ABSTRACT

Title of thesis: **ONTOLOGY-ENABLED TRACEABILITY MODELS FOR ENGINEERING SYSTEMS DESIGN AND MANAGEMENT**

Parastoo Delgoshaei, Master of Science, 2012

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

This thesis describes new models and a system for satisfying requirements, and an architectural framework for linking discipline-specific dependencies through interaction relationships at the ontology (or meta-model) level. In a departure from state-of-the-art traceability mechanisms, we ask the question: What design concept (or family of design concepts) should be applied to satisfy this requirement? Solutions to this question establish links between requirements and design concepts. The implementation of these concepts leads to the design itself. These ideas, and support for design-rule checking are prototyped through a series of progressively complicated applications, culminating in a case study for rail transit systems management.

Last Modified: February 12, 2013

ONTOLOGY-ENABLED TRACEABILITY MODELS FOR ENGINEERING SYSTEMS DESIGN AND MANAGEMENT

by

Parastoo Delgoshaei

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2012

Advisory Committee:
Associate Professor Mark Austin, Chair/Advisor
Professor John Baras
Professor Steven Gabriel

Acknowledgments

Foremost, I would like to thank my parents for their unconditional support and love throughout my entire life. I am very grateful to my older brothers, Parhum and Payam, who always encouraged and supported me – they are my role models. Moreover, I would like to express my sincere gratitude to my advisor, Dr. Mark Austin, for the continuous support during my study and research, for his patience, motivation, enthusiasm and immense knowledge. His guidance and support helped me throughout the research and in the writing of this thesis. I would also like thank Professors John Baras and Steven Gabriel for their insightful comments and serving on my thesis committee. Finally, I would like to thank the National Institute of Standards and Technology (NIST) for their financial support during the past year.

Table of Contents

List of Figures

Chapter 1

Ontology-Enabled Traceability

1.1 Problem Statement

Modern engineering systems that are required to provide a wide range of functionality, operate at high levels of performance, and offer good economy are nearly always team-based efforts. In the design of planes, train and automobiles, for example, a team of project stakeholders will skew the system capabilities to ensure their interests are taken into account. Teams of discipline-specific engineers will conceptualize, design, and build the subsystems that will be integrated into the larger system being developed. Teams of test engineers will make sure the system operates correctly. Teams will operate and maintain the system during its working lifetime. During the design and development of a system, it is the responsibility of the systems engineer to gather and integrate subsystems, and to ensure ensure that every engineer is working from a consistent set of project assumptions. Systems engineers are also responsible for making necessary adjustments to a project when requirements change – for example, the scope of a project might need to be trimmed back to meet new economic realities.

Figure 1.1 illustrates the essential details of the modeling problem faced by systems engineers, and the complementary role it plays to that faced by designers

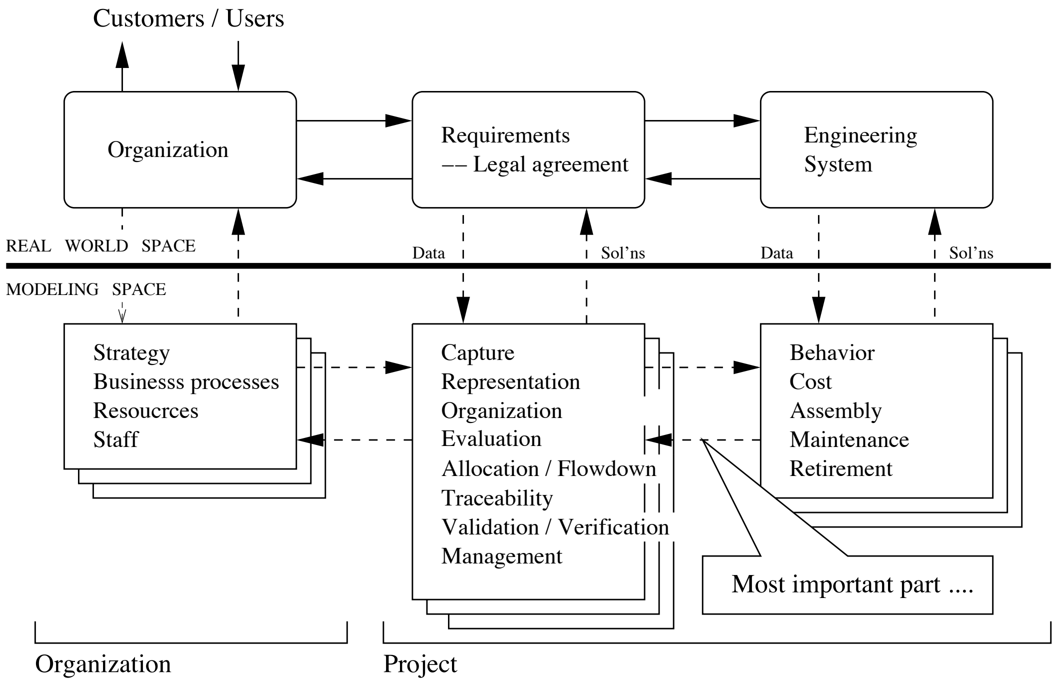


Figure 1.1: A systems engineering view of engineering, requirements, and organizational modeling.

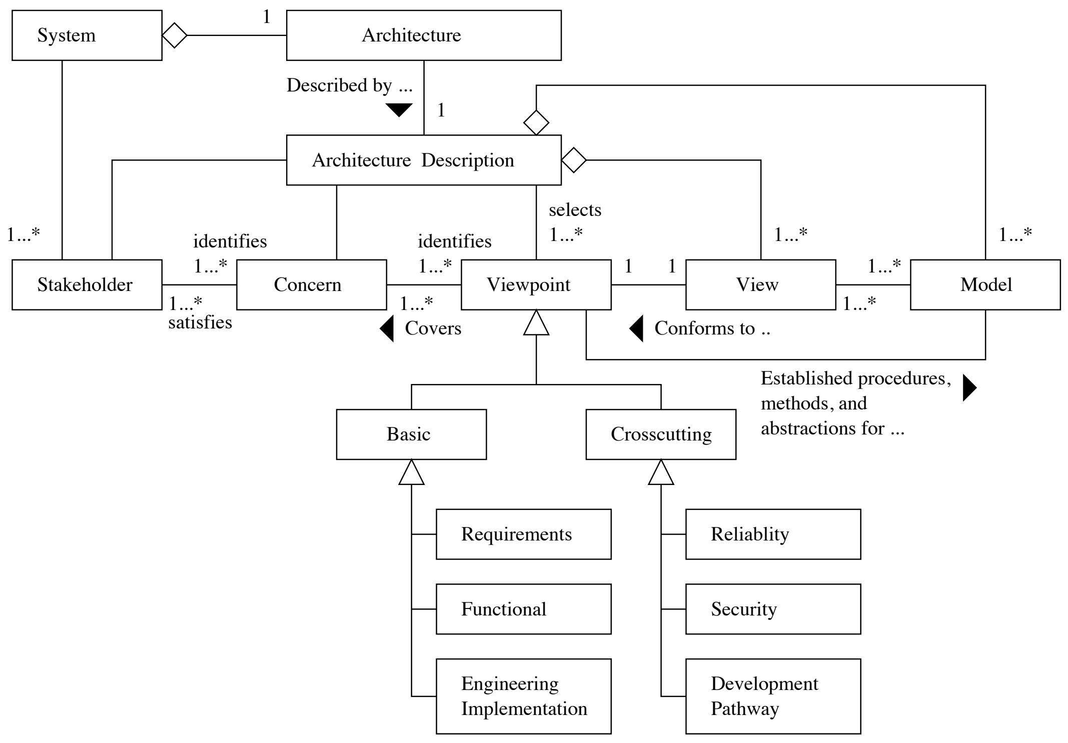


Figure 1.2: Class diagram of concepts contributing to development of architecture descriptions. Assembled from ideas due to Eeles et al. [15], Maier [34], and definitions in the IEEE 1471 Standard [30].

from the traditional engineering domains. Without models to represent the system elements and their connectivity, systems engineers will have difficulty in making quantitative decisions regarding the adequacy of a design, and in choosing rationally among different design alternatives. While a designer from the traditional engineering domains (e.g, from the mechanical or civil domains) is most likely to be concerned with predictions of performance, cost and assembly within their domain, systems engineers need models that make connections between the participating domains, the capture and allocation/flowdown of requirements, and the strategic goals and operations of an organization. These connections are vitally important because without them, there is no rational way of tracking progress, and tracing cause-and-effect relationships between changes in requirements and their impact on a system's measures of effectiveness (cost, performance, functionality), and/or day-to-day system operations and the satisfaction of requirements.

To understand why solutions to the connectivity/traceability problem are hard, consider the class diagram of concepts contributing to development of architecture descriptions shown in Figure 1.2. The heart of the traceability problem lies in a chain of many-to-many relationship involving stakeholders, design concerns, design viewpoints and views, and multiple engineering models, that may be arbitrarily intertwined and complex. The traceability problem is further complicated by the size of problems that need to be handled. As pointed out by Balasubramaniam in a comprehensive study of traceability meta-models, high-end users of traceability work with design problems that have, on average, about 10,000 requirements [4].

Looking forward, it is evident that the importance of traceability connections will only increase. One source of increasing concern stems from the present-day reliance on human-in-the-loop operations steadily giving way to automated control systems, many of them networked together. A second indicator of this trend is the growing class of applications for which long-term managed evolution and/or managed sustainability is the primary development objective. The underlying tenet of our work is that neither of these trends will become fully mature without: (1) An understanding for how and why system entities are connected together, and (2) Formal procedures (methodologies and tools) for assessing the correctness of system operations, estimating system performance, and understanding trade spaces involving competing design criteria.

Glossary of Key Terms

This glossary provides definitions of key terms employed in this work:

- **Action:** (Effect) is the response given to stimuli in a transition, and will normally corresponds to an activity performed during the transition from one state to another state in the statechart.

- **Association:** An association represents a linkage (i.e. a connection line) between two classes in an ontology or a class diagram. An association can have a name and can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties. Bi-directional and uni-directional associations are the most common types of associations.

- **Bi-directional Association:** Refers to a symmetric dependency between two classes. i.e. Class A uses Class B and vice versa.

- **Block Diagram:** A SysML diagram, the represents the principal components of a system and the structural design that connects them together.

- Class Diagram: A UML diagram, which focuses on different classes existing in software systems and their connection with respect to each other.

- **Communication Hub:** A piece of script that assembles model, views, and controllers in MVC architecture.

- **Constraint:** A design constraint refers to some limitation on the conditions under which a system is developed.

- **Controller:** (Mediator) A component of MVC design pattern, that acts as a communication channel between the model and the view.

- **Design Concept:** Refers to an existing entity in the design domain.

- **Directional Association:** Implies directional dependency relationships (e.g., complies with, satisfies, requires, implements). i.e. class A implements class B.

- **Event:** Stimuli that may cause a system to transition from one state to another state in statechart. There are four main categories of events: signal, time, change and call events.

- **Finite State Machine:** A finite-state machine (FSM) is a mathematical model of computation for an abstract machine defined in terms of a finite number of states and transitions, and sequences of input events that will be consumed during the machine operation.

- **Generic Object:** A placeholder representing an entity in the design domain.

- **Guard Condition:** A guard condition is a predicate expression associated with an event. Guard conditions are used to ensure that a transition can only trigger if the evaluation of the guard is true.

- **Individual:** Is a semantic web terminology that represents an instance of a class in the ontology. i.e. data

- **Listener:** (Observer) A class that registers its interest to be notified for changes in other classes (Observable) in observer design pattern.

- **Mediator:** In mediator pattern, a mediator, defines an object that encapsulates how a set of objects should interact.

- **Mediator Pattern:** A behavioral design pattern that is used to manage algorithms, relationships and responsibilities between objects. It mitigates the need for point-to-point connections between objects by defining an object that controls how a set of objects will interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the mediator, rather than with one another.

- **Meta-model:** Is a language that defines how the model should be formed.

- **Model:** A model is an approximation, representation, or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system (IEEE 610.12-1990)

- **Model-Based Systems Engineering:** Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases (INCOSE-TP-2004-004-02, Version 2.03, September 2007).

- **Model-View-Controller (MVC):** Is a system design pattern that separates the representation of information from the user's interaction with it.

- **Observer Pattern:** The observer pattern is applicable to problems where a message sender (observable) needs to broadcast a message to one or more receivers (or observers), but is not interested in a response or feedback from the observers.

- **Ontology:** A model that describes what entities exist in a design domain, and how such entities are related.

- **Ontology Class:** A placeholder for an entity in the system design. An ontology class may have some dataType or objectType properties.

- **DataType Property:** DataType Property defines the relation between instances of classes and literal values, i.e., String using the Protg tool.

- **ObjectType Property:** ObjectType Property defines the relation between instances (individuals) of two classes in semantic web terminology using protg tool.

- **Ontology-Enabled Traceability:** A mechanism that maps a requirement to a design concept in the ontology and from there to the engineering objects.

- **Ontology Web Language:** The Web Ontology Language (OWL) is a knowledge representation languages for defining ontologies.

- **Reasoner (Rule Engine):** A semantic reasoner, reasoning engine, rules engine, or simply a reasoner, is a piece of software able to infer logical consequences from a set of asserted facts or axioms.

- **Reasoning:** To infer new statements based on set of asserted facts in the ontology.

- **Requirement:** A textual representation derived from stakeholders statement of need, or system functionality limitations.

- **Rule Checking:** A mechanism that ensures existing data in the ontology is consistent with rules defined over an ontology. A rule engine often performs this task.

