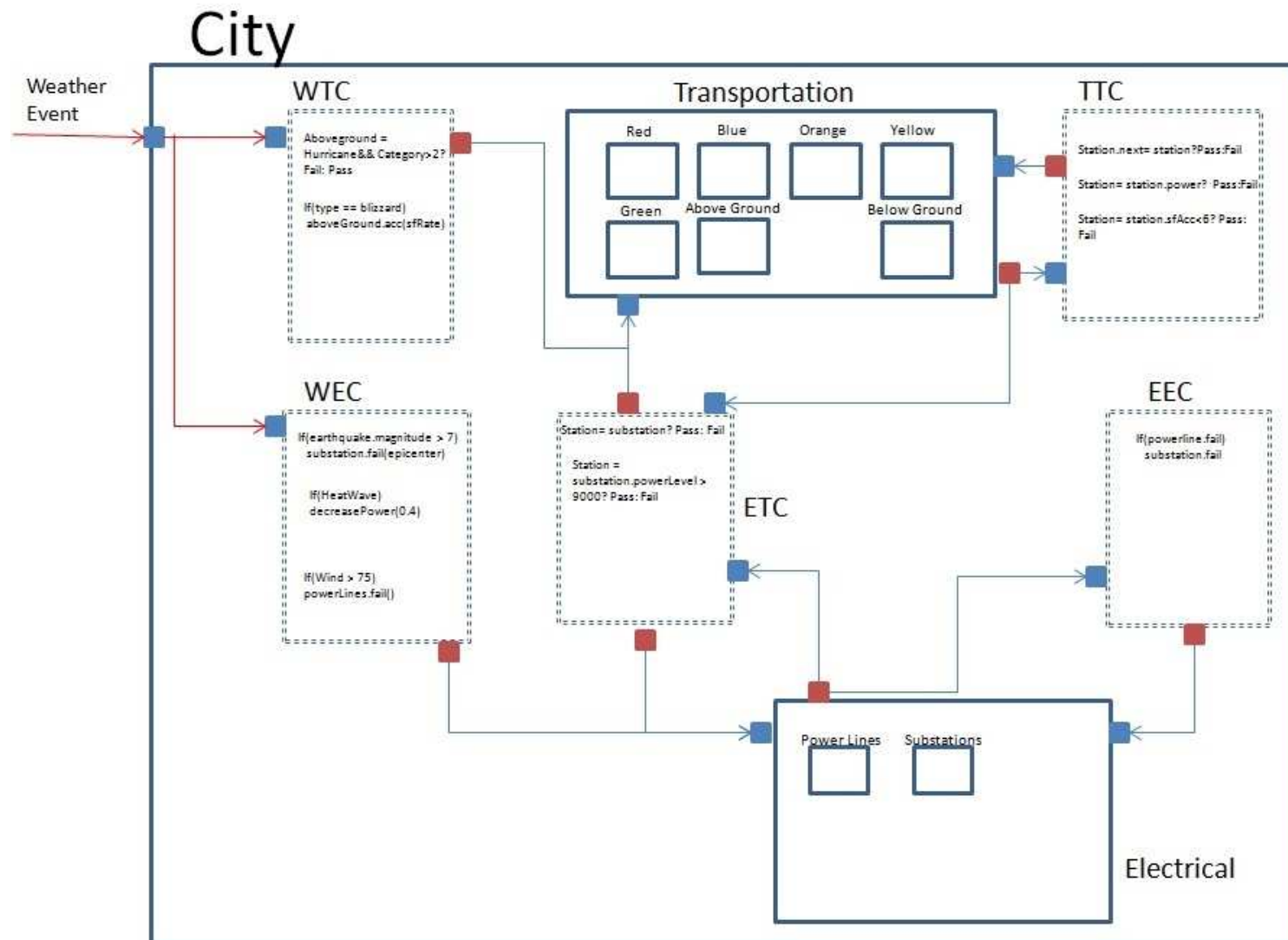


Figure 2.4: Cascading failure, Step 1.

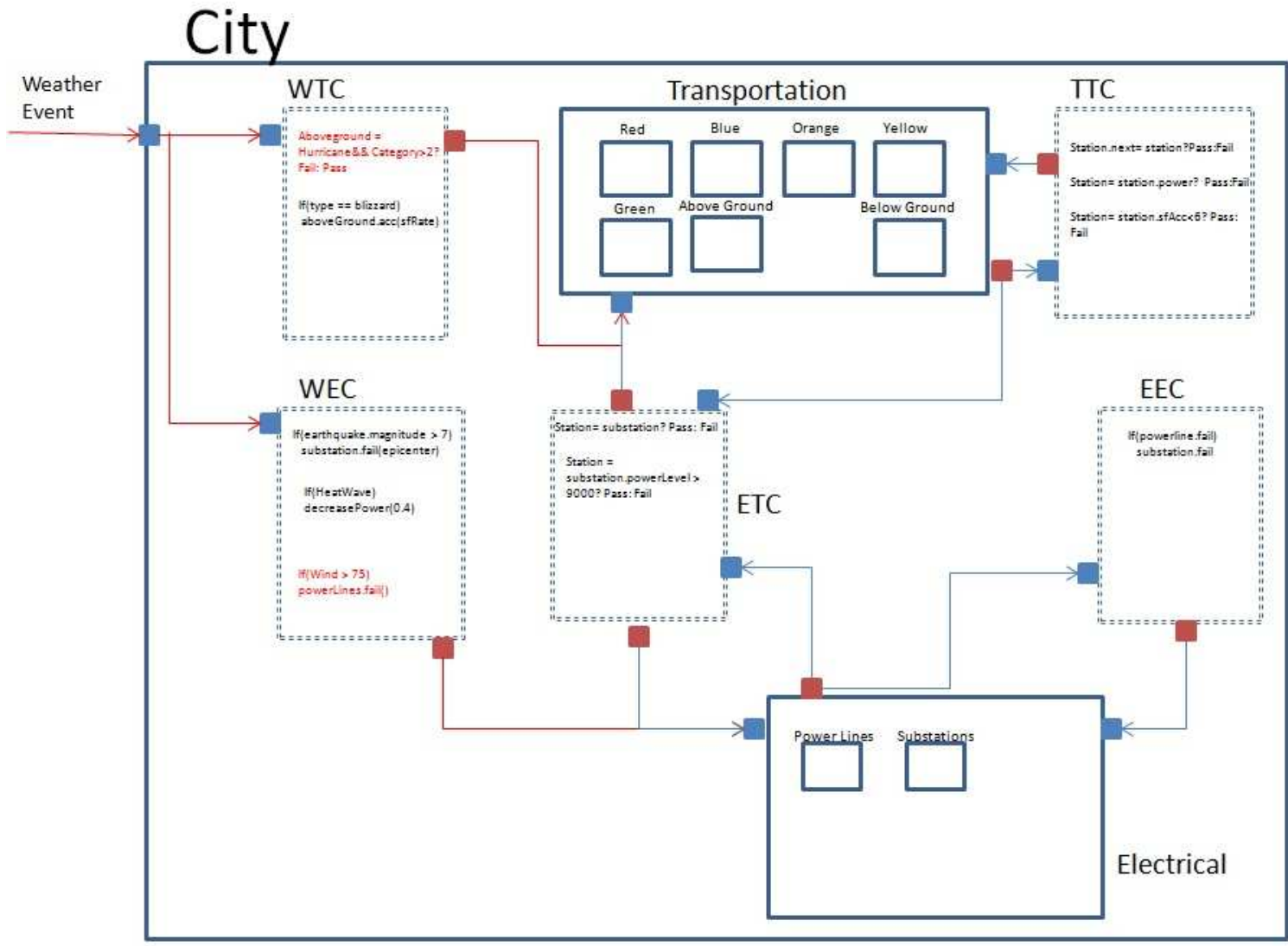


Figure 2.5: Cascading failure, Step 2.

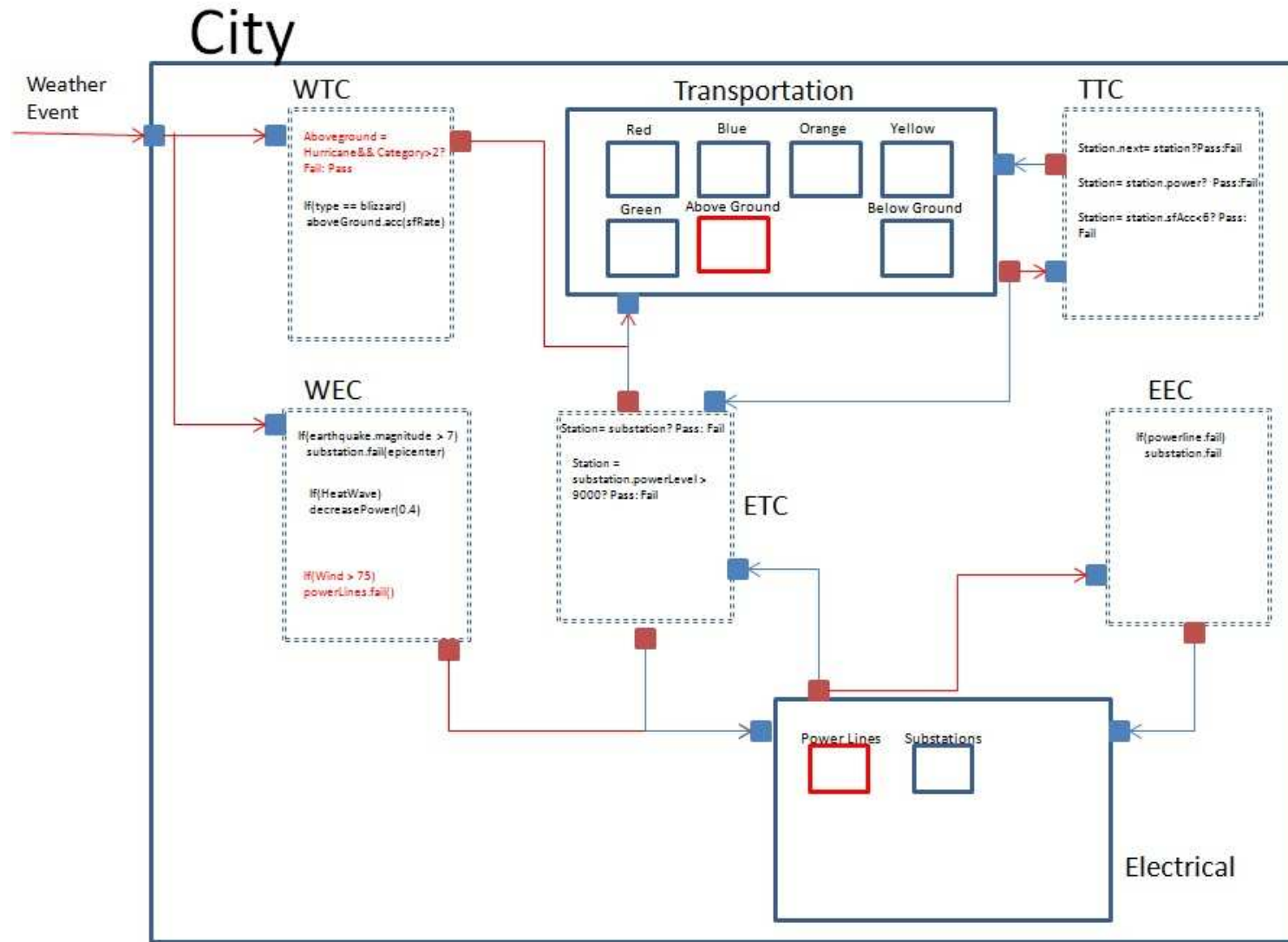


Figure 2.6: Cascading failure, Step 3.

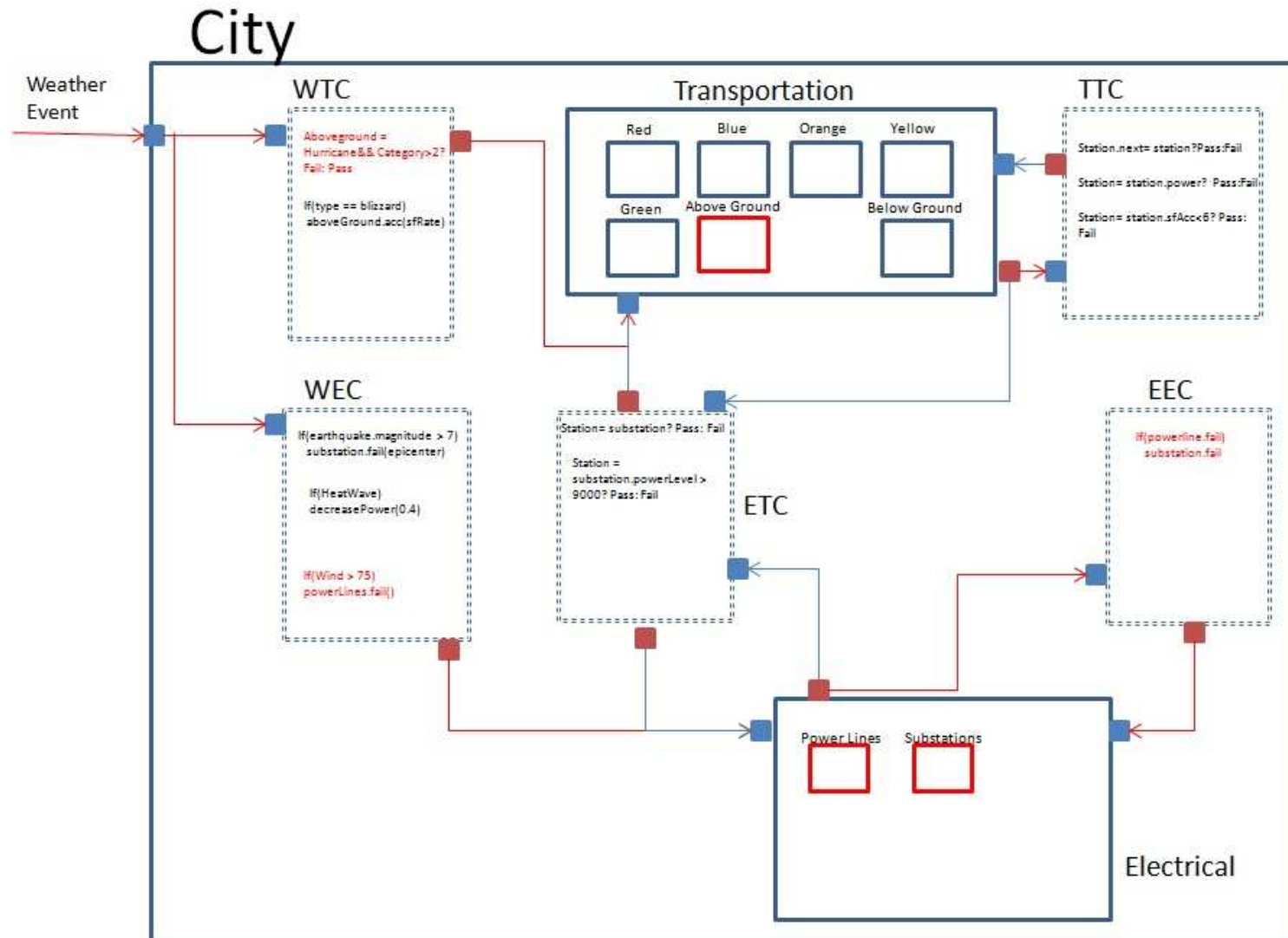


Figure 2.7: Cascading failure, Step 4.