- **Semantic Web:** (Web of liked data) The ultimate goal of the semantic web is to develop systems that can support trusted interactions over the network.

- **Semantic Web Layer Cake:** An informal term used to describe the stack of technologies used in the implementation of the Semantic Web.
- **Semantic Web Technologies:** Semantic Web technologies provide features to build vocabularies, and develop rule repositories and ontologies.
- **Software Design Patterns:** In software engineering a design pattern is a general reusable solution description to a recurring problem. It, also provides the core solution to the problem.
- **State:** A state is a description of the status of a system that is waiting to execute a transition.
- **Statechart:** In model-based systems engineering, state machines (statecharts) are used to describe the state-dependent behavior of a system component throughout its life cycle.
- **Semantic Web Rule Language:** The Semantic Web Rule Language (SWRL) is a rule language that is compatible with OWL ontologies. It is not standardized yet.
- **SysML:** The Systems Modeling Language (SysML) is a graphical modeling language used to define models of systems structure and system behavior.
- **Traceability Mechanism:** A approach which represents traceability between different phases of system development or component in the system.

- **Transition:** A transition is a set of actions to be executed when a condition is fulfilled or when an event is received.
- **Transition Notation:** A notation that is used in statecharts to show how a transition will proceed.e.g. Event(Parameters), guard evaluation/ action which implies that if an event is received, and the guard expression evaluates to true, and action will be performed while the transition is proceeding.
- **Unified Modeling Language:** UML is a graphical modeling language used to define mainly software systems structure and behavior.
- **View:** Visual representation of the model in MVC design architecture.
- **Workspaces:** Requirement, ontology and engineering workspaces are created to store various viewpoints and their associated models.

1.2 From Operations Concepts to Requirements and System Design

Let us assume that the required engineering system does not exist. Figure 1.3 illustrates the development pathway for one level of abstraction, beginning with the formulation of an operations concept, requirements, fragments of behavior, and tentative models of system structure. Requirements need to be organized according to role they will play in the design (e.g., behavior, structure, test) and processed to insure consistency, completeness, and compatibility with the requirements system. System performance can be evaluated with respect to the value of performance attributes. Models of structure specify how the system will accomplish its purpose.

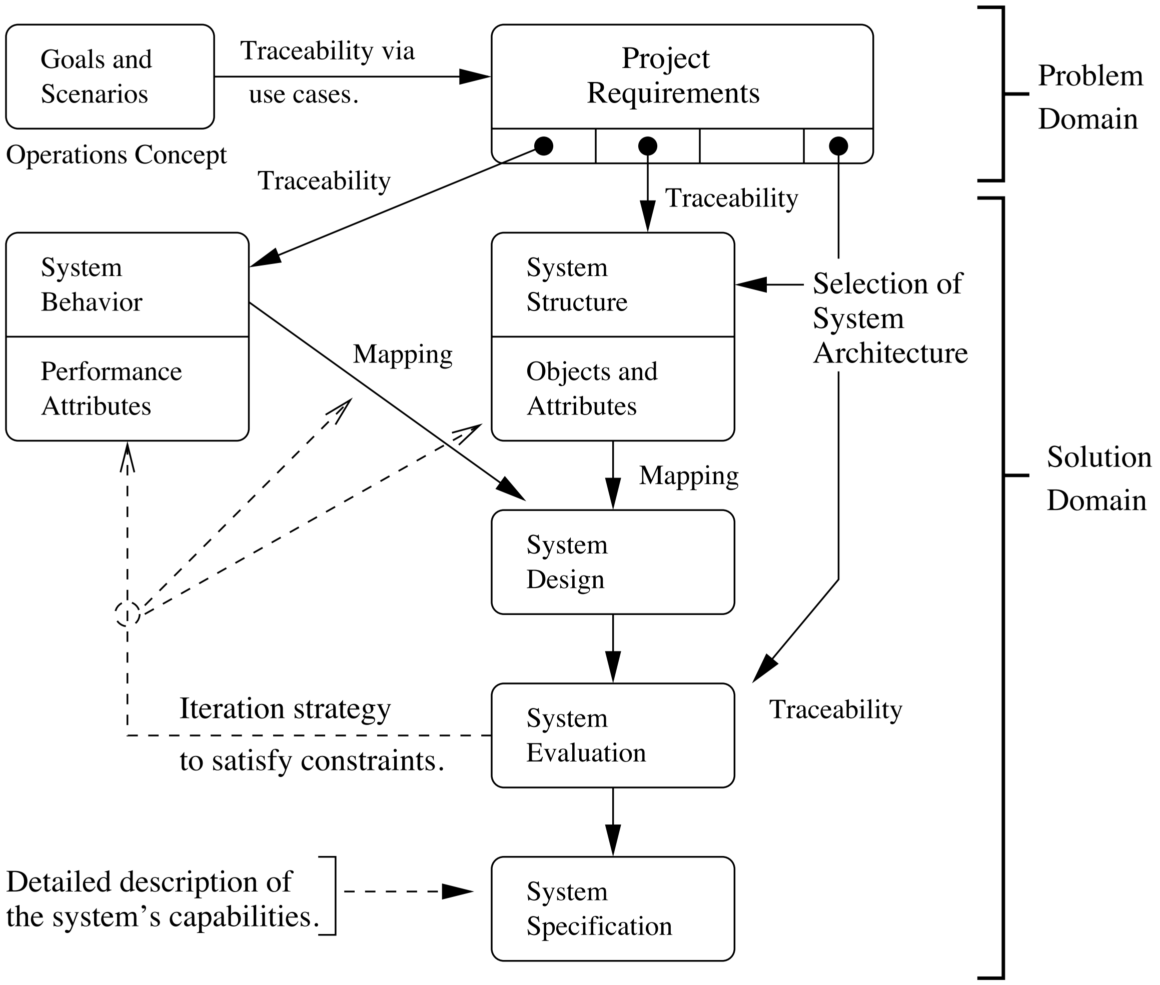


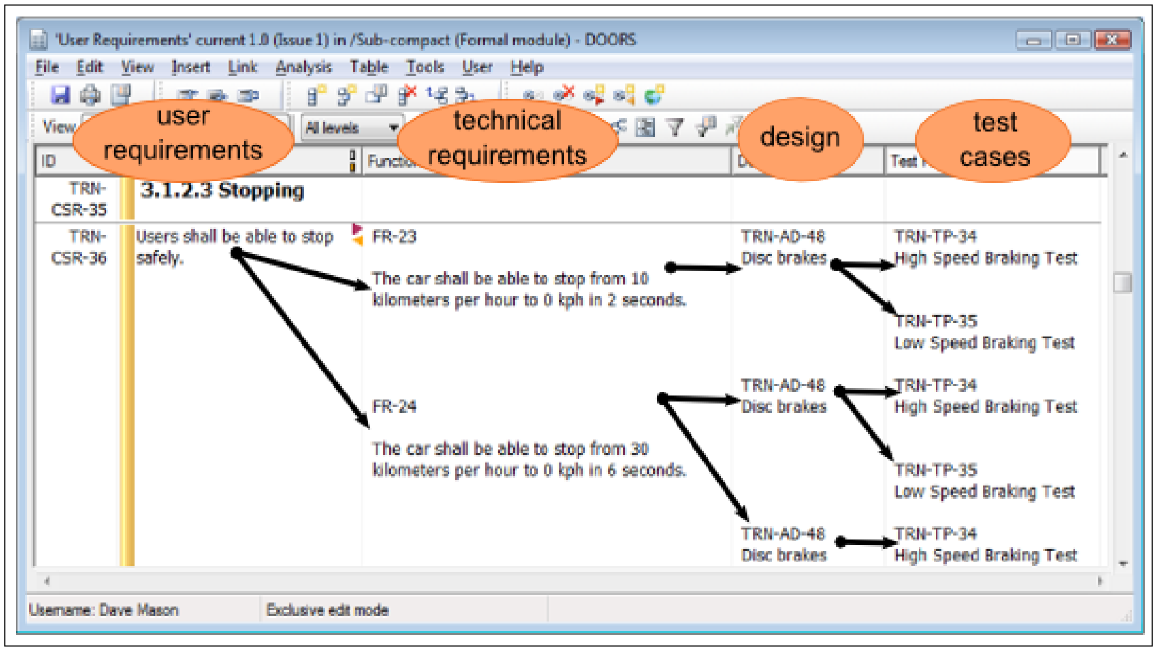
Figure 1.3: System structure and system behavior viewpoints, and traceability mappings for the development pathway goals/scenarios through system evaluation.

System architecture will be evaluated with respect to selected objects, and the value of their attributes e.g., attributes of the physical structure of the design; attributes of the environmental elements that will interact the the system; attributes of the system inputs and system outputs). System designs are created by assigning (or mapping) fragments of required to object and subsystems in the system structure. Thus, the behavior-to-structure mapping defines (in a symbolic way) the functional responsibility of each subsystem/component and establishes constraint satisfaction problems to address several issues: (1) Can all the components be connected in a way that enables the required functionality? (2) What kinds of compatibility of object and component interfaces will need to be satisfied? (3) Can the components be packed into a sufficiently small space?, and (4) Do the objects and components have a sufficient set of attributes against which performance metrics can be assessed? During the system evaluation, functional and performance characteristics are evaluated against the test requirements. Several iterations of development (involving modifications to the operations concepts, system behavior, system structure) will usually be required before all of the system requirements are satisfied.

1.3 State-of-the-Art Requirements Traceability

Present-day requirements management tools such as SLATE [29], CORE [9], and DOORS [14] provide the best support for top-down development where the focus is on requirements representation, traceability, and allocation of requirements to system abstractions. In most of today's requirements management tools, individual requirements are represented as textual descriptions with no underlying semantics. System engineers like to organize groups of requirements (e.g., functional requirements, interface requirements) and abstractions for system development into tree-like hierarchies, in part, because this technique is comfortable and well known. See, for example, the traceability report shown in Figure 1.4. This is state-of-the-art practice. However, when requirements are organized into layers for team development, graph structures are needed to describe the comply and define relationships, sometimes tracing across the same level. This happens because requirements are tightly interdependent with each other across the same level of abstraction. Because the underlying graphical formalism is weak, many questions that a user might want to ask about requirements and/or the system structure remain unanswered or omitted. Simple questions like "Show me all complying and defining requirements that are related to this particular requirement" cannot be answered.

As a case in point, the IBM Teamcenter (SLATE) Requirements Tool aims to improve systems engineering productivity (e.g., better accuracy; accelerated functional design; support for trade studies), particularly at the conceptual stages of development. SLATE is based upon very good data representations for requirements and abstraction blocks (ABs), linked together into a graph structure. The graph edges correspond to relationships between entities in the system development – for example, traceability links (complying and defining links) and connectivity of different abstraction block hierarchies with translational mappings. ABs provide modeling support for attributes (whose values can be used in performance assess-


'User Requirements' current 1.0 (Issue 1) in /Sub-compact (Formal module) - DOORS
File Edit View Insert Link Analysis Table Tools User Help
View
All levels
user requirements
technical requirements
design
test cases
ID
Functi
D
Test
Username: Dave Mason
Exclusive edit mode

TRN-CSR-35 3.1.2.3 Stopping

TRN-CSR-36 Users shall be able to stop safely.

FR-23
The car shall be able to stop from 10 kilometers per hour to 0 kph in 2 seconds.

FR-24
The car shall be able to stop from 30 kilometers per hour to 0 kph in 6 seconds.

TRN-AD-48
Disc brakes

TRN-TP-34
High Speed Braking Test

TRN-TP-35
Low Speed Braking Test

Figure 1.4: Schematic of a traceability report as modeled in DOORS.

ment), functional flows (i.e., data inputs and outputs), links to other ABs (e.g., in parent/child relationships), connectivity to groups and translational mappings, and budgets. See Figure 1.5. Justification for use of abstraction blocks in lieu of more detailed representations of an engineering system is really very simple – at the conceptual stages of development, most of these details (e.g., geometry) remain to be developed. Translational mapping relationships (TRAMs) provide a method for connecting abstraction blocks across hierarchies and for evaluating design alternatives. The upper part of Figure 1.6 shows, for example, trace links connecting requirements to abstraction blocks in a functional decomposition hierarchy. Then, TRAMs relay the existence of dependencies between ABs in the electrical, mechanical and software viewpoints.

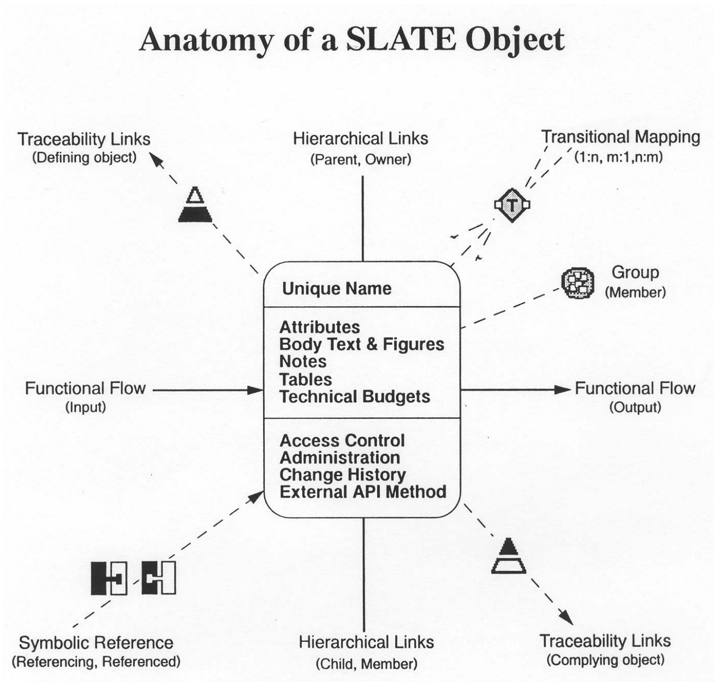


Figure 1.5: Anatomy of a generic object in IBM Telelogic SLATE [29].