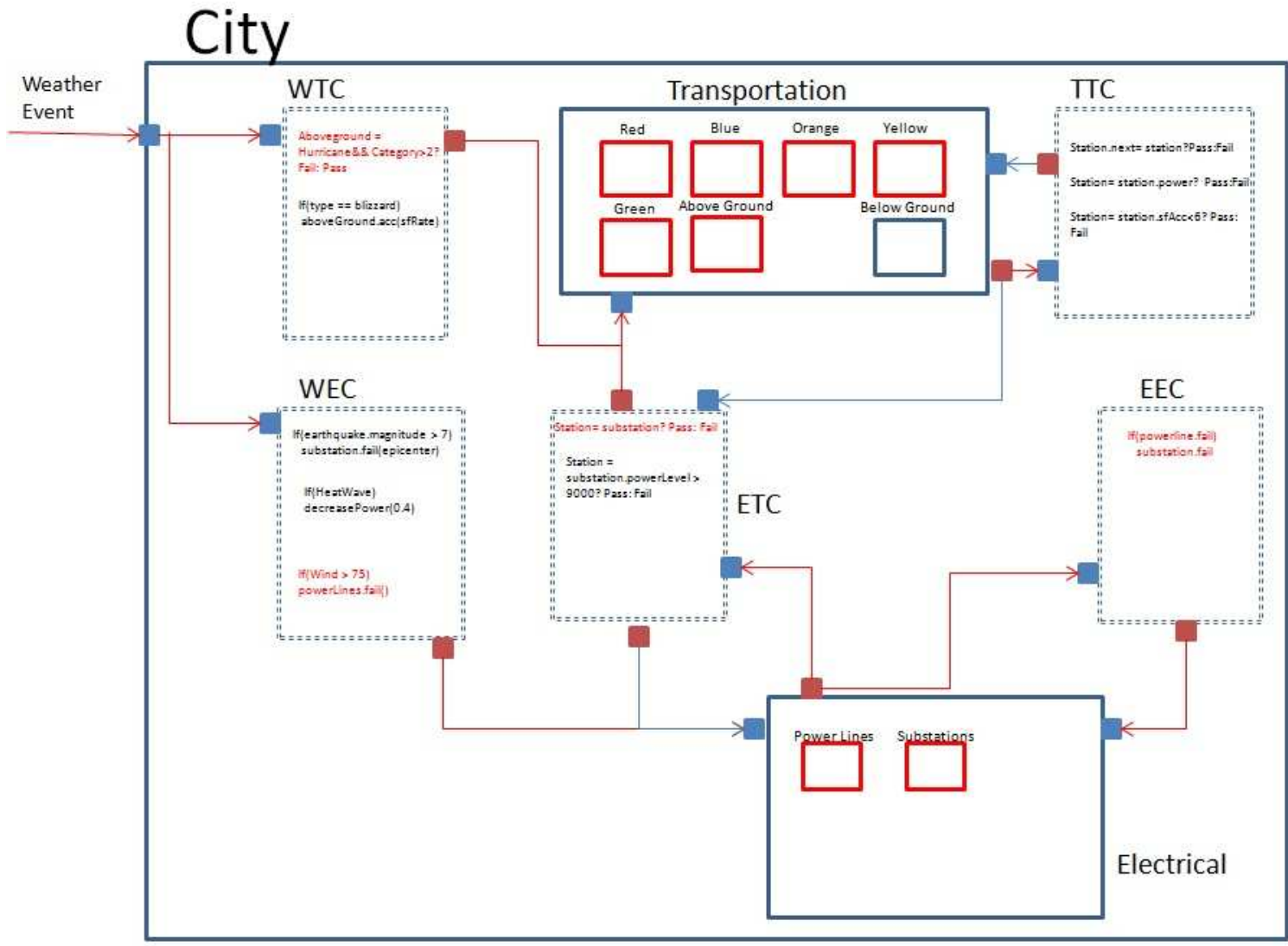


Figure 2.8: Cascading failure, Step 5.

## Chapter 3

### Implementation of Framework in Java

#### 3.1 Framework Design Patterns

In order to implement the framework in Java, it is necessary to allow hierarchical tree structures as well as an ability to send, receive, and process information as information propagates throughout the model. By using a composite design pattern, groups of a certain object can be processed in the same manner as a single instance of the object [9]. The implementation of the composite design pattern is shown by the class diagram in Figure 3.1.

The component interface defines all variables and methods to be used in `BaseComponent` and `MetaComponent`. The `BaseComponent` is essentially a node and cannot contain other components. The `MetaComponent` is another type of `Component`, however it is a container for a group of `Components`. Note that it is possible for a `MetaComponent` to contain other `MetaComponents`, therefore creating a hierarchy tree.

The process function allows each component to process information differently, depending on what type of component it is. For `BaseComponents`, `ComponentEngines` are responsible for processing data and information. The `BaseComponent`'s process method will simply invoke the `ComponentEngine`'s process method.

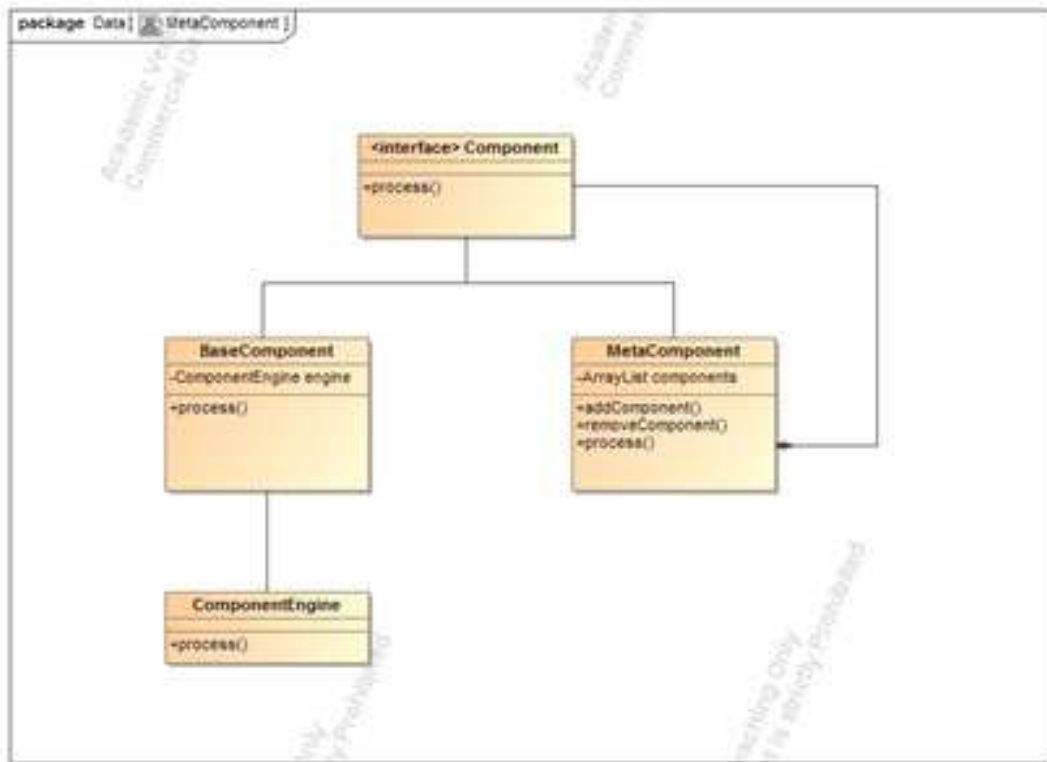


Figure 3.1: Class diagram for the component hierarchy design pattern.

The MetaComponent’s process method loops through the list of components and calls each component’s process method.

To facilitate data flow processing, we attach abstract objects called ports to the components and connect components together with wires. Ports are implemented using a hierarchy of interfaces as showing in Figure 3.2. InputPorts receive information sent from OutputPorts via wires. Ports also utilize Java generics, meaning they are capable of receiving any type of value.

A wire is defined by connecting a single OutputPort to one or more InputPorts. This implies a one-to-many connectivity between source and target ports.

The use of ports and wires implies that the system of components can be modeled as a graph. The use of the composite design pattern transforms the graph to a

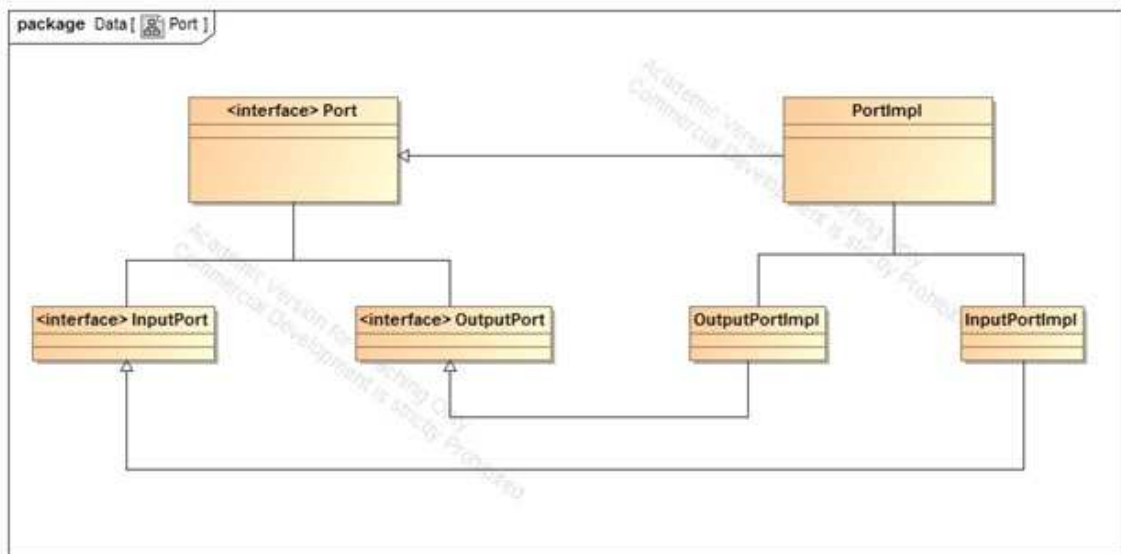


Figure 3.2: Port Architecture.

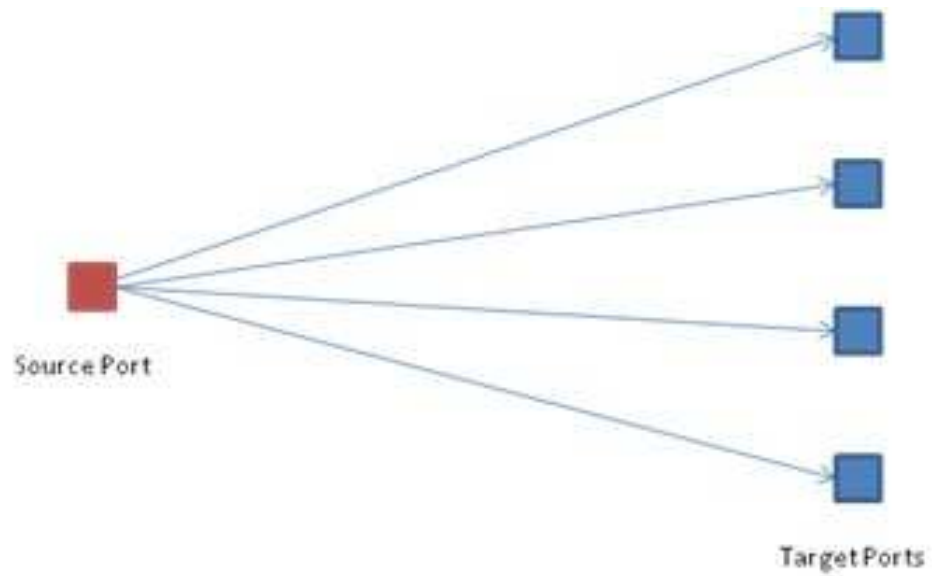


Figure 3.3: Wire Implementation

higraph model [10]. To demonstrate the capabilities of the higraph implementation, three different network configurations of arithmetic processors are shown below.

**MetaComponent: Version 1.** Version 1 of the higraph includes one MetaComponent (mcDest) and one BaseComponent (additionComponent) as illustrated in Figure 3.4.

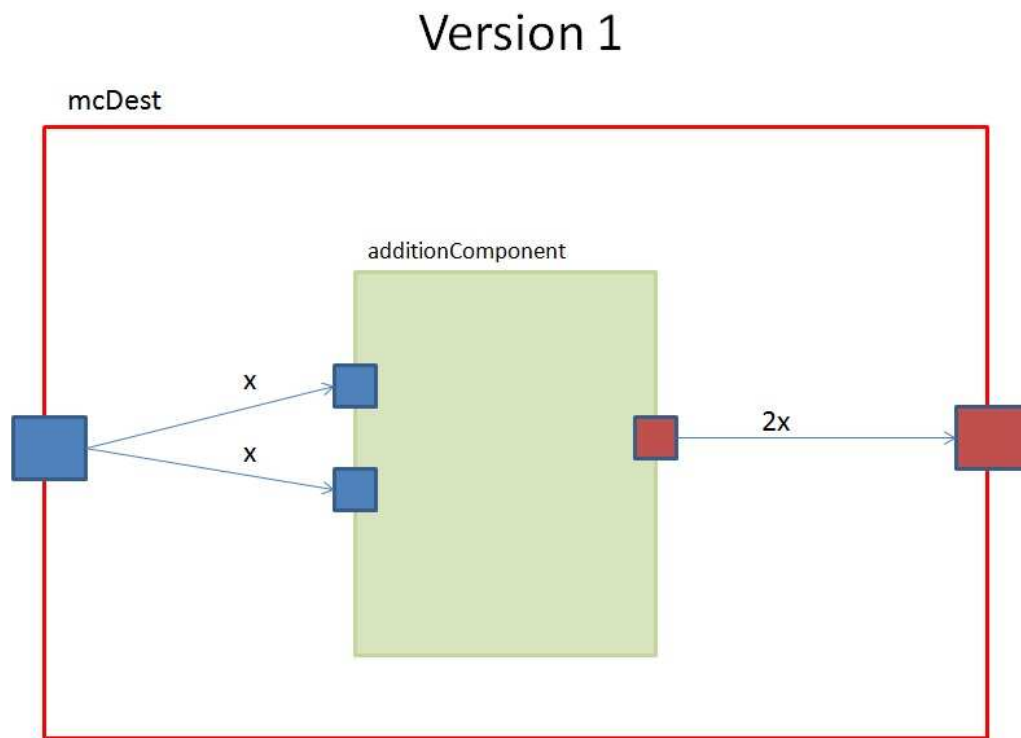


Figure 3.4: Higraph Version 1 - One MetaComponent, One BaseComponent

Blue squares represent InputPorts while red squares represent OutputPorts. The BaseComponent contains an AdditionEngine that adds all the values of the InputPorts attached to the BaseComponent. Note that in order to pass information from a MetaComponent to all its constituent components, InputPorts must be able to connect to InputPorts. The same applies for OutputPorts. The code handles this by

creating a hidden internal Output/Input Ports and mapping them to the respective "external" Input/Output Ports.



Figure 3.5: External/Internal Ports.

When passing the value  $x=7$ , we get the expected result of 14 in the OutputPort.

```

_____ output _____
SubComponents of Destination Component
=====
BC(additionComponent)  Inputs: 7 7  Outputs: 14

InputPorts in Component Destination Component
=====
mcSourceInputPort.value=7

OutputPorts in Component Destination Component
=====
mcDestOutputPort.value=14
_____

```

**MetaComponent: Version 2.** Version 1 loaded the value directly to the MetaComponent's InputPort. In version 2 of the higraph we add a 2nd MetaComponent and connect it to version 1. An integer will be loaded to the OutputPort of mcSource and be passed to mcDest for computation.

For this example,  $x=17$  will be loaded to the model.

```

_____ output _____

```

## Version 2

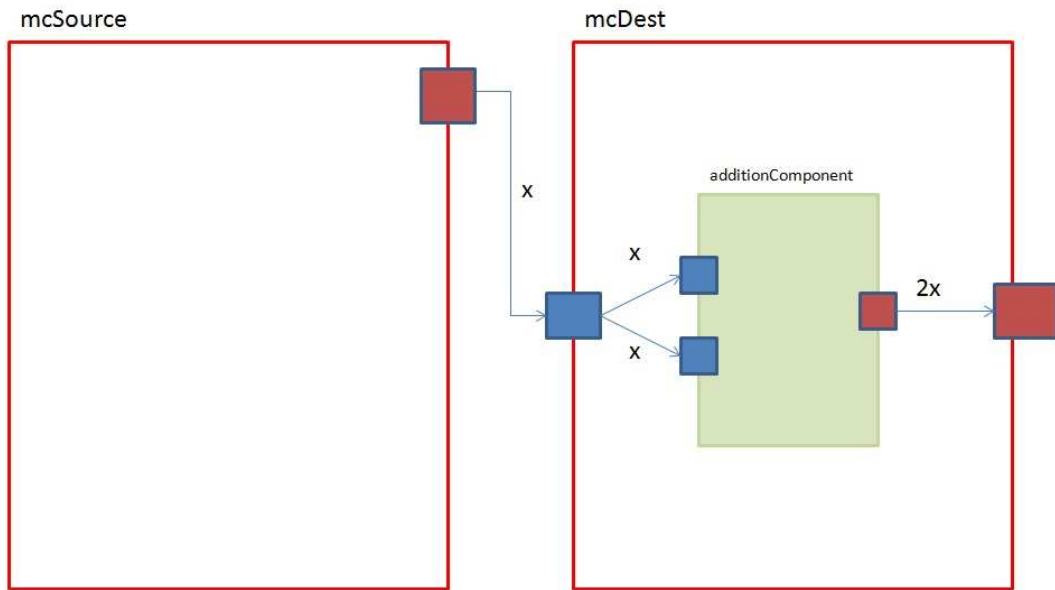


Figure 3.6: Version 2 - MetaComponent feeding data to mcSource.

```
OutputPorts in Component Source Component
=====
mcSourceOutputPort.value=17
SubComponents of Destination Component
=====
BC(additionComponent) Inputs: 17 17 Outputs: 34

InputPorts in Component Destination Component
=====
mcSourceInputPort.value=17

OutputPorts in Component Destination Component
=====
mcDestOutputPort.value=34
```

---

**MetaComponent: Version 3.** In version 3 of the higraph model, we attach a 2nd additionComponent to mcSource, and we wrap the entire graph in a "wrapper" MetaComponent.

# Version 3

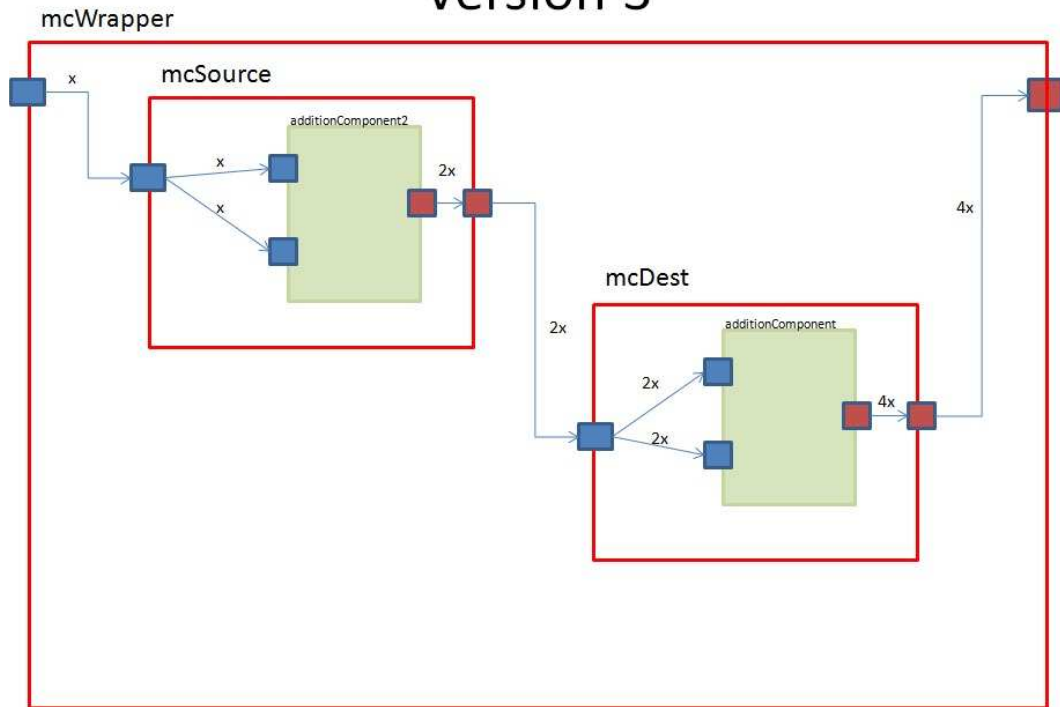


Figure 3.7: Version 3 - MetaComponent wrapping around Version 2.

When we load  $x=7$  to the InputPort of mcWrapper we expect the output to be 28.

```
----- output -----  
InputPorts in Component Wrapper Component  
=====  
mcWrapperInputPort.value=7  
SubComponents of Source Component  
=====  
BC(additionComponent2) Inputs: 7 7 Outputs: 14  
  
SubComponents of Destination Component  
=====  
BC(additionComponent) Inputs: 14 14 Outputs: 28  
  
OutputPorts in Component Wrapper Component  
=====  
mcWrapperOutputPort.value=28  
-----
```

The examples show that the composite and port/wire design patterns can be used to



create a higraph capable of passing data between components and processing them. In the next section, we will discuss how to apply the SoS framework concepts the higraph software architecture.

## 3.2 Model Architecture

This section describes how we will extend the MetaComponent and message processing software design patterns in order to implement the SoS framework. Figure 3.8 shows a class diagram of the full SoS framework architecture. We will go into details of SoS specific objects in the following sections.

### 3.2.1 Message Passing

Messages are used as the mechanism that passes information between components of the model. There are two types of messages: ControllerMessages and SystemMessages. ControllerMessages originate from controllers and will change the state of a node or cluster. When the state of a component changes, a SystemMessage is sent back to the controller with information about the component's current state. This information will be used by the controller for rule processing.

### 3.2.2 Components

**Nodes.** Nodes do not have the same function as a BaseComponent since it requires no real computation. Nodes also do not resemble a MetaComponent since nodes are

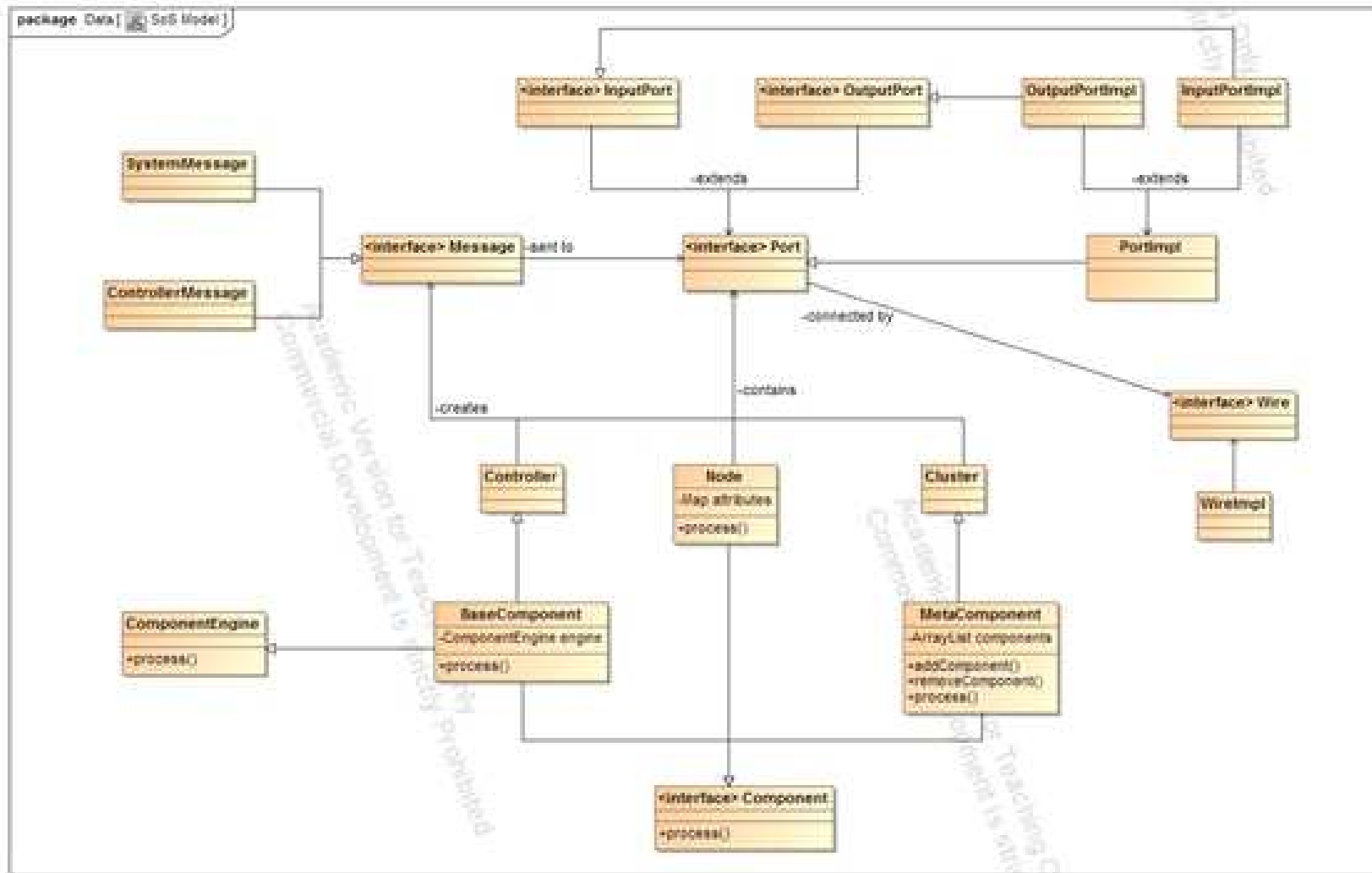


Figure 3.8: SoS Framework Architecture.

single elements and do not contain other objects. Therefore, a node is a separate class that implements the Component interface. Nodes contain a list of attributes associated with the node. Since there are many different types of nodes (i.e., transportation or electrical nodes), the node class can be extended to create subclasses with unique attributes.

When a node receives a message from the controller, the `process()` method extracts the attribute affected and the value to be modified. It will look to see if that attribute exists in the node, and change the value if it does exist. Otherwise, nothing will occur.

**Clusters.** Clusters resembles a MetaComponent and therefore extend the MetaComponent class. The only difference between clusters and MetaComponents is how the message is propagated to its constituent components. Clusters will only propagate the message to the components specified in the incoming message.

**Controller.** We will be using the BaseComponent class as the basis for the Controller class. The rules will be defined as a ComponentEngine and then attached to the controller. This allows for the separation of the rules from the model and is easier to maintain.