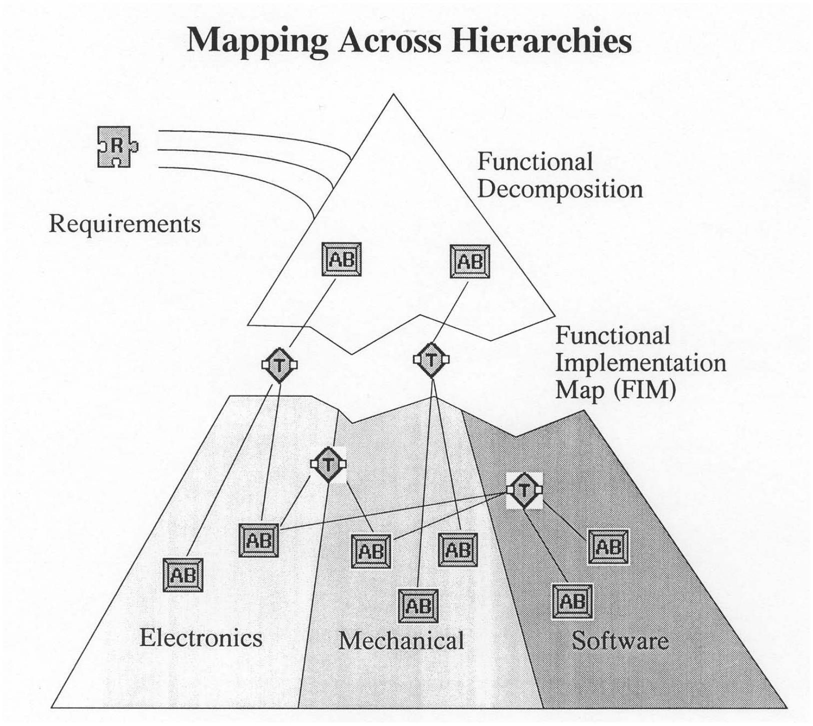


Figure 1.6: Modeling of translational mappings (TRAMs) across hierarchies in IBM Telelogic SLATE [29].

1.4 Ontology-Enabled Traceability Models and Mechanisms

In a step toward mitigating the weaknesses in state-of-the-art capability in requirements modeling and traceability, in this section we describe a new approach to requirements traceability.

1.4.1 Basic Ontology-Enabled Traceability Model

In state-of-the-art traceability mechanisms design requirements are connected directly to design solutions (i.e., engineering objects). An alternative, and potentially better, approach is to satisfy a requirement by asking the basic question: What design concept (or group of design concepts) should I apply to satisfy a requirement? Design solutions are the instantiation/implementation of these concepts.

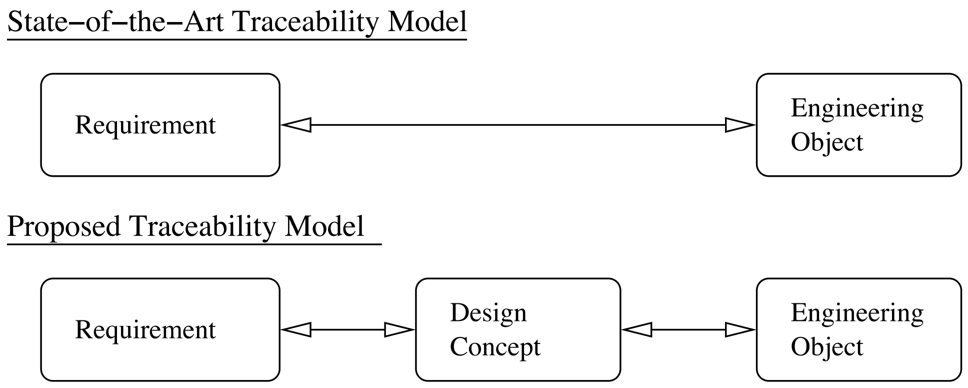


Figure 1.7: Simplified view of ontology-enabled traceability.

Figure 1.7 provides a visual comparison of these two approaches to system traceability. In the basic model the requirements, ontology, and engineering models provide distinct views of a design: (1) Requirements are a statement of "what is required." (2) Engineering models are a statement of "how the required functionality and performance might be achieved," and (3) Ontologies are a statement of "concepts justifying a tentative design solution. In a standard system development the requirements and engineering models will be defined in an iterative manner, with a focus on covering the breadth of a problem before drilling down to work out the design details.

Remark on Ontologies. An ontology is a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic or domain [21, 26, 42]. Two practical examples of ontoloiges for insurance and university applications can be found at the end of Chapter 2. To provide for a formal conceptualization within a particular domain, and for people and computers to share, exchange, and translate information within a domain of discourse, an ontology needs to accomplish three things [32]: (1) Provide a semantic representation of each entity and its relationships to other entities, (2) Provide constraints and rules that permit reasoning within the ontology, and (3) Describe behavior associated with stated or inferred facts. Items 1 and 2 cover the concepts and relations that are essential to describing a problem domain. Items 2 and 3 cover the axioms that are often associated with an ontology. Usually, axioms will be encoded in some form of first-order logic.

Benefits. From a design standpoint, the proposed method offers the following benefits:

1. The use of ontologies within traceability relationships helps engineers deal with issues of system complexity by raising the level of abstraction within which systems may be represented and reasoned with. In fact, by explicitly connecting requirements to engineering system representations through ontologies we are indicating "how and why" requirements satisfaction is taking place.

2. Because ontologies represent concepts for a problem domain, the ontologies are inherently reusable across families of similar projects.

3. A key advantage of the proposed model is that software for "design rule checking" can be embedded inside the design concepts module. Thus, rather than waiting until the design has been fully specified, this model has the potential for detecting rule violations at the earliest possible moment. This is where design errors are easiest and cheapest to fix.

1.4.2 Support for Multiple Viewpoint Design

Figure 1.7 is overly simplified in the sense that it implies one requirement will be satisfied through the activation of one design concept and one design entity. As already illustrated in Figure 1.2, real-world systems will have multiple stakeholders, many requirements, and multiple design criteria that will sometimes be in conflict. To accommodate these relationships in a disciplined way, there needs to be a formal framework for: (1) Connecting stakeholder concerns to engineering entities, (2) Capturing the interactions and restrictions among the various viewpoints, and (3) Systematically abstracting away details of a problem specification that are unrelated to a particular decision.

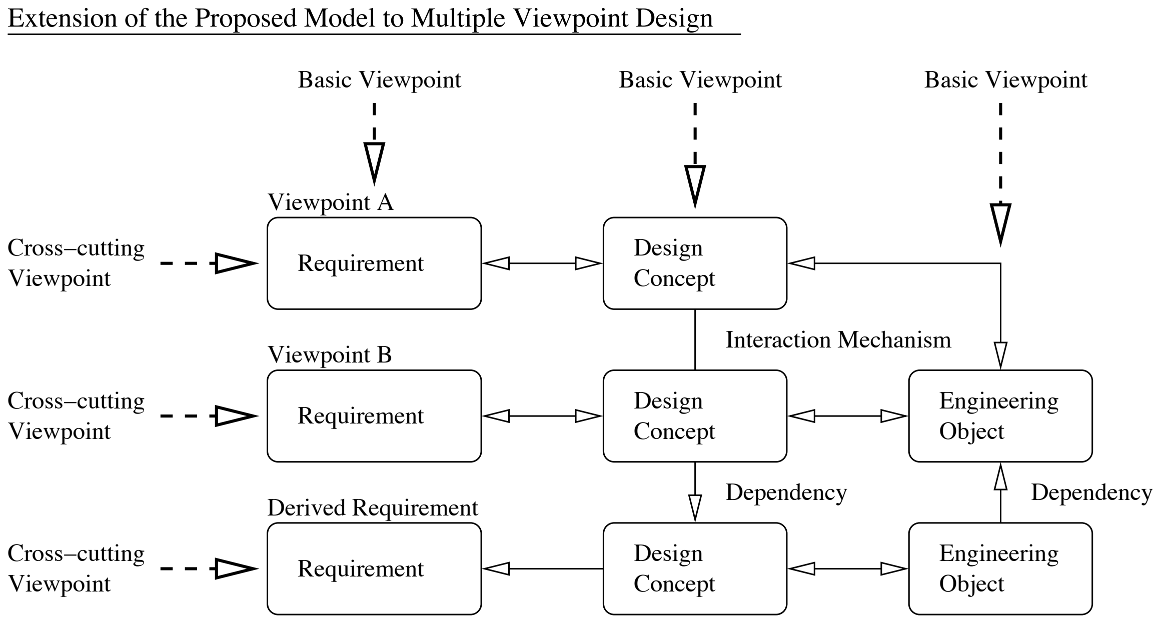


Figure 1.8: Extension of the proposed model to multiple-viewpoint design.

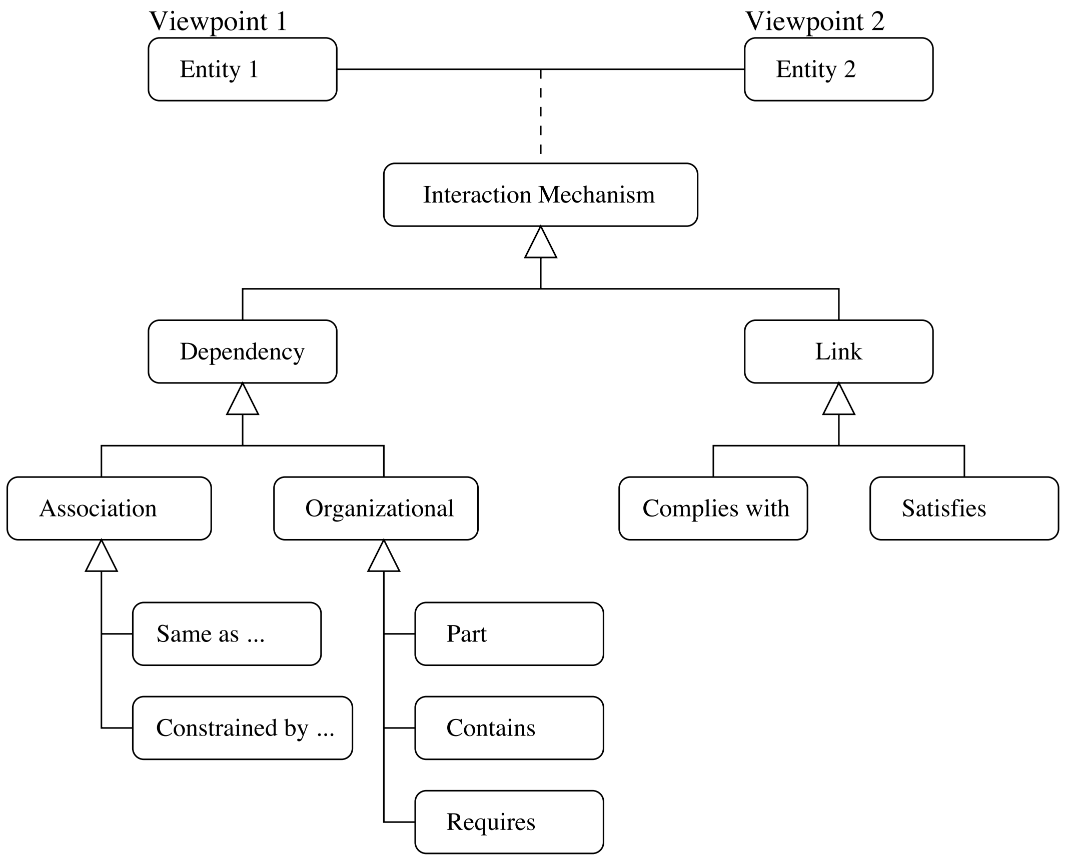


Figure 1.9: Class hierarchy for dependency relationships among design entities.

Figures 1.8 and 1.9 show extensions of the basic model to support the additional complexities associated with team-based design of systems with multiple stakeholders and multiple viewpoints. The basic model is extended from a chain to a graph of design concept entities. Most of the graph edges will involve bi-directional association relationships (e.g., same as and constrained by). However, directional dependency relationships (e.g., complies with, satisfies, requires) will also occur. In the design of building, for example, stakeholders will include the architects, structural engineers, and hvac engineers, among others. Architects are concerned with the design and layout of spaces. Structural engineers are concerned with the design of a system that can safely transfer forces to the building foundation. HVAC engineers are concerned with the air quality and comfort of the occupants. Each discipline will require resources (e.g., cost) and may impose constraints on other disciplines.

The project stakeholders and system developers will look at subsets of the graph. For example, a requirements engineer is concerned about the gathering, representation, and organization of requirements across all viewpoints. This is a basic viewpoint. Similar relationships exist for the ontology engineer and the engineering of the system itself. The project stakeholders will have viewpoints that cut across the different stages of project development, from requirements to selection of design concepts (ontology) and their implementation in the engineering system. Ontologies for different design viewpoints (e.g., system structure, system behavior) may also be linked, thereby establishing dependencies among the viewpoints of different engineering disciplines and their concerns.

Step-by-Step Procedure for System Development. The graph of requirements to ontologies to engineering model relationships can be mapped onto the step-by-step procedure for system development. Figure 1.10 shows the extension of Figure 1.3 to include ontologies for different design viewpoints (e.g., system structure, system behavior) their linking to create dependency relationships, and the use of rule checking.