### 3.2.3 Rule Processing

When the state of a node changes, the SystemMessage created contains the list of all of the node's attributes and its values for evaluation in the controller. The

controller contains a ComponentEngine with rules defined for the specific controller. For instance, the TTC controller will contain a TTC ComponentEngine containing rules related to transportation related behavior. Likewise, the ETC controller will contain a ETC ComponentEngine defining the electric/transportation relationship. For this version of the framework, the rules are defined as if/else statements.

If a rule is triggered, the controller will create a ControllerMessage that is sent back to the originating system, applying a change to the node's attributes and creating an additional SystemMessage. This process repeats itself until the system reaches a "steady state" where no rules are triggered and the nodes do not change state.

### 3.3 Implementing the City Model in Java

We will demonstrate the software version of the SoS framework by implementing a simplified version of the city model described in Chapter 2. The city only contains a transportation system, a subset of the Washington D.C. Metro System shown in Figure 3.9, and a TTC controller. Inside the transportation system, a single cluster called the "Green Line" contains five stations:

```
Greenbelt,  
College Park,  
Prince George's Plaza,  
West Hyattsville,  
Fort Totten.
```

The TTC is connected to the transportation system and contains the TTC ComponentEngine. The code used to set up the city can be found in Appendix A, however

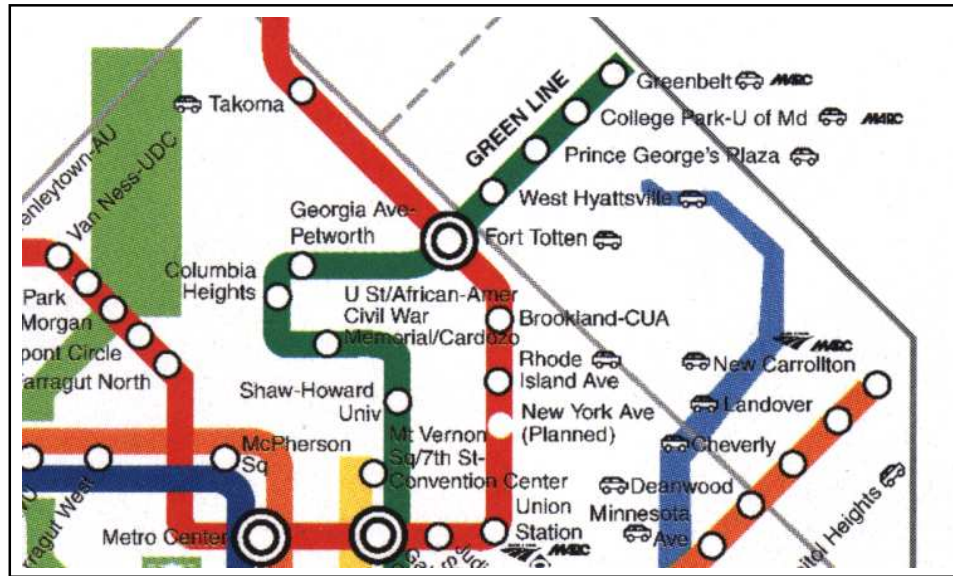


Figure 3.9: Northeast section of the Washington D.C. Metro System.

it can be visually represented as shown in Figure 3.10. A subclass of Node, called StationNode, was used to create the stations inside the cluster. StationNode behaves identically to Node, however it defines the following attributes for each instantiation of StationNode:

- stationName
- stationElevation
- stationStatus
- stationLocation
- stationElectricBy
- waterAcc
- serviceRate
- stationElectricReq

The TTC ComponentEngine contains a set of if/else statements that look for the attribute names and performs logic. In this case, we have a rule that states that if water accumulation at the station is greater than 2.5, the station status will be set to closed. We implement the code as a switch statement:

```
case "waterAcc":
{
```

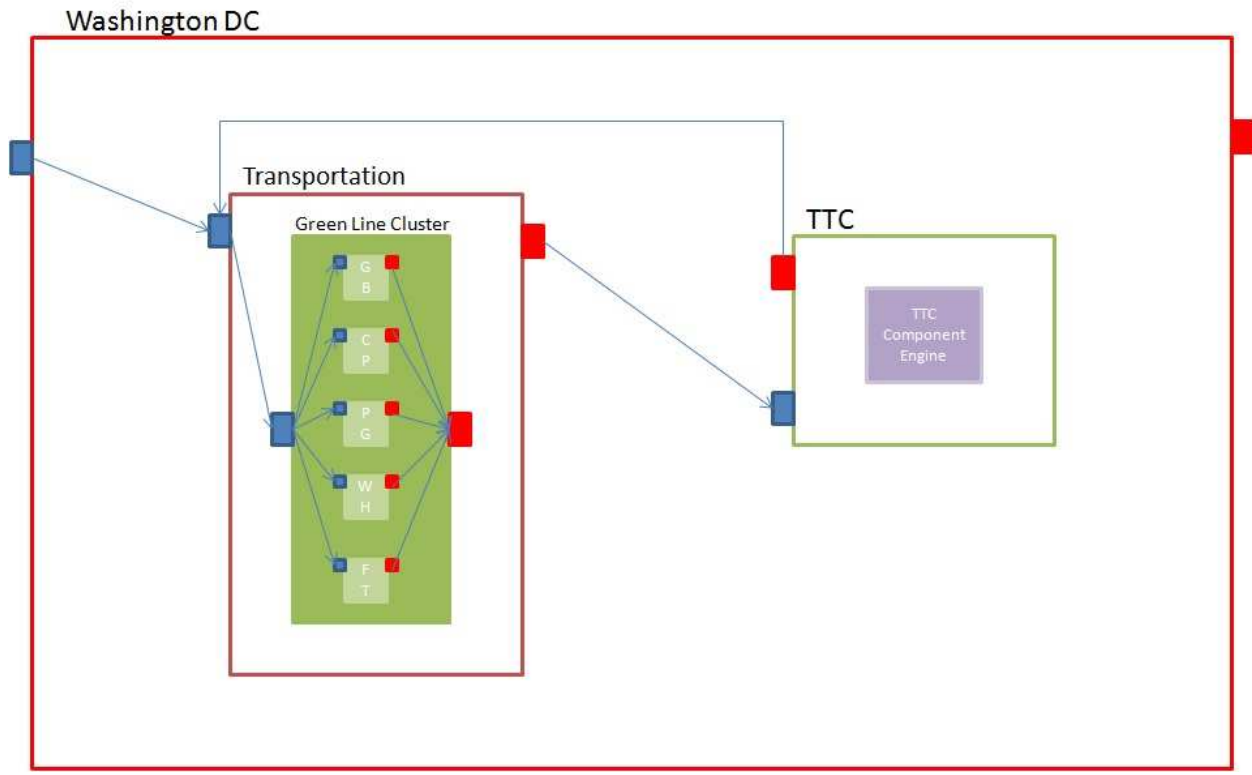


Figure 3.10: Visual Representation of City model in Java.

```

    Double v = (Double) attributes.get(att);
    if(v > 2.5) {
        System.out.println("Closing station: " + sysMessage.
            getfromComponent());
        Message outMessage = new ControllerMessage ("
            WaterLevelMessage", "TTController",
                "stationStatus", "closed",
                    sysMessage.getfromComponent()
                );
    }
}

```

To begin the simulation, a weather event brings 3 inches of rainfall to the College Park metro station. To simplify matters, the weather system and WTC controller is abstracted away, leaving only a ControllerMessage that changes the state of the College Park metro station.

```

    Message flood = new ControllerMessage("Rain", "WTController", "waterAcc",
        (Double) 3.0, "College Park");

```

Next, the message is passed from the city to the transportation system. Once that occurs, processing for the transportation system begins by passing the message to all components of the transportation system. In this case, only the Green Line cluster receives the message.

---

```

----- output -----
Processing for Washington, D.C.
Moving ControllerMessage Rainfrom cityInputPort to Internal OutputPort of cityInputPort
Moving ControllerMessage Rain from Internal OutputPort of cityInputPort to transInputPort
Processing for DC Metro
Moving ControllerMessage Rainfrom transInputPort to Internal OutputPort of transInputPort
Moving ControllerMessage Rain from Internal OutputPort of transInputPort to greenLineIPort

```

---

The cluster then searches for College Park and changes the value of waterAcc to 3.0. Note that no other station receives the message and their water accumulation values remain at null.

---

output

---

```
Processing for Green Line Cluster
Moving ControllerMessage Rainfrom greenLineIPort to Internal OutputPort of greenLineIPort
Processing for Greenbelt
Processing for College Park
Changing Attribute: waterAcc to 3.0
Creating SysMessage for College Park
Processing for Prince George's Plaza
Processing for West Hyattsville
Processing for Fort Totten
```

Attributes of Greenbelt

```
=====
Attribute: waterAcc = null
Attribute: isStationAboveGround = null
Attribute: stationElevation = null
Attribute: stationElectricReq = null
Attribute: stationStatus = null
Attribute: stationElectricBy = null
Attribute: stationName = null
Attribute: serviceRate = null
Attribute: stationLocation = null
```

Attributes of College Park

```
=====
Attribute: waterAcc = 3.0
Attribute: isStationAboveGround = null
Attribute: stationElevation = null
Attribute: stationElectricReq = null
Attribute: stationStatus = null
Attribute: stationElectricBy = null
Attribute: stationName = null
Attribute: serviceRate = null
Attribute: stationLocation = null
```

Attributes of Prince George's Plaza

```
=====
Attribute: waterAcc = null
Attribute: isStationAboveGround = null
Attribute: stationElevation = null
Attribute: stationElectricReq = null
Attribute: stationStatus = null
Attribute: stationElectricBy = null
Attribute: stationName = null
Attribute: serviceRate = null
Attribute: stationLocation = null
```



```
Attributes of West Hyattsville
=====
Attribute: waterAcc = null
Attribute: isStationAboveGround = null
Attribute: stationElevation = null
Attribute: stationElectricReq = null
Attribute: stationStatus = null
Attribute: stationElectricBy = null
Attribute: stationName = null
Attribute: serviceRate = null
Attribute: stationLocation = null
```

```
Attributes of Fort Totten
=====
Attribute: waterAcc = null
Attribute: isStationAboveGround = null
Attribute: stationElevation = null
Attribute: stationElectricReq = null
Attribute: stationStatus = null
Attribute: stationElectricBy = null
Attribute: stationName = null
Attribute: serviceRate = null
Attribute: stationLocation = null
```

---

With the change in the state at the College Park station, a SystemMessage is generated with the attributes of the College Park and sent to the TTC. The TTC ComponentEngine reads the message and checks it against the defined rules. Since the attribute waterAcc is above 2.5, a new ControllerMessage is sent to the transportation system, closing College Park Station.

---

```
----- output -----
Processing for WTC Controller
Closing station: College Park

Attributes of College Park
=====
Attribute: waterAcc = 3.0
Attribute: isStationAboveGround = null
Attribute: stationElevation = null
Attribute: stationElectricReq = null
```

Attribute: stationStatus = closed  
Attribute: stationElectricBy = null  
Attribute: stationName = null  
Attribute: serviceRate = null  
Attribute: stationLocation = null

---

## 3.4 Message Passing using JGraphT

A major issue with the message passing and rule processing scheme described in Section 3.2 is that the model is unable to recognize loops within the model and iterate around them. This means that a Component that generates a SystemMessage will not receive a ControllerMessage from a Controller. Therefore, we must design a mechanism for controlling:

- The order in which objects are processed in the model.
- How often those objects can be processed.

To achieve this, each Component in the model contain a directed graph representation of itself as well as logic for traversing and processing the objects, which we will call the *GraphManager*. JGraphT will be used in implementing the GraphManager. JGraphT [14] is a graph library that provides mathematical graph-theory objects and algorithms for a various types of graphs, making it more than suitable for our purposes.

### 3.4.1 Graph Structure

The vertices of the graph contained in GraphManager will be made up of several types of objects. These objects include:

- The root Component.
- The ports (external and internal) registered to the root Component.

- The Components connected to the root Component (either externally or internally).
- The InputPorts of the connected Components.

When a wire is used to connect an OutputPort to an InputPort in the model, the GraphManager adds the ports as well as parent Components of the port to the graph. In addition, edges are added as follows:

- Parent Component of OutputPort to OutputPort
- OutputPort to InputPort
- InputPort to Parent Component of InputPort

The graph only contains information about the relationships at the Component level. That is, the entire model can be viewed as a hierarchy of graphs. In order to traverse to lower levels of this hierarchy, it is important to ensure there are no cycles within a graph.

Edge Source	Edge Target
Component	Output Port (internal/external)
OutputPort(internal/external)	InputPort(internal/external)
InputPort(external)	Component

Table 3.1: Possible Edge Sources and Targets

Given the way that the graph is built and the structure of the original higraph model, we can assume that the graph can only have certain source/target combinations,

as defined in Table 3.1. Note that there cannot be an edge that has an Internal InputPort as the source. This is because the target of that edge would be the root Component, which would cause a cycle. This property of the graph will be useful when traversing and processing the graph.

## 3.4.2 Traversing and Processing the Graph

### 3.4.2.1 Traversing a Graph

The traversal begins at the root vertex of the graph, which is the Component being processed. The GraphManager checks the degree of outgoing edges from the root. If the degree is equal to 1, then the GraphManager simply performs a simple depth-first iteration of the graph.

If the degree of outgoing edges from the root is greater than 1, then a depth-first iteration is still performed. The search procedure is a little more complicated because the order of the branch searches must be determined. First, the set of all vertices that are connected to the root vertex is created. From Table 1, it is known that this set only contains OutputPorts. Given the structure of the Higraph model, we ideally want to process internal subcomponents first before passing the information along externally. Therefore, we prioritize internal OutputPorts over external OutputPorts by adding the OutputPorts to a *Queue* and using a special *Comparator* to differentiate between internal and external OutputPorts. From here, we run a depth-first iteration in the same order as the queue.

### 3.4.2.2 Processing the Graph

While traversing the graph, the GraphManager stops at each vertex and process each vertex differently depending on the type of vertex it is. Table 3.2 shows the action taken for each type of vertex.

<b>Vertex Type</b>	<b>Action(s) Taken</b>
Internal OutputPort	convertExternalDataToInternalData(), propagateSignal()
External OutputPort	propagateSignal()
Internal InputPort	convertInternalDataToExternalData()
External InputPort	N/A
Component	Component.process()

Table 3.2: Vertex types and actions.