1.4.3 Support for Systems Management

Ontology-enabled traceability mechanisms were originally proposed with an eye to improving the way engineers design and create things. However, it is evident that once a system has been designed and built, the traceability mechanisms can switch purposes and support real-time performance assessment, which in turn, provides data for decision making in systems management.

Figure 1.11, a slight extension of Figure 1.7, shows the role that sensing and design rule checking will play in ontology-enabled traceability support for systems management. In design, requirements are satisfied through the selection of design concepts, which, in turn, are implemented as detailed (engineering) model and further downstream, the as-built system itself. This is a left-to-right flow of activities.

Systems management will correspond to a right-to-left flow. Sensors embedded in the real-world physical system will collect and transmit streams of data

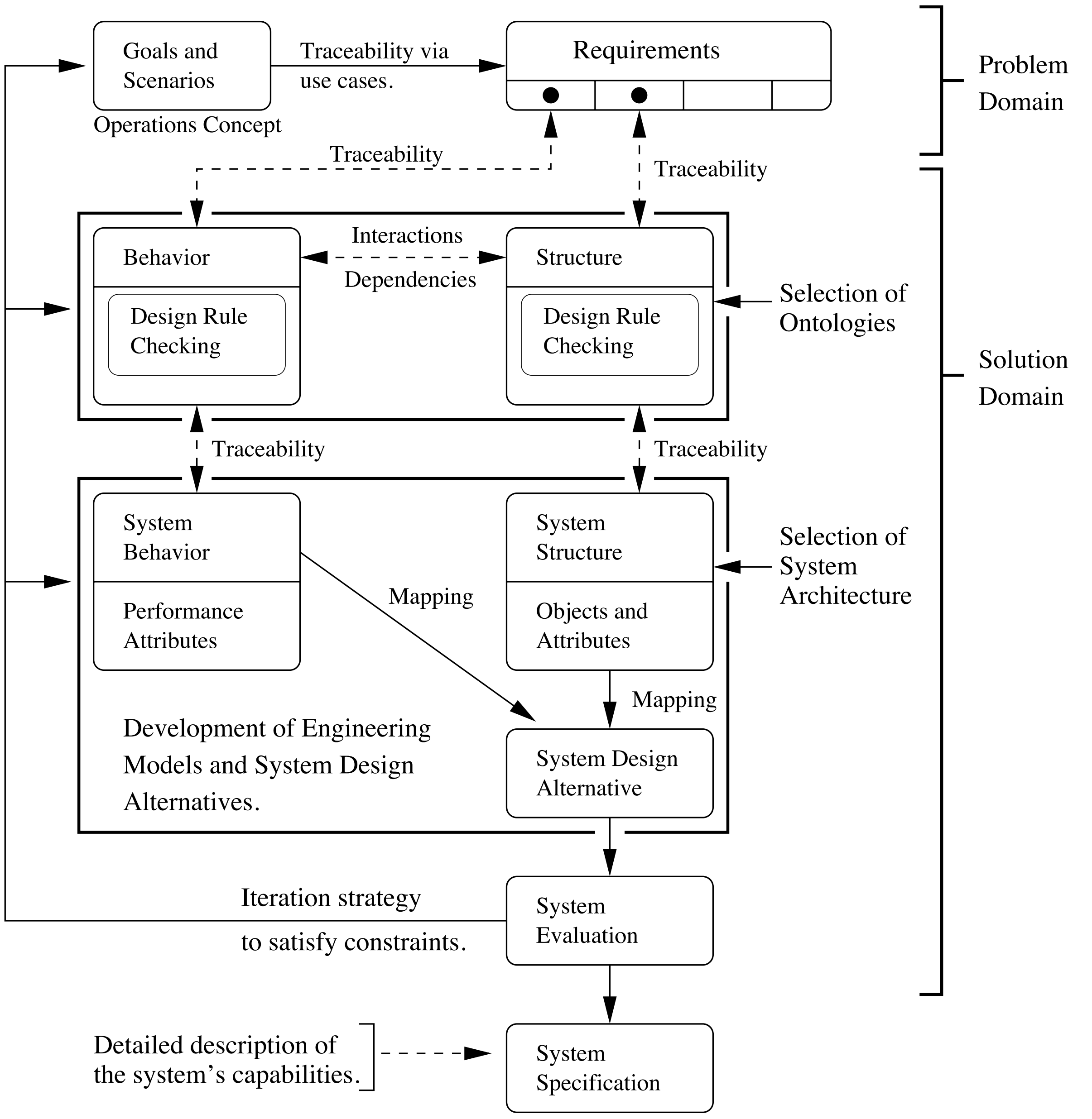


Figure 1.10: Flowchart for system development and traceability.

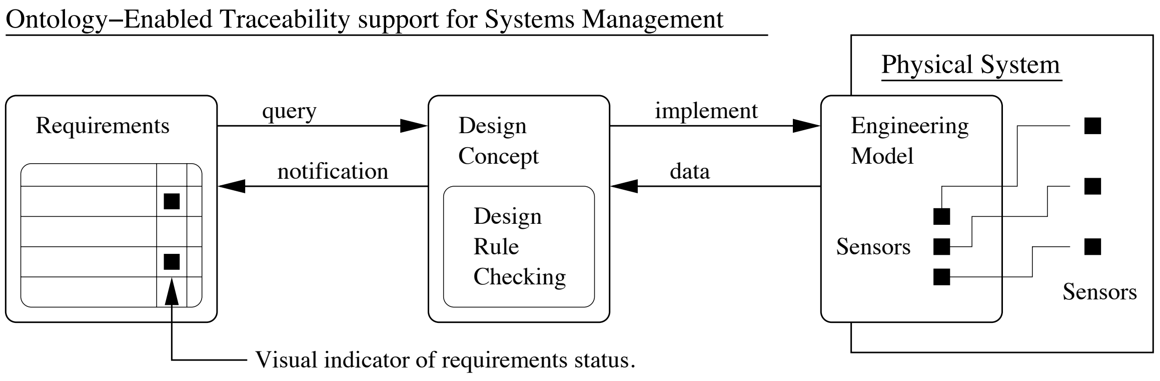
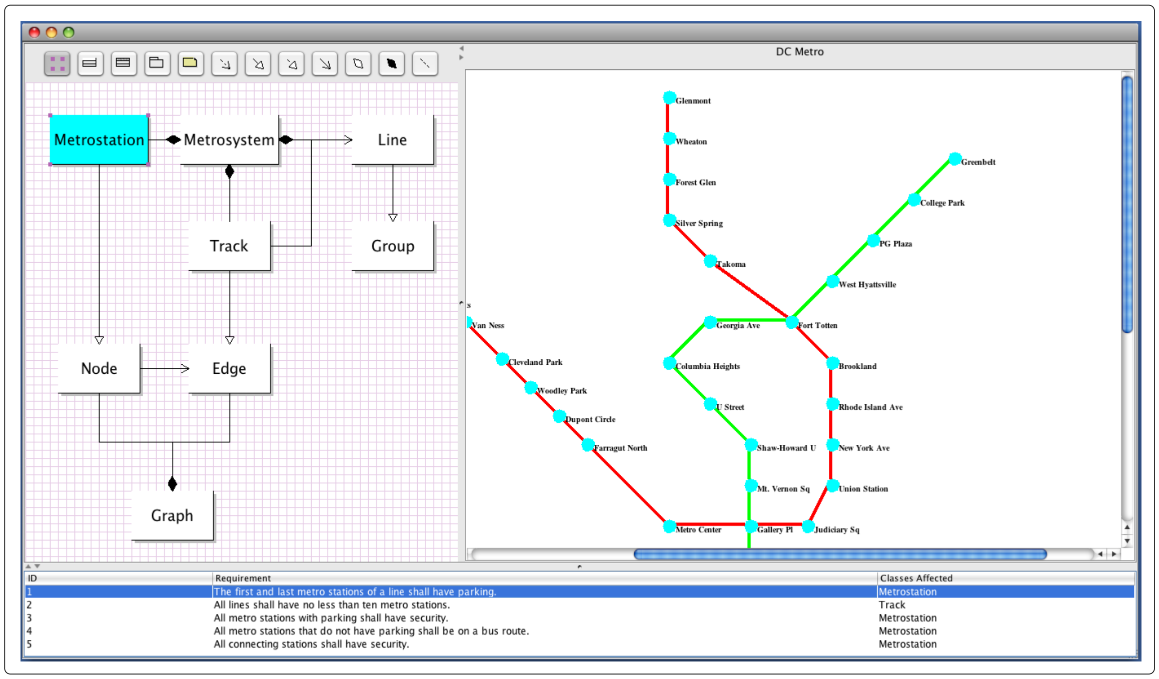


Figure 1.11: Application of ontology-enabled traceability to systems management.

to models of sensors in the engineering model. Then, the data will be forwarded to procedures for design rule checking (design compliance) attached to the ontologies. When a changes in rule checking status occurs (e.g., because something in the real-world system breaks), notifications will be sent to the requirement node. Simple visual indicators can be used to visualize changes in requirements status.

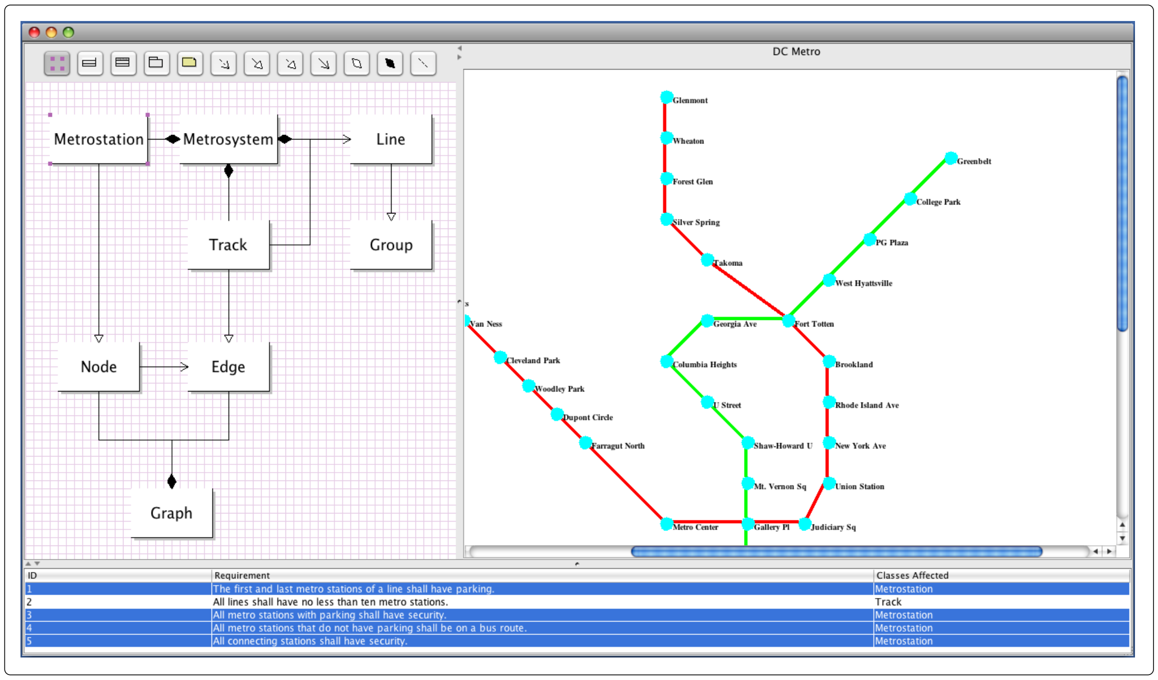
1.5 First-Generation Software Implementation

The basic model for ontology-enabled traceability was formulated by Austin and Bever in 2005-2006 [3, 8], extended to multiple-viewpoint design by Austin in 2010 [2], and prototyped with software that modeled the interaction of requirements, ontologies, and models associated with architectural design of the Washington DC Metro System. Figures 1.12 through 1.15 are screendumps of the requirements, ontology classes, and a plan view of the Washington DC Metro System. The requirements model contains only five requirements and is displayed in a table format. UML class diagrams are used to display ontology concepts because this is the most natural place to start. The top right-hand window is a plan view of the Washington



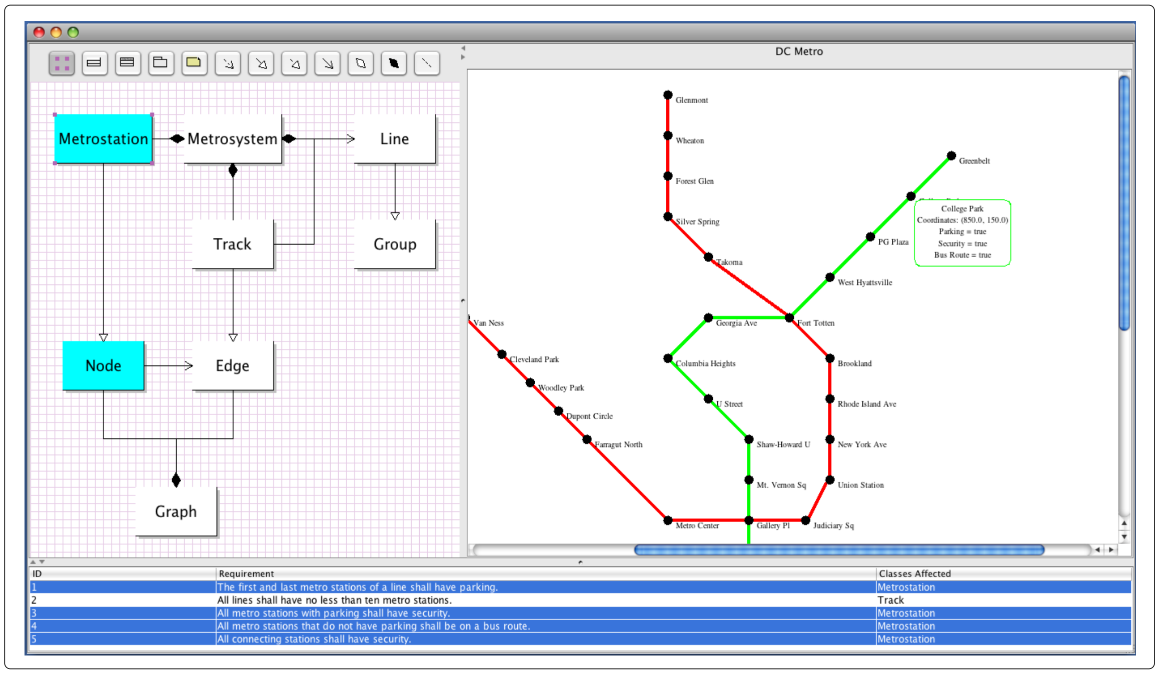
ID	Requirement	Classes Affected
1	The first and last metro stations of a line shall have parking.	Metrostation
2	All lines shall have no less than ten metro stations.	Track
3	All metro stations with parking shall have security.	Metrostation
4	All metro stations that do not have parking shall be on a bus route.	Metrostation
5	All connecting stations shall have security.	Metrostation

Figure 1.12: Tracing a requirement to the UML class diagram and onto the engineering model. Source: Austin and Wojcik [2].