While it is not possible for a cycle to occur on the same graph, it is possible for loops to occur when two graphs point to the same Component. This will cause the GraphManager to traverse the loop infinitely unless there is a way to break the traversal. A simple way of dealing with this is as follows: each Component keeps track of the number of times the GraphManager has visited it. As soon as the number of visits hits a pre-specified threshold, then the GraphManager stops traversing the graph.

## 3.5 Example: Revisiting the City

To demonstrate the GraphManager process, we will use the same city example as described in Section 3.2. The analysis is divided into two parts: (1)

examination of a simplified model having only one metro station in the green line cluster, and (2) the same analysis, but with three metro stations in the cluster. Full details of the problem setup and solution can be found in Appendix C.

**One-Station Scenario.** We begin with a simple scenario by only including College Park station in our model. Figure 3.11 is a visual representation of the simplified City model. Once again, a weather event brings 3 inches of rainfall to the College Park metro station. This message is passed into the City (Washington D.C) level. The city-level graph produced by the GraphManager is shown below and will dictate the order of processing.

From Table 3.2, the order of actions is:

1. `convertExternalDataToInternalData()`
2. `propagateSignal()`
3. `Transportation.process()`

When the GraphManager reaches the Transportation vertex, it will obtain the transportation-level graph and traverse that graph. The transportation level graph is shown in Figure 3.13.

Since the outgoing degree of the Transportation node is 2, we must decide which branch to traverse first. Since one of the child vertices is an internal OutputPort, the GraphManager will traverse the “internal branch” before going to the “external branch.” Again, when the GraphManager reaches the Green Cluster ver-

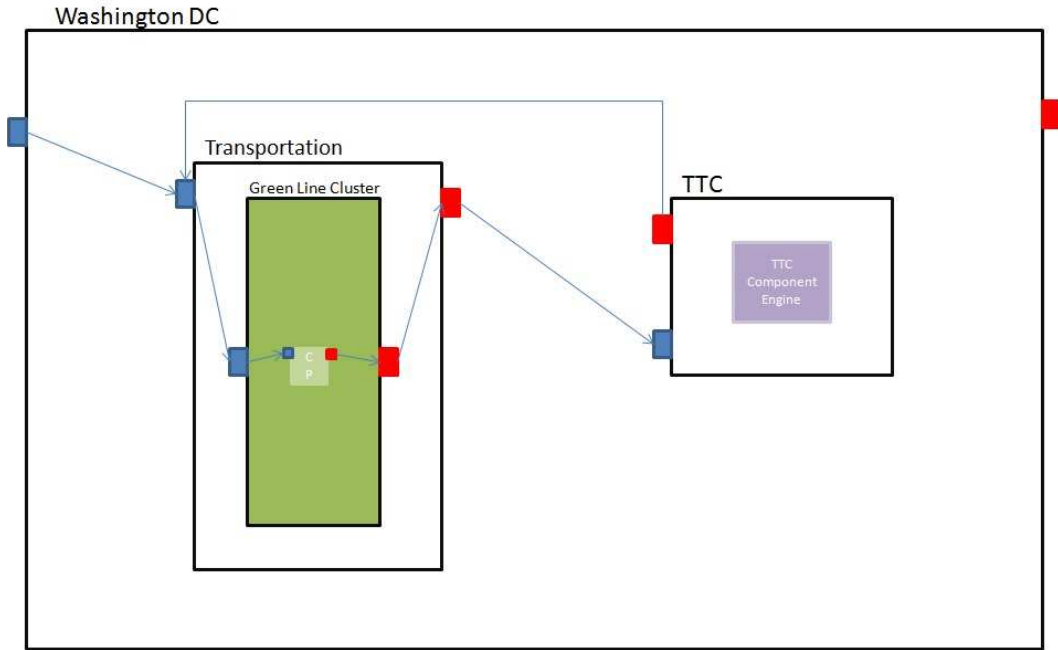


Figure 3.11: Visual Representation of Simplified City Model.

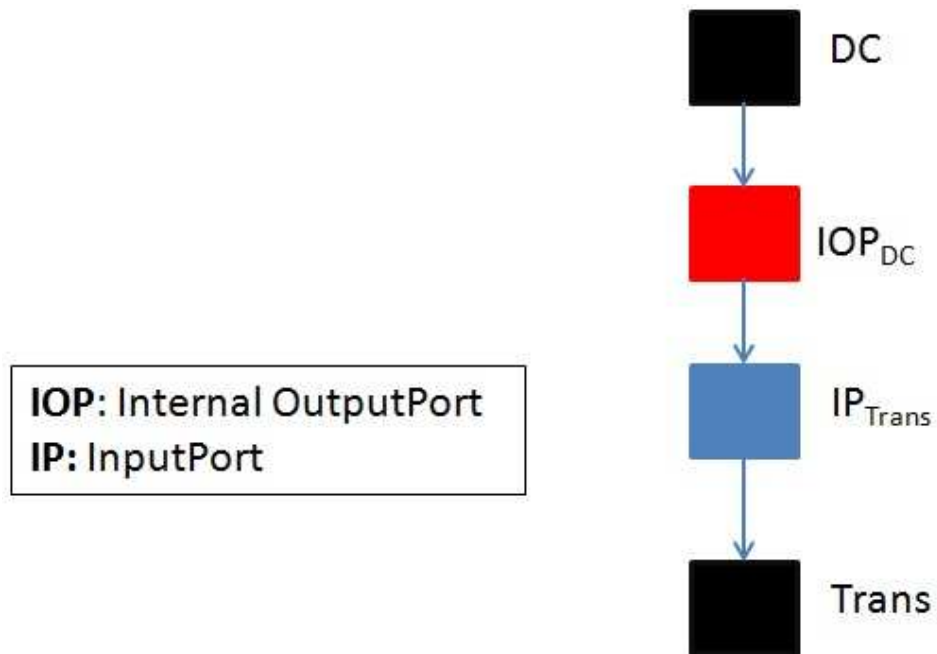


Figure 3.12: High-level city graph.



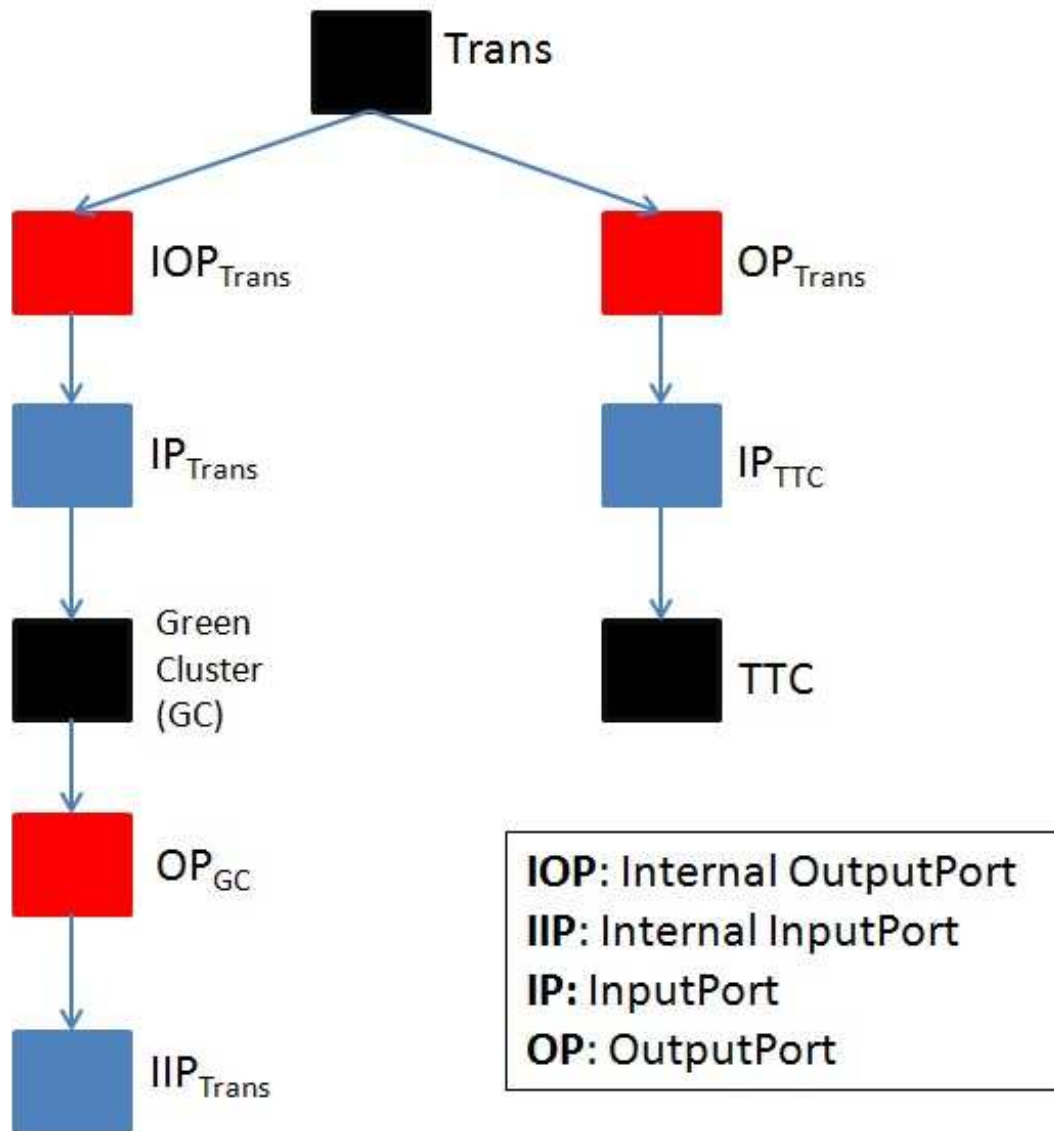


Figure 3.13: Transportation-level graph (embedded inside the city-level graph).

tex, it will obtain the cluster-level graph and traverse it before continuing down the branch.

At this point, the original message has been passed along the edges of the graph and reached its destination Component; College Park Metro Station. The message is “consumed” at the College Park Node and a SystemMessage is generated to be passed along to the TTC Controller. Once the cluster-level graph traversal is complete, the Graph Manager returns to the Transportation-level graph to complete the internal branch and move to the external branch. When the GraphManager reaches the TTC vertex of the external branch, it will traverse the controller-level graph as shown in Figure 3.15

The TTC Controller receives the SystemMessage and checks it against the defined rules. A new Message is created that closes College Park metro station. Note that at the end of the graph, the GraphManager will process the Transportation vertex again, creating a loop. A limit of five visits to a Component was specified in order to stop the GraphManager from traversing indefinitely. The result of this scenario is that College Park has a waterAcc=3.0 and has stationStatus = closed.

---

output

---

```
Attributes of College Park
=====
Attribute: isStationAboveGround = null
Attribute: stationLocation = null
Attribute: waterAcc = 3.0
Attribute: stationElevation = null
Attribute: stationStatus = closed
Attribute: stationElectricReq = null
Attribute: stationName = null
Attribute: serviceRate = null
Attribute: stationElectricBy = null
```

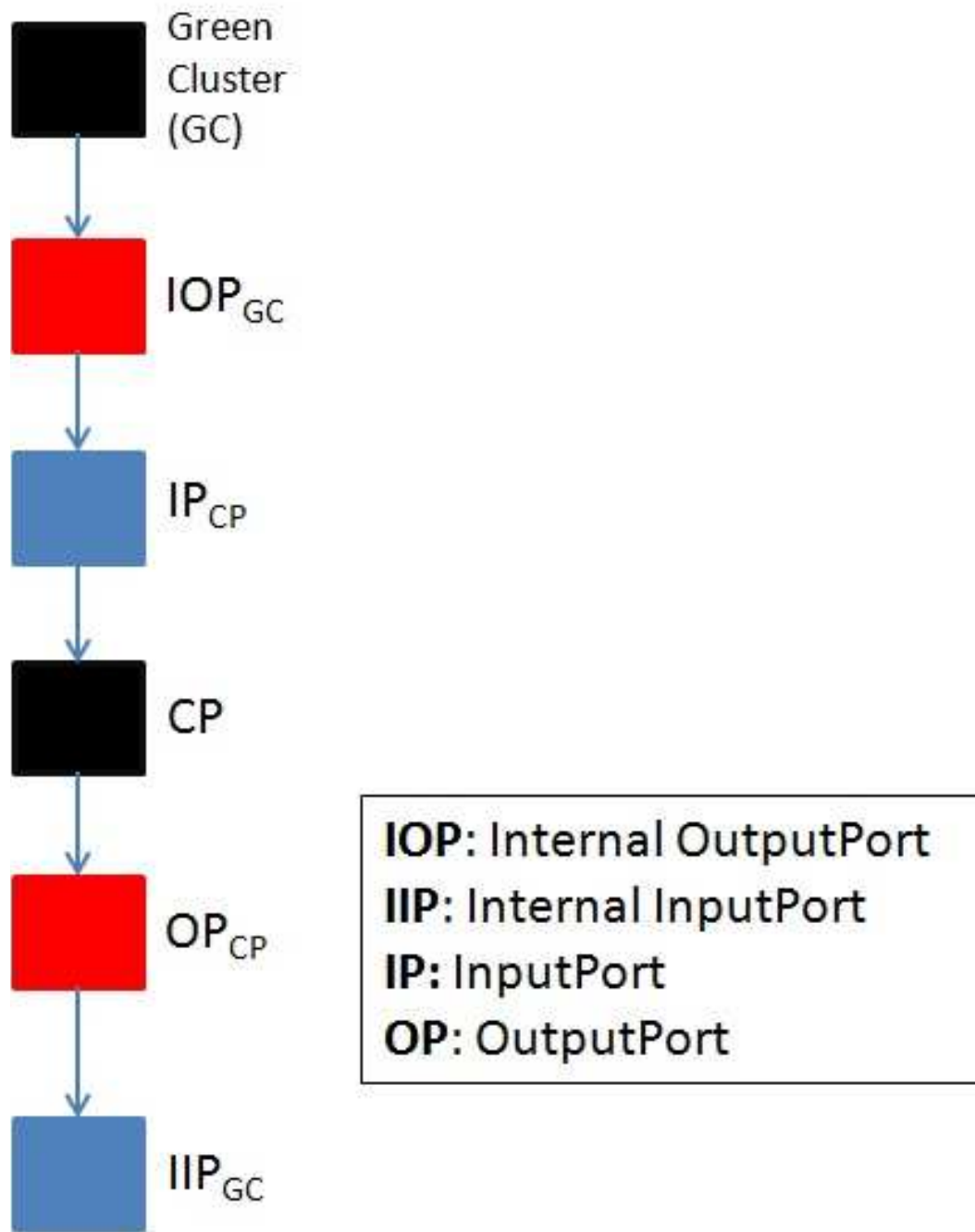


Figure 3.14: Cluster-level graph (embedded inside the transportation-level graph).

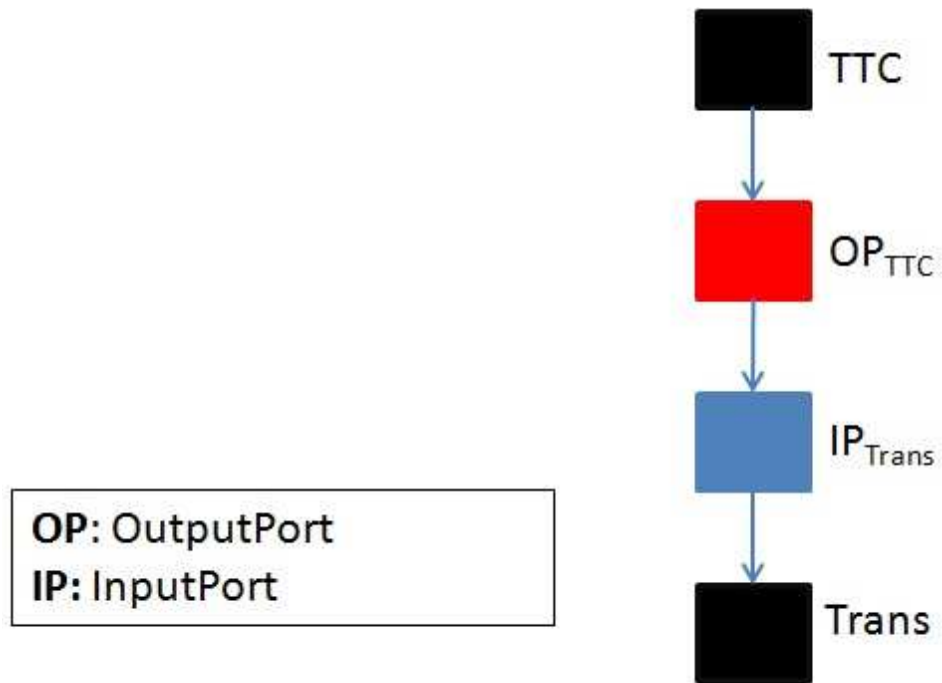


Figure 3.15: Controller-level graph.

**Three-Station Scenario.** In a slightly more difficult scenario, we add an additional 2 stations to the Green Line Cluster: Greenbelt and PG Plaza. In addition, we add extra rules that state:

- If Greenbelt is closed, then College Park is closed
- If College Park is closed, then PG Plaza is closed

The process is identical to the 1 station scenario, however the cluster-level graph is now expanded to have 3 branches. Figure 3.16 shows the cluster-level graph. Since there is no notion of an ordered cluster in this implementation of the framework, the

GraphManager arbitrarily chooses an order for processing the station nodes. While the implementation is straightforward, this leads to unnecessary cycle traversals, and in a large network the required computation could be very time and resource consuming. Nevertheless, eventually the information will be propagated to the correct Components.

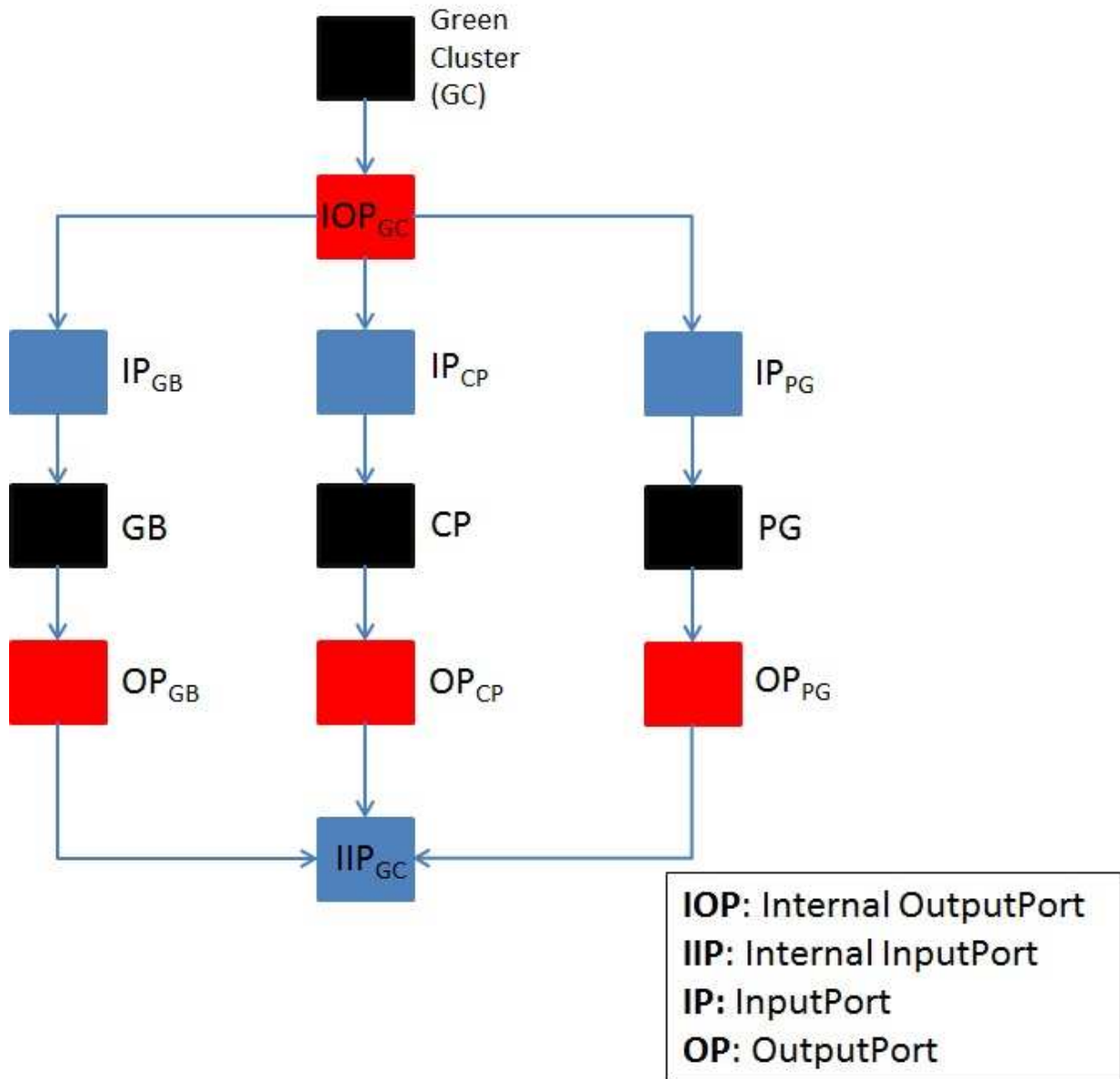


Figure 3.16: Three-station Cluster-Level Graph.

In this example, a weather event hits Greenbelt with 3 inches of rain. The rainfall closes Greenbelt, which subsequently closes College Park and then PG Plaza. The result is that stationStatus=closed for all 3 stations, however only Greenbelt has waterAcc = 3.0.

---

output

---

Attributes of Greenbelt

=====  
Attribute: isStationAboveGround = null  
Attribute: stationLocation = null  
Attribute: waterAcc = 3.0  
Attribute: stationElevation = null  
Attribute: stationStatus = closed  
Attribute: stationElectricReq = null  
Attribute: stationName = null  
Attribute: serviceRate = null  
Attribute: stationElectricBy = null

Attributes of College Park

=====  
Attribute: isStationAboveGround = null  
Attribute: stationLocation = null  
Attribute: waterAcc = null  
Attribute: stationElevation = null  
Attribute: stationStatus = closed  
Attribute: stationElectricReq = null  
Attribute: stationName = null  
Attribute: serviceRate = null  
Attribute: stationElectricBy = null

Attributes of PG Plaza

=====  
Attribute: isStationAboveGround = null  
Attribute: stationLocation = null  
Attribute: waterAcc = null  
Attribute: stationElevation = null  
Attribute: stationStatus = closed  
Attribute: stationElectricReq = null  
Attribute: stationName = null  
Attribute: serviceRate = null  
Attribute: stationElectricBy = null

---

## 3.6 Summary

In this chapter, we used the MetaComponent and message passing software design patterns to implement a Java version of the SoS framework described in Chapter 2. The Java implementation provides users with the ability to create clusters of nodes, allowing for different viewpoints and levels of abstraction. The model also easily identifies cross cutting issues and can accurately model cascading effects through the city.

However, there are a few weaknesses of the Java framework. First, improvements can be made in the rule processing portion of the model. Currently, there is an exponential growth in the number of controllers needed for the model. In a real world situation, there could be hundreds of individual systems in a city, requiring thousands of controllers. This makes rule management nearly impossible, not to mention the enormous amount of computing power required to run a simulation. Secondly, manual setup of the objects of a model can be very tedious. As evidenced in the example, a simple city with one system, five nodes, and a controller took over 150 lines of code. This, combined with the exponential growth of controllers makes setting up a real-world example currently infeasible. Finally, another improvement that can be made is to make the model multi-threaded. Currently, the model processes a single rule at a time, and will create cascading messages one at a time. To accurately create a model of a city, each message should be its own thread.

## Chapter 4

# Implementation of Framework using Ontology and Rule-Based Modeling

### 4.1 Description of Ontologies and RDF Graphs

The concept of ontology-based modeling was used to formally define city systems. An ontology represents a set of concepts within a domain using a shared vocabulary to denote the types, properties, and relationships between the concepts [10]. An ontology can be viewed as a definition of a meta-model for a particular domain.

For this project we used Apache Jena to implement the Web Ontology Language (OWL) and Resource Description Framework (RDF) ontology language. In this framework, a model or domain is defined by an ontology. The key elements of an ontology are [1]:

- **Classes:** Concepts or objects in the domain.
- **Properties:** Attributes or features of a class.
- **Individuals:** Instances of a class.

An RDF graph is a labeled, directed multigraph defined by sets of arcs called statements. Statements are also called RDF Triples and have a structure (S, P, O)



where:

- **S = Subject:** The class where the arc starts from.
- **P = Properties:** The property labeling the arc.
- **O = Object:** The class or value where the arc ends.

Subjects are denoted by Universal Resource Identifiers (URIs). Each property will have a specific meaning and may define its permitted values, the types of resources it can describe, and its relationship with other properties. Objects are denoted by a “string” or URI.

The graph-based nature of RDF means that it can resolve circular references, an inherent problem of the hierarchical structure of XML [26]. An assertion is the smallest expression of useful information. RDF captures assertions made in simple sentences by connecting a subject to an object and a verb, as shown in Figure 4.1.

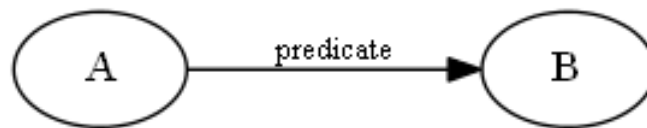


Figure 4.1: Example of RDF triple where node A is a subject, predicate is a verb, and node B is an object.

A statement represents some knowledge about the model. The knowledge and relationships can be constrained with a set of rules contained in the ontology. These rules can also be used to infer new information about the model and can be used to control how a model behaves. Jena provides support for several ontology reasoners such as Pellet, however for this project, we will be using a simple, generic reasoner.

## 4.2 Refactoring Framework to Jena

### 4.2.1 Mapping Framework Concepts to Ontology-Based Concepts

The first step to refactoring is understanding how concepts from our previous model fit in to the ontology based model.

<b>Framework Model</b>	<b>Ontology Model</b>
City System	Ontology
Node	Class/Properties
Cluster	RDFLists
Rule	Rules
Ports	N/A
Message	N/A

Table 4.1: Mapping between frameworks.

Table 4.1 shows the mapping between the frameworks. Each system in the city will have its own ontology. Therefore, the components of the city will be represented as a class each with a set of properties that determine the state of that object. Clusters will be represented by a concept called *RDFLists*, which will be discussed later in the paper. Most notable in the refactoring to ontology based modeling is the simplicity of implementing rules. Instead of having 4 or 5 different classes to propagate rules, the only thing that is needed is a set of rules attached to the ontology. The creation of rules will be discussed later.

## 4.2.2 Creating Ontologies

In order for the rules to propagate correctly, the classes, properties, and relationships between classes must be defined properly and free of inconsistencies and redundancies. Since the goal of the research is to understand the cascading effects of a failure between city systems, certain concepts and objects that do not relate to the project goal can be abstracted.

Once the ontology was created, Individuals and statements were added to the model. In order to serialize the data, Individuals were stored in XML files. To create the XML files, regular Java classes were created and had the same structure as its ontology counterpart. In other words, the fields and members of the Java classes are mirrored by the properties associated with ontology classes. To facilitate the creation of the XML file, the Java class was annotated using the Java Architecture for XML Binding (JAXB) framework [13]. Instances of each class were created and outputted to an XML file.

The XML was then parsed and Individuals were created directly from the file. Additional steps were needed to ensure the relationships between objects were linked properly. RDFLists were then created in order to represent ordered lists of objects. An RDFList is a special object in the OWL framework and has two properties; first and rest. First is the value at that particular index, while rest is a pointer to another RDFList object, and represents the next object in the list. As such, RDFLists act similar to a one-way linked list and can be used to traverse and

---

```

term, ... term -> hterm, ... hterm    // forward rule
hterm      := term
term       := (node, node, node)      // triple pattern
           or (node, node, functor)   // extended triple pattern
           or builtin(node, ... node) // invoke procedure
functor    := functorName(node, ... node) // structured literal
node       := uri-ref                 // e.g. http://foo.com/eg
           or prefix:localname        // e.g. rdf:type
           or <uri-ref>               // e.g. <myscheme:myuri>
           or ?varname                 // variable
           or 'a literal'              // a plain string literal
           or 'lex'^^typeURI           // a typed literal
           or number                   // e.g. 42 or 25.5

```

---

Table 4.2: Format for Jena Rules.

obtain specific objects within a list.