ID	Requirement	Classes Affected
1	The first and last metro stations of a line shall have parking.	Metrostation
2	All lines shall have no less than ten metro stations.	Track
3	All metro stations with parking shall have security.	Metrostation
4	All metro stations that do not have parking shall be on a bus route.	Metrostation
5	All connecting stations shall have security.	Metrostation

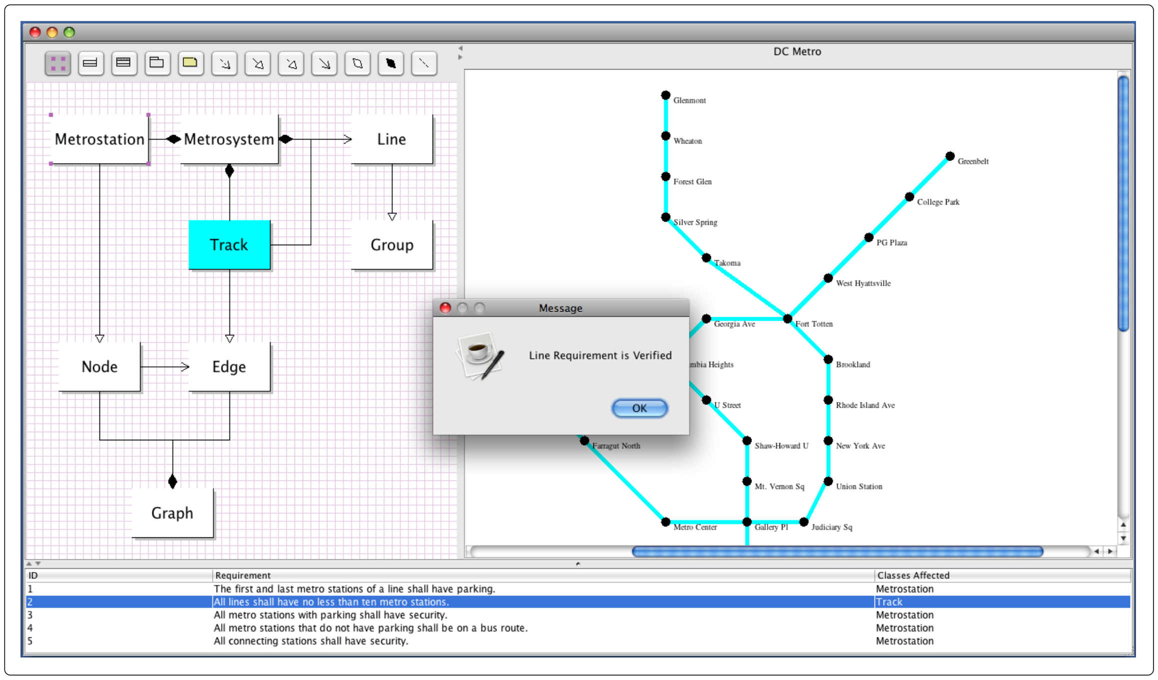
Figure 1.13: Graphical display of requirements and engineering model objects associated with the MetroStation class. Source: Austin and Wojcik [2].



College Park
Coordinates: (850.0, 150.0)
Parking = true
Security = true
Bus Route = true

ID	Requirement	Classes Affected
1	The first and last metro stations of a line shall have parking.	Metrostation
2	All lines shall have no less than ten metro stations.	Track
3	All metro stations with parking shall have security.	Metrostation
4	All metro stations that do not have parking shall be on a bus route.	Metrostation
5	All connecting stations shall have security.	Metrostation

Figure 1.14: Graphical display of requirements and ontology classes associated with the College Park Metro Station. Source: Austin and Wojcik [2].



Line Requirement is Verified

ID	Requirement	Classes Affected
1	The first and last metro stations of a line shall have parking.	Metrostation
2	All lines shall have no less than ten metro stations.	Track
3	All metro stations with parking shall have security.	Metrostation
4	All metro stations that do not have parking shall be on a bus route.	Metrostation
5	All connecting stations shall have security.	Metrostation

Figure 1.15: Implementation of rule checking for the track requirement. Source: Austin and Wojcik [2].

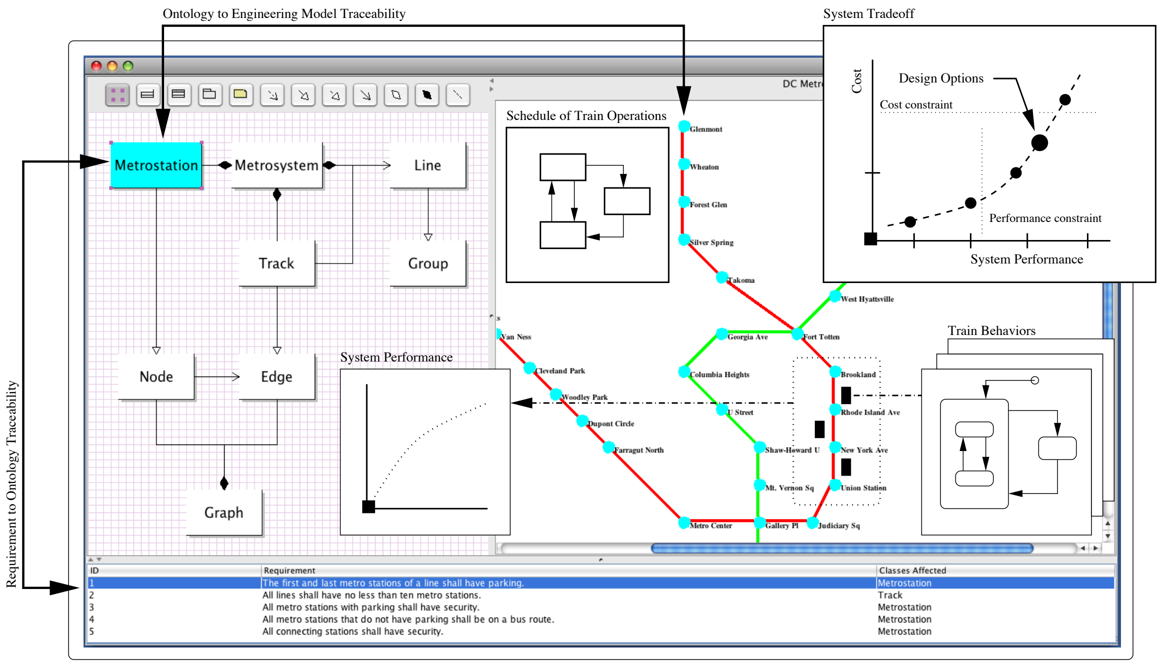
DC Metro System.

Designers are provided with the tools to freely interact with the symbols in each viewpoint, and for changes in status to be synchronized across viewpoints. Such a framework transforms the models and views into spreadsheet-like support for engineering design and systems management. As a case in point, Figure 1.14 illustrates the screen interface when a designer's mouse is positioned over the College Park Metro Station. A pop-up bubble displays attributes of the College Park Metro Station (e.g., whether or not there is parking). Under the hood, the requirements, ontology, and engineering models synchronize their states through the use of traceability mechanisms implemented as graphs of listener mechanisms. Thus, when a user interacts with an object in the engineering view, messages for event interaction are propagated to the ontology and requirements views. So we see that conceptually the College Park Metro Station is both a Metro Station in a Transportation network, and Node in a Graph. Four of the five requirements shown along the bottom of Figure 1.14 employ the concept of Metrostation in their satisfaction.

1.6 Contributions of this Thesis

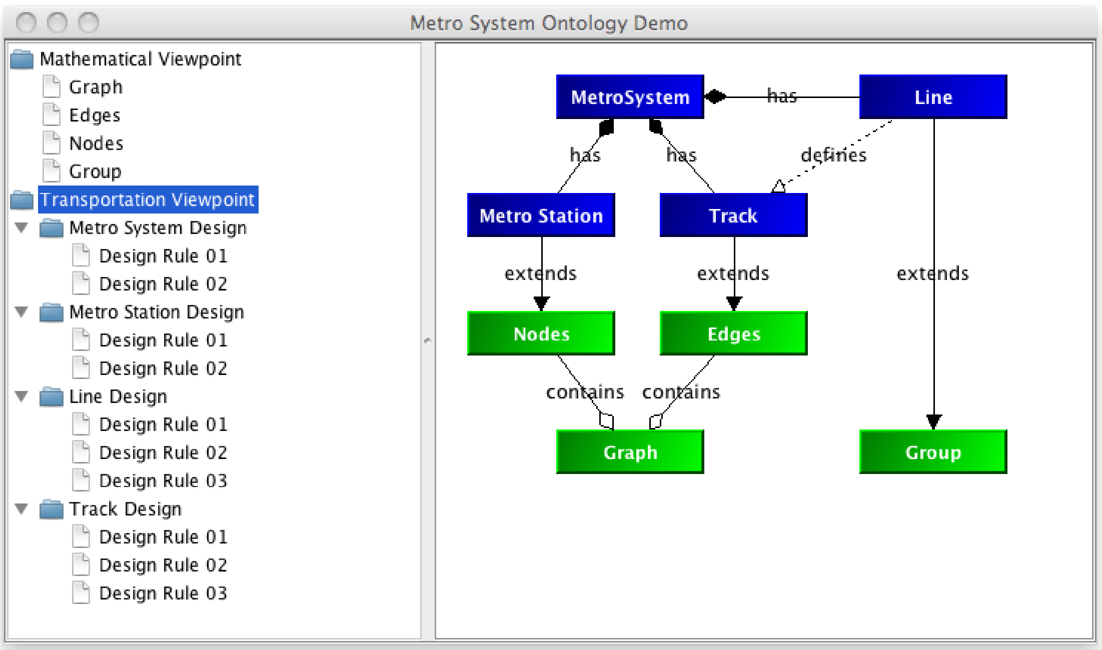
Figures 1.16 and 1.17 illustrate the natural extensions of Figures 1.12 through 1.15, and the motivating force for the four contributions of this thesis:

1. **Software Design Patterns.** This work involves the design and implementation of software design patterns (e.g., model-view-controller, mediator, and observer) to support the management and visualization of traceability rela-



ID	Requirement	Classes Affected
1	The first and last metro stations of a line shall have parking.	Metrostation
2	All lines shall have no less than ten metro stations.	Track
3	All metro stations with parking shall have security.	Metrostation
4	All metro stations that do not have parking shall be on a bus route.	Metrostation
5	All connecting stations shall have security.	Metrostation

Figure 1.16: Annotated prototype for requirements-ontology-engineering traceability in the Washington DC Metro System.


Metro System Ontology Demo

- Mathematical Viewpoint
  - Graph
  - Edges
  - Nodes
  - Group
- Transportation Viewpoint
  - Metro System Design
    - Design Rule 01
    - Design Rule 02
  - Metro Station Design
    - Design Rule 01
    - Design Rule 02
  - Line Design
    - Design Rule 01
    - Design Rule 02
    - Design Rule 03
  - Track Design
    - Design Rule 01
    - Design Rule 02
    - Design Rule 03

Figure 1.17: Graph and tree views of a metro system ontology. Transportation and mathematical concerns are shown in blue and green respectively.

tionships.

2. **Web Ontology Language (OWL).** This work investigates the use of OWL to capture and represent design concepts used in the various applications.

3. **Semantic Web Ruling Language (SWRL).** This work investigates the use of SWRL for rule checking of systems operations which may change over time.

4. **Graph Visualization Techniques.** This work resulted in a visualization package for executable modeling of statechart behaviors.

In Figure 1.16, an end-of-line parking requirement is associated with the Metro-station ontology which, in turn, traces to all of the metro station instances in the Washington DC Metro System model. Figure 1.17 highlights the importance of presenting multiple visual perspectives (e.g., tree and graph views) to designers and visualizing dependencies among system perspectives.