### 4.2.3 Defining Rules

The last step of creating the ontology model was to create the rules that govern the model behavior. Jena’s reasoning API allows for additional information to be inferred from instance data and class descriptions. This gives Jena a distinct advantage over other frameworks since rule propagation is inherently built in.

Table 4.2 summarizes the format (or grammar) for the specification of rules in Jena [1]. If a statement in the RDF model matches the triple pattern defined on the left side of the arrow, then the additional statements on the right hand side of the arrow can be inferred. Rules can also be used to change the value of a particular statement in the model. A change in the value of a property may trigger additional rules to be fired, thus creating a cascading effect throughout the model.

Unfortunately, OWL and RDF graphs were designed with the assumption that the model would be static. Therefore, there is little to no support for dynamic modeling and simulation. To get around this, builtins were used to manipulate the model. Builtins are essentially Java functions that take a RDF statement and manipulate it outside of the reasoning engine. The output of a builtin is then used inside the reasoning engine. For the purposes of this project, we created builtins to change values from true to false. We also created logic to iterate through RDFLists and modify elements in a list.

### 4.3 Implementing the City Model using Jena

Continuing with the city model, a transportation ontology representing the DC metro was created. In addition to the stations of the metro system, classes were created for the train, track, station, and trips of the subway system. Figure 4.2 shows the classes and relationships between the classes in the model.

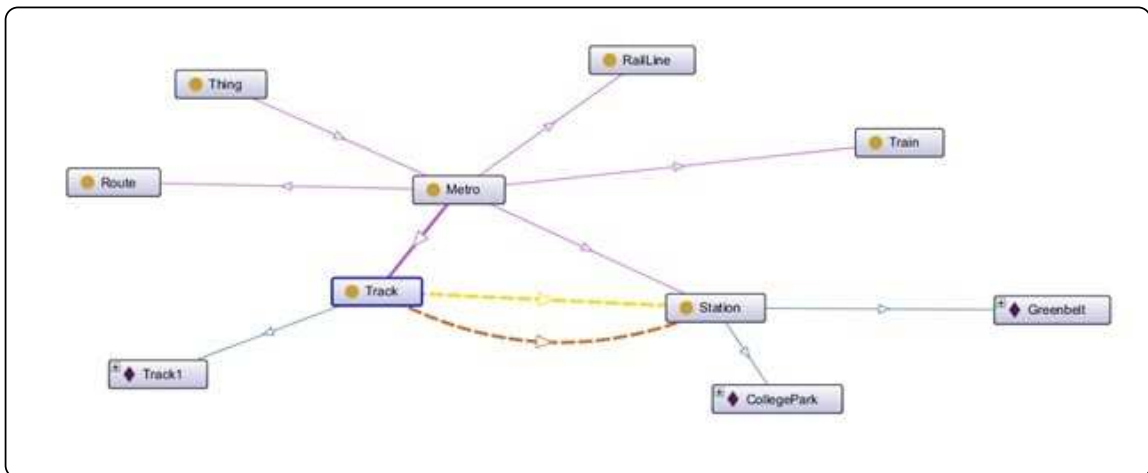


Figure 4.2: Transportation Ontology, created using Protégé [16]

Properties for each class are defined below in Table 4.3. Important to note is the properties `tripStations`, `tripTracks`, and `arrivalTimes` are RDFLists of type `Station`, `Track`, and `Date` respectively.

Class	Properties
Station	<code>stationName</code> , <code>stationLocation</code> , <code>serviceRate</code> , <code>stationElevation</code> , <code>isStationAboveGround</code> , <code>stationElectricBy</code> , <code>stationElectricReq</code> , <code>stationStatus</code> , <code>waterAcc</code> , <code>snowAcc</code> ;
Track	<code>trainName</code> , <code>trainCapacity</code> , <code>trainStatus</code> , <code>trainElectricReq</code> , <code>onTrack</code> , <code>atStation</code> , <code>trainTrip</code> , <code>waterAcc</code> , <code>snowAcc</code>
Trip	<code>tripStations</code> , <code>tripTracks</code> ; <code>tripName</code> , <code>arrivalTimes</code>
Train	<code>trainName</code> , <code>trainCapacity</code> , <code>trainStatus</code> , <code>trainElectricReq</code> , <code>onTrack</code> , <code>atStation</code> , <code>trainTrip</code>

Table 4.3: Class Properties of Transportation Ontology

**Simplified Model of the Green Line, Washington DC Metro.** As a proof of concept, the green line of the DC metro was modeled. The green line consists of 21 stations, 40 track segments (20 in each direction), and two trips (one in each direction). The notion of time and location were abstracted from the model due to time constraints and complexity. The resulting RDF graph consisted of 986 initial statements.

**Rules.** The rules for the city system are defined as follows:

- A track or station is above/below ground if the elevation is greater/less than 0
- If a track or station has water accumulation greater than 6 inches, then its status becomes false.

- If a track or station is above ground and has 12 inches of snow accumulation, then its status becomes false.
- When the water or snow accumulation drops below the threshold, the status is returned back to true.
- If a track status is false, then all subsequent tracks are also set to false.
- If a station status is false, then all subsequent tracks are also set to false.

A complete specification for the Jena rule definitions can be found in Appendix B.

**Weather Event Scenario.** Let us consider the following weather event scenario.

A thunderstorm hits the Washington DC area, bringing 3 inches of rain and 40 mile per hour winds to the area. Below is the rule created to represent this event:

```
[WeatherTrans: (?x rdf:type af:Track) (?x af:
  isTrackAboveGround "true"^^xs:boolean) (?x af:waterAcc ?y)
  sum(?y, 3.0, ?newY) noValue(?x af:ruleFiredFor af:Weather
) -> remove(2) (?x af:waterAcc ?newY) (?x af:ruleFiredFor
af:Weather)]
```

This rule changed the status of all above ground tracks and distribution lines to false. In addition, builtin functions in Java iterated through all trips and looked for closed tracks. If the function encountered a closed track, it iterated through the rest of the list and closed the subsequent tracks. After checking for all rules, the inferred transportation model grew to 1109 statements. An additional 35 statements were modified.

## Chapter 5

### Conclusions and Future Work

#### 5.1 Conclusions

The conclusions of this exploratory work are as follows:

1. This paper explores the use of higraphs and applies SoS techniques to create a framework for modeling a SoS, using a city as an example. The paper defines the different aspects of the model and how the model passes information to different portions of the system. The model provides users with the ability to create clusters of nodes, allowing for different viewpoints and levels of abstraction. The model also easily identifies cross cutting issues and can accurately model cascading effects through the city. Given the complexities and scale of actual system of systems, this framework model needed to be implemented on a computer for simulation and management purposes.
2. With this realization in place, we then explored the use of Java design patterns to create a software package from the SoS framework. Through the use of the MetaComponent and data flow processing design patterns we were able to simulate the effect of an external event on a city. One of the main weaknesses of this software package lies in its approach to rule processing. With the design of the framework and software architecture, there is an exponential



growth in the number of controllers needed for the model. In a real world situation, there could be hundreds of individual systems in a city, requiring thousands of controllers. This makes rule management nearly impossible, not to mention the enormous amount of computing power required to run a simulation. Another improvement that can be made is to make the model multi-threaded. The current model processes a single rule at a time, and will create cascading messages one at a time. To accurately create a model of a city, each message should be its own thread. Such an implementation would capture the many concurrent processes occurring in cities, but would complicate the model implementation through the need to synchronize processes at key points.

3. In a first step toward addressing these issues, the entire framework was refactored and recreated using ontology and rule based methodologies. In particular, we used the Apache Jena RDF framework for its powerful reasoning tool which allows for rule checking and inference models. Using Jena allowed us to create formal definitions of objects in our system, which allowed us to create rules to manage the behavior of our system and infer new information based on the current state of the system. Jena's reasoning tool allowed the state of multiple objects to be modified at once, and processed those state changes at the same time.
4. There were some difficulties with using Apache Jena. First, RDF and ontology modeling was intended for static models, meaning the properties of the model

do not change. Thus, there is little to no support for changing the values of a statement. In order to change a value, we had to find the exact statement in the model, remove it, and enter in a completely new statement. Another issue with Jena is the inability to create rules that affect ordered lists. In the Java version of the framework, clusters and nodes were treated as components, and the framework was able to treat them the same. In Jena, there is no way to apply a rule to all elements of a list.

## 5.2 Future Work

1. Looking ahead, we want a framework that combines the flexibility of user-created nodes and clusters in the SoS framework with the rule processing capabilities of an ontology and rule based framework. Future work needs to include support for ontology integration. One approach would be to create an umbrella ontology that combines all of the city ontologies with a set of rules describing the relationships between ontologies. Another interesting topic would be to create a weather ontology as well as introduce spatial knowledge to the model.
2. Future work can be done in the processing of cycles. Instead of setting a visit threshold, the traversal should cease when there are no new messages created by the System or the Controller. Additional work can be done to Clusters to introduce the notion of dependence within objects in a cluster. This could reduce the number of traversals in the model.

## Bibliography

- [1] 2014. An Introduction to RDF and the Jena RDF API, Apache Jena – The Apache Software Foundation, (Accessed April 3, 2014).
- [2] Anderson, C.F., Battjes J.A., Daniel D.E., et al. The New Orleans Hurricane Protection System: What Went Wrong and Why. *American Society of Civil Engineers*, 2007.
- [3] Austin M.A. *Emerging Strategies of Systems Engineering Development*. Reading from Lecture Notes for ENSE621/ENPM641 Systems Concepts, Issues and Processes, Institute for Systems Research, University of Maryland, College Park, MD 20742, 2012.
- [4] Austin M.A. *System of Systems (SoS)*. Reading from Lecture Notes for ENSE621/ENPM641 Systems Concepts, Issues and Processes, Institute for Systems Research, University of Maryland, College Park, MD 20742, 2012.
- [5] Berkowitz P. We Went into the Mall and Began Looting: A Letter on Race, Class, and Surviving the Hurricane” MRZine, 9 Sept. 2005. Web. 14 Apr. 2014.
- [6] Burdett R. and Sudjic D. *The Endless City*. Phaidon Press, 2008.
- [7] Dupagne A. and Teller A. Hypergraph Formalism for Urban Form Specification. In *COST C4 Final Conference*, Kiruna, September 21-22 1998.
- [8] Fogarty K. System Modeling and Traceability Applications of the Higraph Formalism. *MS Thesis in System Engineering, Institute for Systems Research*, May 2006.
- [9] Fogarty K. and Austin M.A. System Modeling and Traceability Applications of the Higraph Formalism. *Systems Engineering*, 12(2):117–140, 2009.
- [10] Gruber T.R. *A Translation Approach to Portable Ontology Specifications*. Stanford University, CA, September 1992.
- [11] Harel D. On Visual Formalisms. *Communications of the ACM*, 31:514–530, 1988.
- [12] Headway Software Inc. Closed Loop Development with Headway ReView. June 2001.
- [13] JAXB Reference Implementation Project Kenai, JAXB Reference Implementation Project Kenai. GlassFish Community, 2014 (Accessed April 3, 2014).
- [14] JGraphT. See <http://www.jgrapht.org> (Accessed April 25, 2014).
- [15] Paige R.F. Heterogeneous Specifications and their Application to Software Development. Technical report, Toronto, Canada, August 1995.
- [16] Protege Ontology Editor and Knowledge Acquisition System. For details, see <http://protege.stanford.edu>. 2003.
- [17] Ramaswamy M. and Sarkar S. Using Directed Hypergraphs to Verify Rule-Based Expert Systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):221–237, March-April 1997.

- [18] Selberg S. and Austin M.A. Toward an Evolutionary System of Systems Architecture. In *Proceedings of Eighteenth Annual International Symposium of The International Council on Systems Engineering (INCOSE 2008)*, Utrecht, The Netherlands, June 15-19 2008.
- [19] SysML Partners. "Systems Modeling Language Specification, Version 1.0 alpha, November 2005. See <http://www.sysml.org/artifacts/specs/SysMLv1.0a-051114R1.pdf>.
- [20] Timeline: Who Knew When the Levees Broke." NPR. NPR, 10 Feb. 2006. Web. 14 Apr. 2014.
- [21] Tonn S. A System of Systems (SoS) Model and Applications, ENSE 623 Class Project Report. *Institute for Systems Research, University of Maryland*, Fall Semester 2009.
- [22] Unified Modeling Language (UML). See <http://www.omg.org/uml>, 2003.
- [23] Venn Diagram Survey, The Electronic Journal of Combinatorics, June, 2005. For details, see <http://www.combinatorics.org/files/Surveys/ds5/VennSymmEJC.html> (Accessed April 3, 2014).
- [24] Wissen, M. and Ziegler, J. A Methodology for the Component-Based Development of Web Applications. In *Proceedings of 10th Int. Conf. on Human-Computer Interaction (HCI International 2003)*, volume 1, Crete, Greece, 2003.
- [25] World Population Projected to Reach 9.6 Billion by 2050, United Nations Department of Economic and Social Affairs, UN News Center. UN, 13 June 2013. Web. 03 Apr. 2014.
- [26] eXtensible Markup Language (XML). See <http://www.w3.org/XML>. 2004.

# Appendix A: Java Framework Model Setup

This appendix contains the Java code for the city model setup.