The scope of work conducted by Wojcik and Austin [2] was restricted to design of a metro system architecture (i.e., stations and tracks). The first-cut implementation has no time, no behavior, and no trains! In this project, simplified timetable-driven train behaviors will be added to the system model. Time-dependent behavior of the train scheduler and individual trains will be modeled as a network of communicating statechart behaviors. Traceability mechanisms will link the requirements to the ontologies to the engineering models. At the component level, such an extension will require traceability connections between functional and performance requirements and individual states, the value of attributes within states of behavior models, and the value of guard conditions that will enable transitions between states. The existence of a direct pathway from requirements to performance attributes and their evaluation will allow for trade-space studies of constraint settings versus performance. At the system level, traceability mechanisms will be evaluated in terms of relationships among and aggregations of lower-level entities. For example, system-level requirements for safety might trace to a value representing the spacing of trains. Overall metro system performance could be evaluated in terms of passenger capacity of the operating trains.

While it is relatively straightforward to prototype a small-scale system that implements basic model of ontology-enabled traceability, the problem with "quick and dirty implementations is that they nearly always end up being an intertwined mess of software code that is neither scalable, nor reusable, nor readily extensible. The ad-hoc implementation of larger-scale systems may not even be tractable [11, 12, 13]. As such, the only way forward is to step back and approach the software architectural design problem from first principles, namely: use of software components and their interfaces, mechanisms for communication between components, and use of software design patterns to structure the overall software architecture.

The scope of work for this project is as follows: Chapter 2 describes the role that Semantic Web formalisms can play in the implementation of ontologies, and formal approaches to reasoning and rule checking. The application of software design patterns to the system architecture design and implementation of workspaces is covered in Chapter 3. The hope is that software patterns will facilitate the implementation of multiple models of visualization. Chapter 4 discusses a software implementation of statechart behavior and its application to standalone statechart behavior and implementation of requirements-ontology-engineering traceability in a simple lamp. A rail transit systems management case study – an ontology will define the design domain and a sets of rules will describe to system constraints – is described in Chapter 5. Chapter 6 covers the project conclusions and future work.

Chapter 2

Systems Engineering and the Semantic Web

2.1 Problem Statement

The Semantic Web is important to the Systems Engineering community because it provides formalisms (i.e., models and tools) for sharing and reasoning with data on the Web. As companies move toward the team-based development of projects and products, having Web access to design specifications and models adds value to business operations. An early example of this trend is the STEP AP233 standard [35, 37], which allows for systems engineering data exchange among tool vendors. This project is concerned with opportunities for using Semantic Web technologies to create ontological descriptions of design domains, supported by rule-based reasoning about the adequacy of a design and its operations.

Figure 2.1 shows the essential details of a three-part framework – textual requirements, ontology model and engineering model – for the implementation of ontology-enabled traceability mechanisms and rule-based design assessment. Textual requirements are connected to the ontology model, and logical and mathematical design rules, and from there to the engineering model. Ontology models encompasses the design concepts (ontology classes) which have properties (c.f., attributes in classes) to represent the consequence of constraint and design rule evaluations.

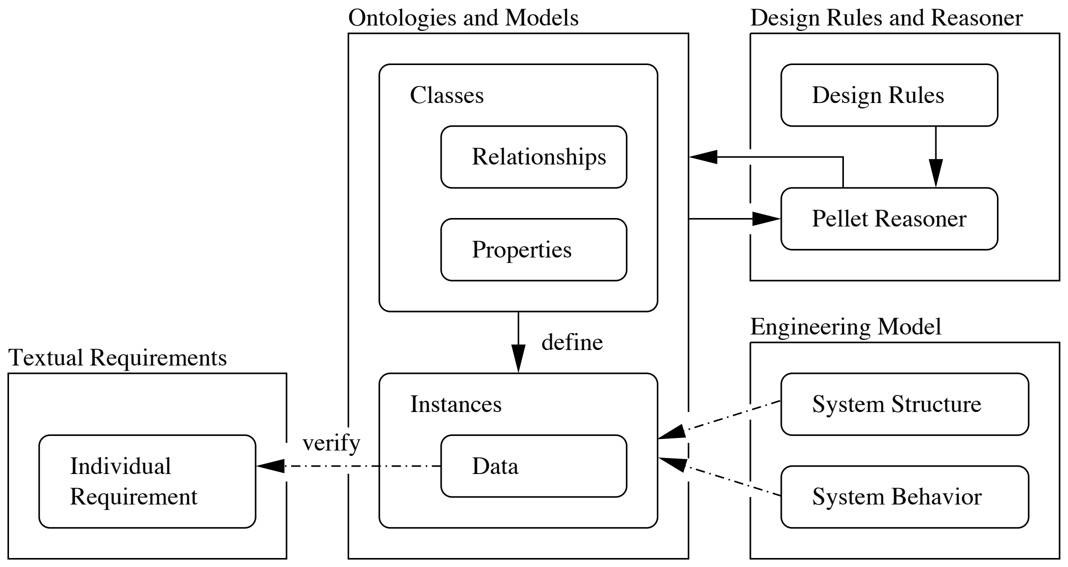


Figure 2.1: Framework for implementation of ontology-enabled traceability and design assessment.

The advantages in this approach to problem solving are as follows [33, 41]: (1) Rules that represent policies are easily communicated and understood, (2) Rules retain a higher level of independence than logic embedded in systems, (3) Rules separate knowledge from its implementation logic, and (4) Rules can be changed without changing source code or underlying model. A rule-based approach to problem solving is particularly beneficial when the application logic is dynamic (i.e., where a change in a policy needs to be immediately reflected throughout the application) and rules are imposed on the system by external entities. Both of these conditions apply to the design and management of engineering systems.

In a departure from state-of-the-art systems engineering practice, where Unified Modeling Language (UML) and System Modeling Language (SysML) are used to visualize and informally assess models of system structure and behavior, this chapter describes the pathway from textual requirements to formal representations for rules. The Semantic Web Rule Language (SWRL) will be used for the representation of design constraints (rules) or domain regulations. Ontologies will be created with Protege tool. Inference and rule checking in ontology models will be handled by the Pellet Reasoner.

2.2 Semantic Web Vision

In his original vision for the World Wide Web, Tim Berners-Lee described two key objectives [7]: (1) To make the Web a collaborative medium, and (2) To make the Web understandable and, thus, processable by machines. During the past twenty years the first part of this vision has come to pass – today's Web provides a medium for presentation of data/content to humans. Although, machines are used primarily to retrieve and render information, humans are still expected to interpret and understand the meaning of the content.

The Semantic Web [26] aims to give information a well-defined meaning, thereby creating a pathway for machine-to-machine communication and automated services based on descriptions of semantics [20]. The realization of this goal will require mechanisms that can work and reason with data and semantic description of data. In our view, future generations of computer support for the exchange, management, and visualization of design data will make use of Semantic Web technologies.

2.3 Technical Infrastructure

Figure 2.2 displays the technical infrastructure that supports the Semantic Web vision. Each new layer builds upon, and provides compatibility with, the layers of technology below it. The bottom layer provides standardized support hypertext web (i.e., via Universal Resource Identifiers) and multi-lingual languages. URIs are a generalized mechanism for specifying a unique address for an item on the web. Unicode serves an important role in representing and manipulating the text and documents in different human languages. Moving to the next layer, XML, is a markup language that enables the construction and management of documents composed of structured portable data. XML grew out of demands to make the hypertext markup language (HTML) more flexible. The technology itself has two aspects; First, it is an open standard which describes how to declare and use simple tree-based data structures within a plain text file (human readable format). XML namespaces provides a way for the markups from multiple sources to be used. This is similar to the use of namespaces in Java. A second key benefit in representing data in XML is that we can filter, sort and re-purpose the data for different devices using the Extensible Stylesheet Language Transformation (XSLT) [44, 48].

The middle layers of Figure 2.2 the resource description framework (RDF) allows for the representation of statements in the form of triples, and collections of triples in the form of graphs (RDF graphs). The graph-based nature of RDF means that it can resolve circular references, an inherent problem of the hierarchical structure of XML. An RDF Schema (RDFS) provides the basic vocabulary for RDF

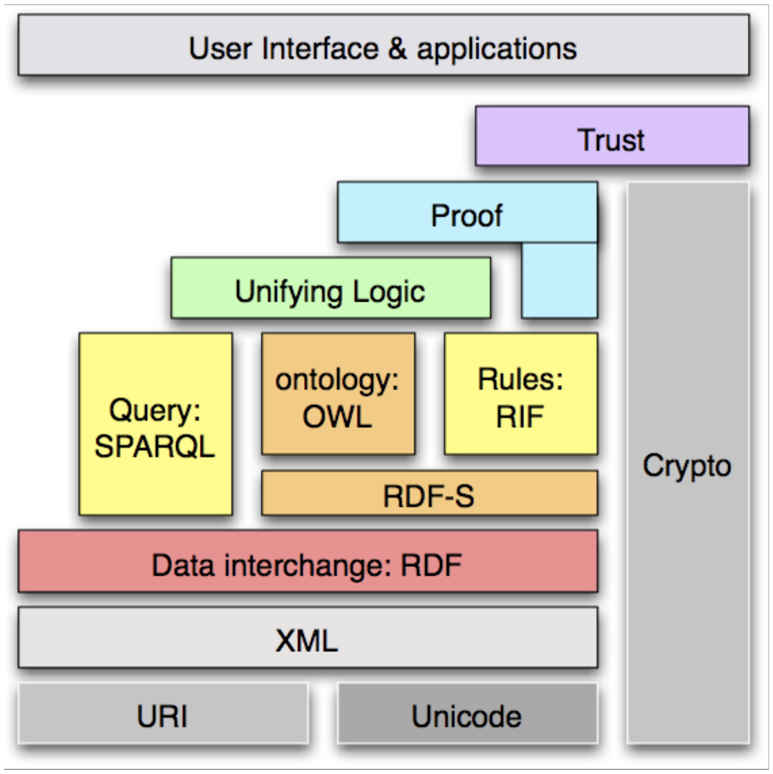


Figure 2.2: Technologies in the Semantic Web Layer Cake

statements, and machinery to create hierarchies of classes and properties. The Web Ontology Language (OWL) extends RDFS by adding: (1) Advanced constructs to describe the semantics of RDF statements, and (2) Vocabulary support for relationships between classes (complementOf and disjointWith), equality of classes (sameAs, equivalentClass), richer properties (symmetric, transitive, inverseOf), and restrictions on properties (cardinality, someValuesFrom, allValuesFrom) [46]. Together, these features and language capabilities provide the foundations for reasoning with first order and descriptive logic.

The top layers of Figure 2.2 are proposals to enhance the semantic web and are yet standardized. The Semantic Web Rule Language (SWRL) will bring support for rules in way that cannot be directly described using first order logic. The

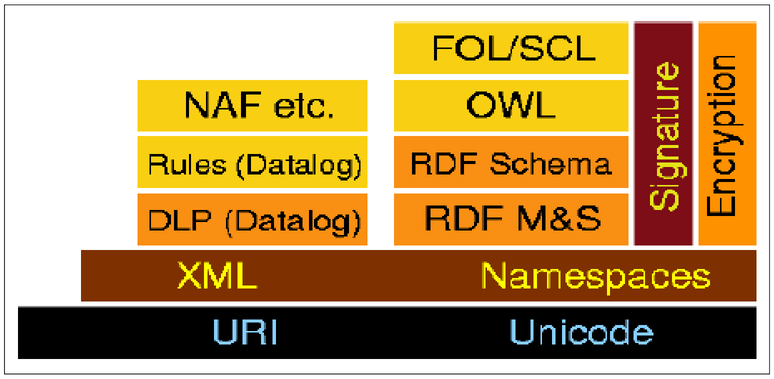


Figure 2.3: Two Towers Architecture for the Semantic Web (Source: Horrocks et al. [28]).

proof and trust layers introduce logical reasoning, establishment of consistency and correctness, and evidence of trustworthiness into the Semantic Web framework. Encampusing all the layers, cryptography ensures and that Semantic Web statements are coming from a trusted source. And finally, the user interface layer enables humans to use semantic web applications.

Recent Modification to the Semantic Web Layer Cake. The Semantic Web Layer Cake architecture shown in Figure 2.2 has been the de facto representation for Semantic Web for the past decade. Recent work by the W3C Web Ontology Working Group [5] has included instantiating the ontology layer with OWL and development of the rules layer. Horrocks and co-workers [28] assert that the current semantic web stack has some fundamental misconceptions on the relationships between the various languages. To fix this problem they propose the modified two towers structure shown in Figure 2.3. In this new architecture stack, Descriptive Logic Programs (DLPs) are layered on top of RDFS and form a common base for parallel rules (presumably intended as datalog/logic programming style rules) and OWL layers.

2.4 Working with Ontologies, Rules and Reasoners

Pathway from SysML and UML to Semantic Web. In state-of-the-art systems engineering practice, UML class diagrams and SysML block diagrams are commonly used to represent the structural aspects of a system. Occasionally they are also used to represent relationships among domain concepts. The key problem with these diagrams is their semi-formal semantics and their inability to support formal assessment a system. We assert that this problem can be overcome through the use of ontology-enabled rule sets and rule engines. Semantic web technologies can be used in offline applications, to check for inconsistency in the design domain (inconsistent relationships in the ontology) or by rule checking mechanisms for any constraint violations.

Pathway from Textual Requirements to Rules. Table 2.1 summarizes some of the key characteristics of good requirements [10]. During the early stages of system development, it is commonplace to express requirements in a natural language format. Natural language descriptions are easy to develop, but suffer from ambiguity in intent. To this end, and in the move from UML to SysML, systems engineers added the requirements diagram to the suite of visual modeling formalisms. A requirements diagram holds a textual representation of a user's need and/or physical constraints in the system.