---

```
public class CitySystemMain {
    public static void main(String[] args) {

        // Set up overall City System model ...

        MetaComponent<Message> citySystem = new MetaComponent<Message>();
        citySystem.setName("Washington, D.C.");

        // Set up City Ports ...

        InputPort<Message>  cityInputPort  =
            new InputPortImpl <Message>(citySystem);
        OutputPort<Message> cityOutputPort =
            new OutputPortImpl <Message>(citySystem);

        cityInputPort.setName("cityInputPort");
        cityOutputPort.setName("cityOutputPort");

        citySystem.addExternalPort(cityInputPort);
        citySystem.addExternalPort(cityOutputPort);

        // Transportation System Setup ...

        MetaComponent<Message> transSystem = new MetaComponent<Message>();
        transSystem.setName("DC Metro");

        // Set up Transportation Ports

        InputPort<Message>  transInputPort =
            new InputPortImpl <Message>(citySystem);
        OutputPort<Message> transOutputPort =
            new OutputPortImpl <Message>(citySystem);

        transInputPort.setName("transInputPort");
        transOutputPort.setName("transOutputPort");

        transSystem.addExternalPort(transInputPort);
        transSystem.addExternalPort(transOutputPort);

        // Add it to the City

        citySystem.addSubComponent(transSystem);

        // Add connection from City to trans

        citySystem.addConnection(cityInputPort, transInputPort);
        citySystem.addConnection(transOutputPort, cityOutputPort);
    }
}
```

```

// Green Line Setup ...

Cluster<Message> greenLine = new Cluster<Message>();
greenLine.setName("Green Line Cluster");

// Add cluster Ports

InputPort<Message> greenLineIport =
    new InputPortImpl<Message>(greenLine);
OutputPort<Message> greenLineOport =
    new OutputPortImpl<Message>(greenLine);
greenLineIport.setName("greenLineIPort");
greenLineOport.setName("greenLineOPort");

greenLine.addExternalPort(greenLineIport);
greenLine.addExternalPort(greenLineOport);

// Create metro stations

StationNode<Message> greenbeltStation =
    new StationNode<Message>("Greenbelt");
StationNode<Message> collegeParkStation = new
    StationNode<Message>("College Park");
StationNode<Message> pgPlazaStation =
    new StationNode<Message>("Prince George's Plaza");
StationNode<Message> wHyattStation =
    new StationNode<Message>("West Hyattsville");
StationNode<Message> fortTottenStation =
    new StationNode<Message>("Fort Totten");

// Add to stations to the cluster

greenLine.addSubComponent(greenbeltStation);
greenLine.addSubComponent(collegeParkStation);
greenLine.addSubComponent(pgPlazaStation);
greenLine.addSubComponent(wHyattStation);
greenLine.addSubComponent(fortTottenStation);

// Add Input and Output Ports to each station

InputPort<Message> gbStationIport =
    new InputPortImpl<Message>(greenbeltStation);
OutputPort<Message> gbStationOport =
    new OutputPortImpl<Message>(greenbeltStation);
gbStationIport.setName("gbStationIPort");
gbStationOport.setName("gbStationOPort");

greenbeltStation.addExternalPort(gbStationIport);
greenbeltStation.addExternalPort(gbStationOport);

InputPort<Message> cpStationIport =
    new InputPortImpl<Message>(collegeParkStation);
OutputPort<Message> cpStationOport =
    new OutputPortImpl<Message>(collegeParkStation);

```

```

cpStationIport.setName("cpStationIPort");
cpStationOport.setName("cpStationOPort");

collegeParkStation.addExternalPort(cpStationIport);
collegeParkStation.addExternalPort(cpStationOport);

InputPort<Message> pgStationIport =
    new InputPortImpl<Message>(pgPlazaStation);
OutputPort<Message> pgStationOport =
    new OutputPortImpl<Message>(pgPlazaStation);
pgStationIport.setName("pgStationIPort");
pgStationOport.setName("pgStationOPort");

pgPlazaStation.addExternalPort(pgStationIport);
pgPlazaStation.addExternalPort(pgStationOport);

InputPort<Message> whStationIport =
    new InputPortImpl<Message>(wHyattStation);
OutputPort<Message> whStationOport =
    new OutputPortImpl<Message>(wHyattStation);
whStationIport.setName("whStationIPort");
whStationOport.setName("whStationOPort");

wHyattStation.addExternalPort(whStationIport);
wHyattStation.addExternalPort(whStationOport);

InputPort<Message> ftStationIport =
    new InputPortImpl<Message>(fortTottenStation);
OutputPort<Message> ftStationOport =
    new OutputPortImpl<Message>(fortTottenStation);

ftStationIport.setName("ftStationIPort");
ftStationOport.setName("ftStationOPort");

fortTottenStation.addExternalPort(whStationIport);
fortTottenStation.addExternalPort(whStationOport);

// Add connections from the stations to the cluster

greenLine.addConnection(greenLineIport, gbStationIport);
greenLine.addConnection(greenLineIport, cpStationIport);
greenLine.addConnection(greenLineIport, pgStationIport);
greenLine.addConnection(greenLineIport, whStationIport);
greenLine.addConnection(greenLineIport, ftStationIport);

greenLine.addConnection(gbStationOport, greenLineOport);
greenLine.addConnection(cpStationOport, greenLineOport);
greenLine.addConnection(pgStationOport, greenLineOport);
greenLine.addConnection(whStationOport, greenLineOport);
greenLine.addConnection(ftStationOport, greenLineOport);

// Finally, add greenLine to Trans and add Connections

transSystem.addSubComponent(greenLine);

```

```

transSystem.addConnection(transInputPort, greenLineIport);
transSystem.addConnection(greenLineOport, transOutputPort);

// =====
// WTC Controller Setup ...
// =====

Controller<Message> wtcController = new
    Controller<Message>(1,1, new WTCEngine<Message>());

wtcController.setName("WTC Controller");
citySystem.addSubComponent(wtcController);

// Add connection from trans to Controller

transSystem.addConnection(transOutputPort, wtcController.getInputPort(0));

// Add connection from Controller to trans

transSystem.addConnection(wtcController.getOutputPort(0), transInputPort);

// Create the flood

Message flood = new ControllerMessage("Rain", "Sandy", "waterAcc",
                                       (Double) 3.0, "College Park");
cityInputPort.setValue(flood);
citySystem.process();
}
}

```

---



# Appendix B: Transportation Ontology Rules

This appendix contains the Jena rules that accompany the transportation ontology.

---

```
@prefix af: <http://cityModel.org/transportation#>.
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix xs: http://www.w3.org/2001/XMLSchema#

// =====
// Weather Event
// =====

// [Weather: (?x rdf:type af:Track) (?x af:isTrackAboveGround "true"^^xs:boolean)
//          (?x af:waterAcc ?y) sum(?y, 3.0, ?newY)
//          noValue(?x af:ruleFiredFor af:Weather) ->
//          remove(2) (?x af:waterAcc ?newY) (?x af:ruleFiredFor af:Weather)]

// =====
// Track rules
// =====

[UndergroundTrack: (?x rdf:type af:Track) (?x af:trackElevation ?y)
                    lessThan(?y, 0.0) ->
                    (?x af:isTrackAboveGround "false"^^xs:boolean)]

[AbovegroundTrack: (?x rdf:type af:Track) (?x af:trackElevation ?y)
                    greaterThan(?y, 0.0) ->
                    (?x af:isTrackAboveGround "true"^^xs:boolean)]

[Electrified:       (?x rdf:type af:Track) (?x af:trackElectricAvail ?y)
                    greaterThan(?y, 0.0) ->
                    (?x af:isElectrified "true"^^xs:boolean)]

[NotElectrified:   (?x rdf:type af:Track) (?x af:trackElectricAvail ?y)
                    equal(?y, 0.0) ->
                    (?x af:isElectrified "false"^^xs:boolean)]

[WaterFail:        (?x rdf:type af:Track) (?x af:waterAcc ?y) greaterThan(?y, 2.0)
                    (?x af:trackStatus "true"^^xs:boolean)
                    getChangeStatus("true"^^xs:boolean,?f)
                    noValue(?x af:trackStatus ?f) -> remove(3) (?x af:trackStatus ?f)]

[SnowFail:         (?x rdf:type af:Track) (?x af:snowAcc ?y) greaterThan(?y, 12.0)
                    (?x af:trackStatus "true"^^xs:boolean)
                    getChangeStatus("true"^^xs:boolean,?f)
                    noValue(?x af:trackStatus ?f) -> remove(3) (?x af:trackStatus ?f)]

[PrecipNotFail:   (?x rdf:type af:Track) (?x af:waterAcc ?y)
                    (?x af:snowAcc ?z) lessThan(?z, 12.0) lessThan(?y, 2.0)
                    (?x af:trackStatus "false"^^xs:boolean)
                    getChangeStatus("false"^^xs:boolean,?f)
                    noValue(?x af:trackStatus ?f) -> remove(5) (?x af:trackStatus ?f)]
```

```
// =====  
// Station rules  
// =====  
  
[AboveGroundStation: (?x rdf:type af:Station) (?x af:stationElevation ?y)  
  greaterThan(?y, 0.0) ->  
  (?x af:isStationAboveGround "true"^^xs:boolean)]  
  
[BelowGroundStation: (?x rdf:type af:Station) (?x af:stationElevation ?y)  
  lessThan(?y, 0.0) ->  
  (?x af:isStationAboveGround "false"^^xs:boolean)]
```

---

## Appendix C: City Network Modeling

This appendix contains the java code for specification and execution of the city network problem described in Chapter 3.

---

```
/*
 * =====
 * TestCityNetwork.java: Develop a simplified model of a transportation
 * network within a city. Process the consequences of a weather event.
 *
 * Written by: Alan Nguyen                                April 2014
 * =====
 */

package demo;

import port.*;
import component.*;
import engine.*;
import org.jgrapht.*;
import wire.*;
import message.*;

public class TestCityNetwork {
    public static void main(String[] args) {

        System.out.println("*** ");
        System.out.println("*** Begin TestCityNetwork() .... ");
        System.out.println("*** =====");
        System.out.println("*** ");

        // Create City ...

        MetaComponent<Message> citySystem = new MetaComponent<Message>();
        citySystem.setName("Washington, D.C.");

        // Set up city ports ...

        InputPort<Message> cityInputPort = new InputPortImpl <Message>(citySystem);
        cityInputPort.setName("cityInputPort");

        citySystem.addExternalPort(cityInputPort);

        // =====
        // Create transportation system ...
        // =====

        MetaComponent<Message> transSystem = new MetaComponent<Message>();
        transSystem.setName("DC Metro");

        // Set up transportation ports
```

```

InputPort<Message> transInputPort = new InputPortImpl <Message>(transSystem);
transInputPort.setName("transInputPort");

OutputPort<Message> transOutputPort = new OutputPortImpl <Message>(transSystem);
transOutputPort.setName("transOutputPort");

transSystem.addExternalPort(transInputPort);
transSystem.addExternalPort(transOutputPort);

// Add it to the City

citySystem.addSubComponent(transSystem);

// Add connection from City to trans

citySystem.addConnection(cityInputPort, transInputPort);

// =====
// WTC controller setup ...
// =====

Controller<Message> ttcController = new Controller<Message>( 1, 1,
                                                            new TTCEngine<Message>());
ttcController.setName("TTC Controller");
citySystem.addSubComponent(ttcController);

// Add connection from trans to Controller

transSystem.addConnection(transOutputPort, ttcController.getInputPort(0));

// Add connection from Controller to trans

ttcController.addConnection(ttcController.getOutputPort(0), transInputPort);

// =====
// Green line setup ...
// =====

Cluster<Message> greenLine = new Cluster<Message>();
greenLine.setName("Green Line Cluster");

// Add cluster ports

InputPort<Message> greenLineIport = new InputPortImpl<Message>(greenLine);
greenLineIport.setName("greenLineIport");
OutputPort<Message> greenLineOport = new OutputPortImpl<Message>(greenLine);
greenLineOport.setName("greenLineOport");

greenLine.addExternalPort(greenLineIport);
greenLine.addExternalPort(greenLineOport);

// Add it to the TransSystem

transSystem.addSubComponent(greenLine);

```

```

// Add a Connection from trans->cluster and cluster->trans

transSystem.addConnection(transInputPort, greenLineIport);
transSystem.addConnection(greenLineOport, transOutputPort);

// =====
// Metro Station setup ...
// =====

StationNode<Message> collegeParkStation = new StationNode<Message>("College Park");

// Add Input and Output Ports

InputPort<Message> cpStationIport = new InputPortImpl<Message>(collegeParkStation);
cpStationIport.setName("cpStationIport");
OutputPort<Message> cpStationOport = new OutputPortImpl<Message>(collegeParkStation);
cpStationOport.setName("cpStationOport");
collegeParkStation.addExternalPort(cpStationIport);
collegeParkStation.addExternalPort(cpStationOport);

StationNode<Message> greenBeltStation = new StationNode<Message>("Greenbelt");

// Add Input and Output Ports

InputPort<Message> gbStationIport = new InputPortImpl<Message>(greenBeltStation);
gbStationIport.setName("gbStationIport");

OutputPort<Message> gbStationOport = new OutputPortImpl<Message>(greenBeltStation);
gbStationOport.setName("gbStationOport");
greenBeltStation.addExternalPort(gbStationIport);
greenBeltStation.addExternalPort(gbStationOport);

StationNode<Message> pgPlazaStation = new StationNode<Message>("PG Plaza");

// Add Input and Output Ports

InputPort<Message> pgStationIport = new InputPortImpl<Message>(pgPlazaStation);
pgStationIport.setName("pgStationIport");
OutputPort<Message> pgStationOport = new OutputPortImpl<Message>(pgPlazaStation);
pgStationOport.setName("pgStationOport");
pgPlazaStation.addExternalPort(pgStationIport);
pgPlazaStation.addExternalPort(pgStationOport);

// Add to Cluster

greenLine.addSubComponent(greenBeltStation);
greenLine.addSubComponent(collegeParkStation);
greenLine.addSubComponent(pgPlazaStation);

// Add Connections

greenLine.addConnection(greenLineIport, gbStationIport);
greenLine.addConnection(gbStationOport, greenLineOport);

```

```

greenLine.addConnection(greenLineIport, cpStationIport);
greenLine.addConnection(cpStationOport, greenLineOport);

greenLine.addConnection(greenLineIport, pgStationIport);
greenLine.addConnection(pgStationOport, greenLineOport);

// Create the flood ...

Message flood = new ControllerMessage("Rain", "Sandy",
                                     "waterAcc", (Double) 3.0, "Greenbelt");
cityInputPort.setValue(flood);

// Let's see the graph and then process it!

System.out.println(greenLine.graphManager.getGraph());
citySystem.process();

greenBeltStation.listAttributes();
collegeParkStation.listAttributes();
pgPlazaStation.listAttributes();

System.out.println("*** ");
System.out.println("*** =====");
System.out.println("*** End TestCityNetwork() .... ");
System.out.println("*** ");
}
}

```

---