Requirements can be classified as being either functional (FR) or non-functional (NFR). The purpose of a FR is to specify one or more actions that a

Attribute	**Description**
Complete	There is no missing information regarding this requirement.
Consistent	This requirement does contradict with other requirements.
Traceable	Confirms that the requirement satisfies the need (no more - and no less than what is required).
Feasible	The requirement can be implemented within the constraints of the project.
Unambiguous	The requirement is concisely stated and it is clear what it means.
Verifiable	The implementation of the requirement can be tested through basic possible methods.

Table 2.1: Characteristics of good requirements.

system must be able to perform, without considering physical constraints [31]. For example, in a transportation network, a functional requirement might be:

The objects shall move from one point to another point in the network.

NFRs complement FRs by placing restrictions or performance or interface constraints on the system under development. Typically, these constraints will affect the internal architecture, design, implementation, or testing decisions. For example, the corresponding NFR for the transportation network application might place a bound on the allowable time that can be taken to traverse the network. The solution to NFRs can only be good and bad, but not right or wrong.

2.4.1 Ontology Concepts and Models

Ontologies provide a way of viewing the domain of interest, and for organizing information. Ontologies are also required for sharing a common vocabulary and the meanings for different terms. Computer science uses ontologies to describe specific conceptual terms and relationships in a standardized machine readable format [6, 27]. Ontologies can be represented either graphically or in a structured text format using different languages and standards, i.e., OWL, RDF. The former is usually used when the goal is to visualize a shared understanding of the domain and the relationships among those concepts. And the latter approach is used for computer applications that use the ontology facilitates programmatic exploration and editing of ontologies.

Individuals are instances of classes, and properties are used to relate one individual to another individual. This property is defined for the class and individuals own it as a member of the class. Properties state relationships between individuals or from an individual to data value. Property can be of the types ObjectTypeProperty or DataTypeProperty. ObjectTypeProperty defines the relation between instances of two classes. A DataTypeProperty defines the relation between instances of classes and literal values such as string, number, boolean, and date.

Example 2.1. A University Ontology. Tables 2.3 and 2.2 summarize the data and object properties associated with the concepts of professor and course, part of a university domain. The university ontology specification provides basic description

Course	Datatype Property	Object Property
	hasTitle :text hasSection :int numberOfCredits :int hasCapacity :int	hasInstructor :Professor belongsToDepartment :Department hasStudents :Student hasPrerequisite :Course

Table 2.2: University Ontology - Course Class

Professor	Datatype Property	Object Property
	hasName :text hasRank :text hasEducation :text dateEmployed :date	isTeacherOf :Course hasGraduateStudent :Student hasHomeDepartment :Department isAuthorOf :Book

Table 2.3: University Ontology - Professor Class

of the concepts and properties for describing the university domain. Each of these concepts, or classes, may own a datatype property to represent an attribute or an object property to represent a relationship with another concept.

Ontology Modeling in Protege. Protege is a free, open source ontology software editor and knowledge-base framework [40]. This integrated development environment (IDE) helps the user to create ontology classes, assign properties (datatype and object) to these classes. The output of Protege can be an XML or an OWL file readable for computer programs. The main benefit here is to eliminate the error-prone process of creating the ontology OWL file manually. A second benefit is support for reasoners use these OWL files to for inference and rule checking procedures. Ontologies are built upon classes and the relationships among them. Each class may have none or some individuals (members) with specific values for the attributes of that class.

Example 2.2. A Car Insurance Ontology. Figure 2.4 shows a sample car insurance ontology created in the Protege environment.

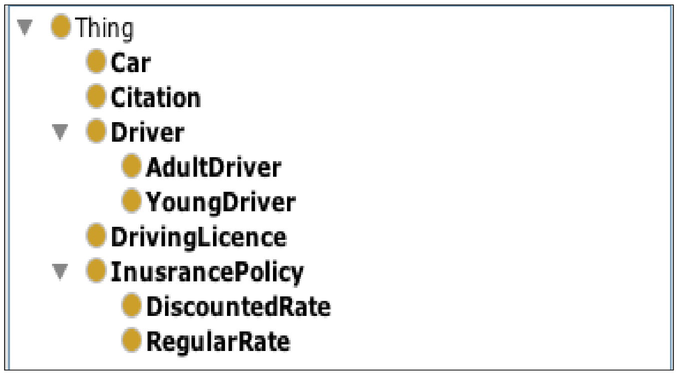
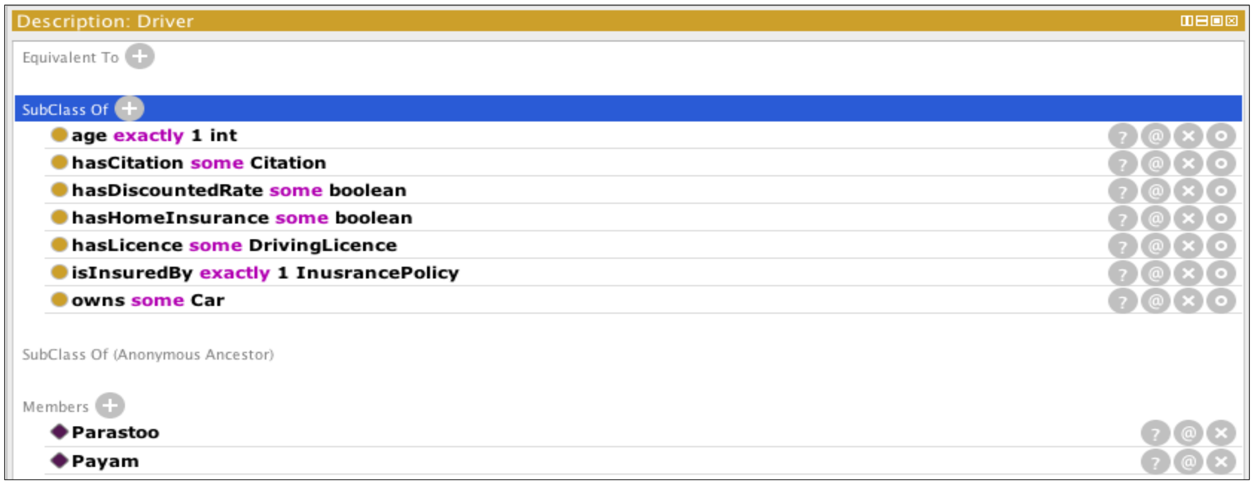


Figure 2.4: Car insurance ontology.

The key concepts of this domain are the “Car,” “Citation,” “Driver,” “InsurancePolicy” and “DrivingLicense” classes. The “Driver” and “insurancePolicy” classes are generalizations (i.e., super classes) of the “AdultDrive,” “YoungDriver,” “DiscountedRate” and “RegularRate” classes, respectively. Figure 2.5 is a close-up view of the driver class; in this class “hasCitation” is an ObjectTypeProperty which represents the relationship between “Driver” and “Citation” classes. Furthermore, “age” is a DataTypeProperty that shows the value of this attribute has to be an integer. And Figure 2.6 represents the car class in that ontology – it shows that the the connection between these two classes is through the ObjectTypeProperty link “Driver” “Owns” “Car.” This class has two members: “Honda” and “Hyundai.” Similarly, in Figure 2.6, the “Car” class is connected to the “Driver” and “InsurancePolicy” classes through the “OwnedBy” and “Insured” properties, respectively.



Description: Driver

Equivalent To

SubClass Of

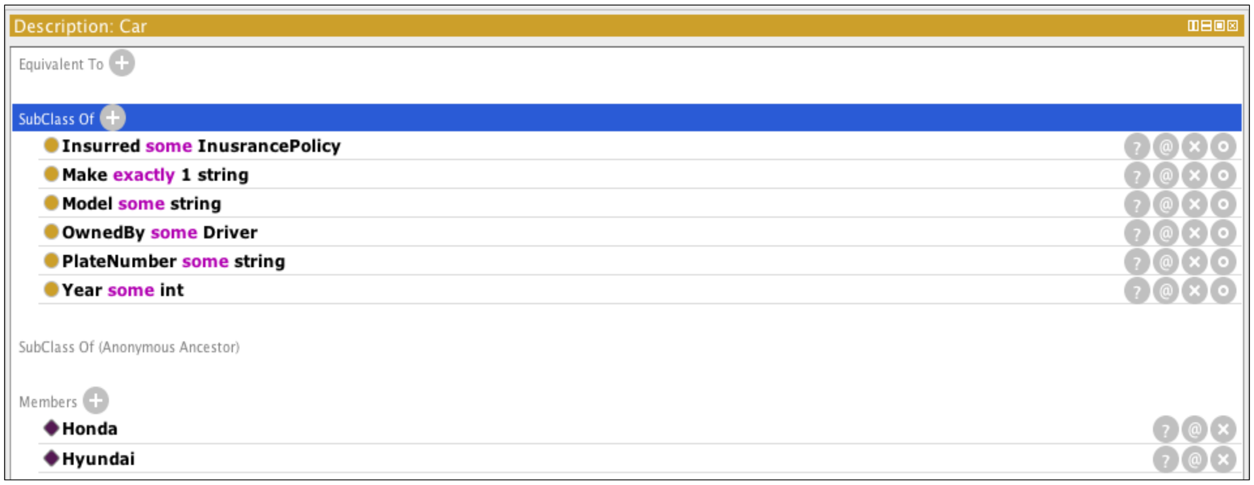
age exactly 1 int
hasCitation some Citation
hasDiscountedRate some boolean
hasHomeInsurance some boolean
hasLicence some DrivingLicence
isInsuredBy exactly 1 InusrancePolicy
owns some Car

SubClass Of (Anonymous Ancestor)

Members

Parastoo
Payam

Figure 2.5: Detailed description of the “Driver” class in the insurance ontology.



Description: Car

Equivalent To

SubClass Of

Insurred some InusrancePolicy
Make exactly 1 string
Model some string
OwnedBy some Driver
PlateNumber some string
Year some int

SubClass Of (Anonymous Ancestor)

Members

Honda
Hyundai

Figure 2.6: Detailed description of the “Car” class in the insurance ontology.

2.4.2 SWRL Rules

As previously mentioned, rules are used to impose the physical constraints or regulations applied to the concepts of a system ontology. The underlying idea of a rule engine is to externalize problem solving logic. Therefore, a rule engine can be viewed as a sophisticated interpreter of if-then statements. The if-then statements are the rules. An individual rule is composed of two parts, a condition and an action: when the condition is met, the action is executed. The inputs to a rule engine are a collection of rules and data objects (i.e, the domain model). The outputs are determined by the inputs, and may include the original data objects with modifications, new data objects, and possible side effect actions.

Example 2.3. Reasoning with SWRL. The following examples demonstrate the capabilities of SWRL. First, consider the rule:

```
hasEngine( ?M, ?e ) ^ checkEngineStatus(?E,false) -> Defective(?M)
```

This rule states that if a motor has an engine and the engine the status of the engine is failed, then the motor is defective. SWRL has a number of built-in functions that can be utilized for basic algebraic or logical expressions. Suppose, for example, that we want to use peak acceleration as a measure for vehicle classification. The rule:

```
Engine(?C) ^ hasAcceleration(?C, ?A) ^ swrlb:lessThanOrEqual(?A, 4) -> SportsCar(?C)
```

states that if the peak car acceleration greater than or equal to 4 m/s^2, then the car is considered as a sport car. Finally, the script of code:

```
Person(?x)^ hasAge(?x, ?age) ^ calorieIntakePerDay(?x, ?caloriesConsumed) ^
swrlb:greaterThan(?age, 55) ^ swrlb:subtract(?deltaAge, ?age, 55) ^
swrlb:multiply(?deltaCalories, ?deltaAge, 30) ^
swrlb:subtract(?recommendedCalories,  2550, ?deltaCalories) ^
swrlb:greaterThan(?caloriesConsumed, ?recommendedCalories) -> isOvereating(?x, true)
```

shows how SWRL can be used to define the constraints in the domain. This example talks about the amount of calories one person takes and considering the age he or she can be categorized as an over eating person.

2.4.3 The Pellet Reasoner

This project employed Pellet as the external reasoner to the ontology. Pellet can can infer logical consequences from a set of asserted facts or axioms, and can be utilized for inconsistency checking prior to rule checking procedures. The snippet of code:

```
Reasoner r = PelletReasonerFactory.theInstance().create();
```

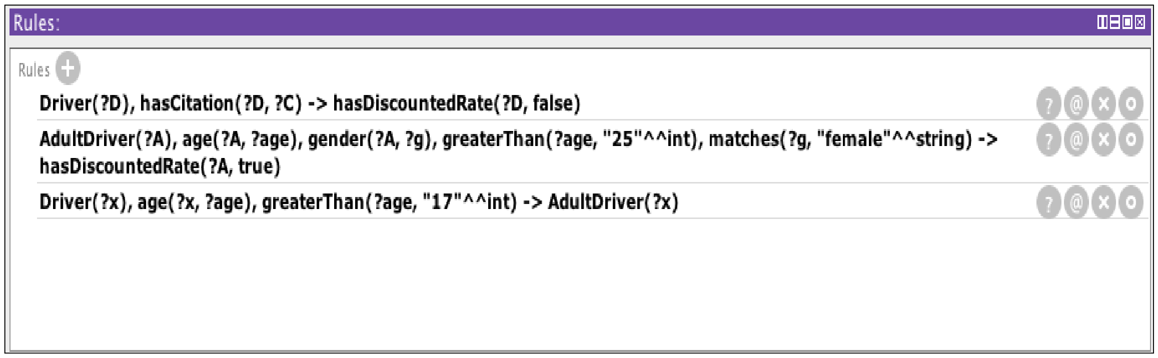
creates an instance of the Pellet reasoner and attach it to an ontology. Obtaining the ontology OWL file created with Protege, and associating it with a reasoner is easy:

```
String ont = "file:./data/WashingtonMetro.owl";
model      = ModelFactory.createOntologyModel(r);
model.read( ont );
```

When changes occur in the ontology, the reasoner is notified and ontology will be updated based on the reasoner's inference on the modified data.

Example 2.4. Reasoning for Driving Insurance Premiums. In the car insurance ontology example, the insurance companies policies can be captured via SWRL rules.

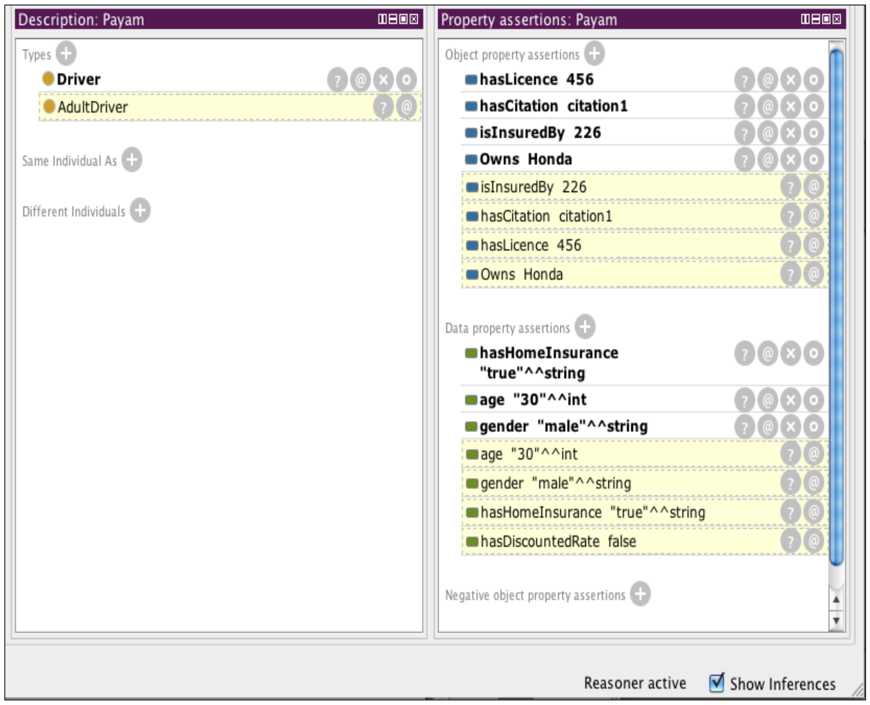
Figure 2.7 shows three different rules implied in this ontology: (1) If a driver has a citation, he can not benefit from discounted premium rate, (2) A female driver older than 25 can get discount on her premium rate, and (3) If someone is older than 17, then they are considered to be an adult.



```
Driver(?D), hasCitation(?D, ?C) -> hasDiscountedRate(?D, false)
AdultDriver(?A), age(?A, ?age), gender(?A, ?g), greaterThan(?age, "25"^^int), matches(?g, "female"^^string) ->
hasDiscountedRate(?A, true)
Driver(?x), age(?x, ?age), greaterThan(?age, "17"^^int) -> AdultDriver(?x)
```

Figure 2.7: SWRL Rules associated with insurance car Insurance premiums.

Figures 2.8 and 2.9 are snapshots of the two driver individuals created in the Protege environment. The yellow highlighted boxes represent data that has been inferred through reasoning procedures and application of the Pellet Reasoner. Notice that due to their age values and rule (3), both members are categorized as adult drivers. Furthermore, Figure 2.8 shows how rule (1) is verified, and Figure 2.9 is a proof for rule (2).


Description: Payam

Types
Driver
AdultDriver

Same Individual As

Different Individuals

Property assertions: Payam

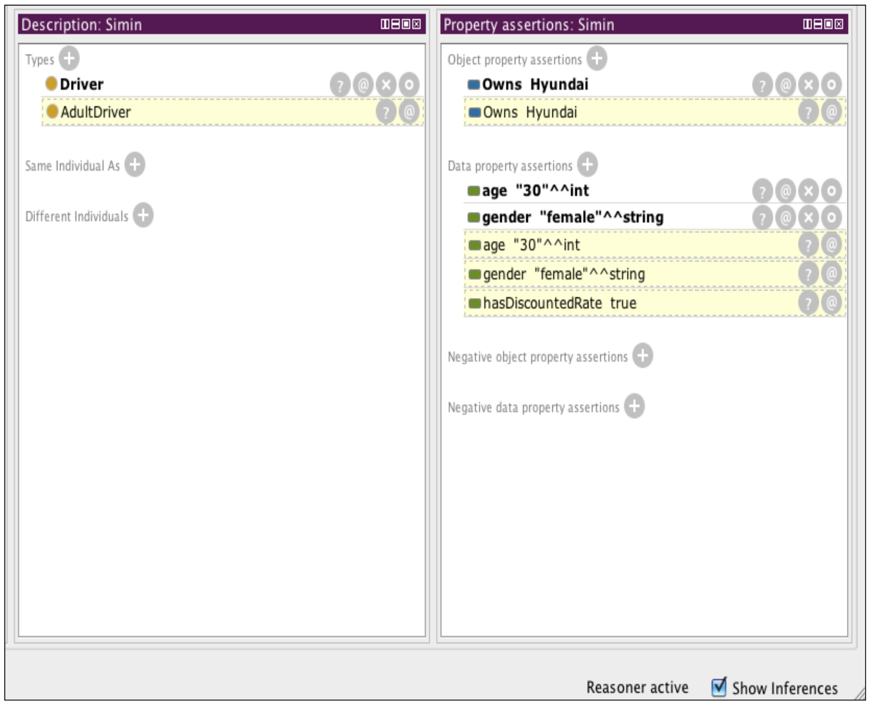
Object property assertions
hasLicence 456
hasCitation citation1
isInsuredBy 226
Owns Honda
isInsuredBy 226
hasCitation citation1
hasLicence 456
Owns Honda

Data property assertions
hasHomeInsurance "true"^^string
age "30"^^int
gender "male"^^string
age "30"^^int
gender "male"^^string
hasHomeInsurance "true"^^string
hasDiscountedRate false

Negative object property assertions

Reasoner active ☑ Show Inferences

Figure 2.8: Example of an "AdultDriver" member with a "Citation."


Description: Simin

Types
Driver
AdultDriver

Same Individual As

Different Individuals

Property assertions: Simin

Object property assertions
Owns Hyundai
Owns Hyundai

Data property assertions
age "30"^^int
gender "female"^^string
age "30"^^int
gender "female"^^string
hasDiscountedRate true

Negative object property assertions

Negative data property assertions

Reasoner active ☑ Show Inferences

Figure 2.9: Example of a female "AdultDriver."

Chapter 3

Software Design Patterns and System Architecture

3.1 Problem Statement

The implementation of software solutions for ontology-enabled traceability is complicated by the very nature of the problem to be supported. Traceability mechanisms are required to provide linkages between discipline-specific domains and across various stages of system development (e.g., requirements, design, management). Each domain will have its own way of modeling and visualizing their concerns. Teams of engineers will work concurrently on the various stages of a systems development. These requirements dictate the traceability mechanisms will provide linkages across heterogeneous models, multiple types of visualization, and will need to synchronize data/information models and views, and across concurrent processes. We would also like software solutions to be scalable. These complexities indicate a strong need for a disciplined approach to software development, based upon patterns of implementation that have worked well in the past and, perhaps, even new patterns of development. To that end, this chapter describes the software patterns employed in our implementation described in Chapters 3 through 5.

3.1.1 Definition and Benefits

Definition. Experienced designers know that instead of always returning to first principles, routine design problems are best solved by adapting solutions to designs that have worked well for them in the past. A design pattern is simply: (1) A description of a problem that occurs over and over again, and (2) A description of a core solution to that problem stated in such a way that it can be reused many times [1, 19]. In other words, a design pattern prescribes a [problem, solution] pair. The design pattern identifies the participating subsystems and parts, their roles and collaborations, and distribution of responsibilities.

Benefits. For a wide range of domains, this approach to problem solving is popular because it encodes many years of professional experience in the how and why of design, and is time efficient. Design patterns crop up in many avenues of day-to-day life. For example, that layout of streets in planned communities follows familiar patterns [1]. Gamma and co-workers [19] point out that patterns facilitate reuse – one persons pattern can be another persons fundamental building block. Software design patterns are particularly beneficial in the development of architectures for distributed systems.

3.2 Software Design Patterns used in this Project

We believe that behavioral patterns, such as the Observer pattern, will play a fundamental role in the efficient implementation of traceability relationships span-

Behavior	**Structure**	**System**
Command	Adapter	**Model-View-Controller**
Interpreter	Bridge	Session
Mediator	**Composite**	Router
Observer	Decorator	Transaction

Table 3.1: Commonly used software design patterns. Those highlighted in bold are relevant to ontology-enabled traceability.

ning requirements to ontologies to various aspects of system structure and system behavior. Similarly, the model-view-controller pattern (MVC) implemented with appropriate interfaces and abstract class definitions will allow for the assembly of traceability models that are both scalable and readily extensible.

3.2.1 Mediator (Behavior) Design Pattern

The mediator design pattern is used to manage algorithms, relationships and responsibilities between objects.. It mitigates the need for point-to-point connections between objects by defining an object that controls how a set of objects will interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the mediator, rather than with one another.

To see why this pattern is useful, the left- and right-hand sides of Figure 3.1 show two approaches to connecting an ensemble of models to a collection of views. In the point-to-point solution, each model is connected to each view. For n models and n views this requires O(n^2) interfaces.

This strategy of development simplifies communication between models and

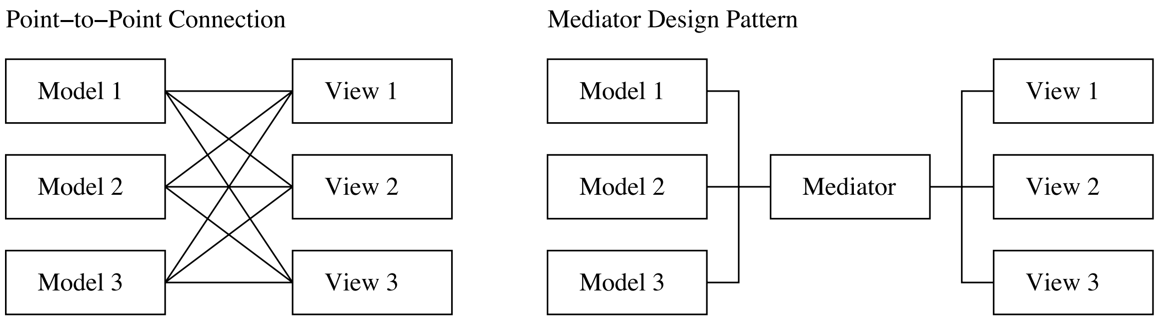


Figure 3.1: Communication between models and views. left: point-to-point connection, right: use of a mediator.

views because they do not need to implement the specific details of communication with each other. Moreover, this pattern provides maximum flexibility for expansion, because the logic for the communication is contained within the mediator.

The mediator pattern can be combined with the classical model0view-controller (MVC) pattern, resulting in a hybrid MVC design pattern[38]. In this project, controllers will be implemented as part of the model-view-controller design pattern, a means for synchronizing data among sets of models and views. The controllers will also act as mediators of communication. Further details will be provided in Section 3.2.4.

3.2.2 Observer (Behavior) Design Pattern

The observer pattern is applicable to problems where a message sender needs to broadcast a message to one or more receivers (or observers), but is not interested in a response or feedback from the observers.

Figure 3.2 shows the relationship of classes and interfaces in an implementation of the observer design pattern. During the problem setup, an observer will

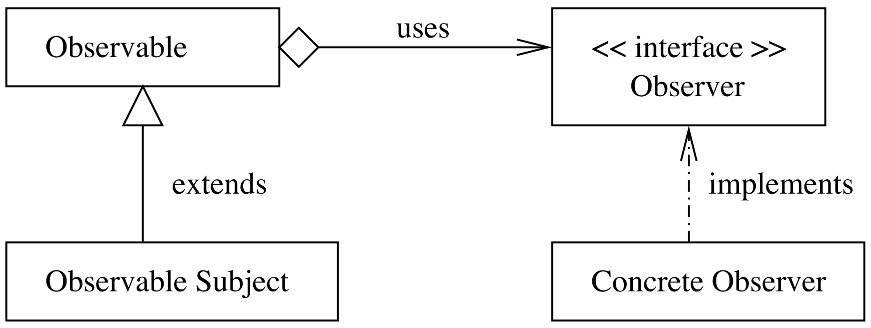


Figure 3.2: Relationship of classes and interfaces in the observer design pattern.

register for changes in the state of an observable object. When such a change occurs, the observer is notified and will handle the change through and event handler method.

Perhaps the best known application of the observer design pattern is the the graphical user interface (GUI) event model used in Java. Graphical components (listeners) register for changes in other components by implementing the PropertyChangeListener interface – one can think of the event source component as a discrete observable, and the event change listener as a discrete observer.

In this study, the observer pattern will be used to notify the controllers about a property change that has occurred in the model. There a a number of ways of achieving this functionality, including use of the

```
firePropertyChangeEvent( propertyname, old value, new value );
```

method in Java. This feature facilitates the communication from model to the controller when a model property has changed. Graphical components (listeners) will register for changes in other components by implementing the PropertyChangeListener interface. This feature is used in the views where they can be added to the components of GUI and be notified when an event happens. i.e., the view is notified when a curser is moved on the GUI panel.

3.2.3 Composite Hierarchy (Structure) Design Pattern

In many systems engineering applications, system structures have a hierarchal design and follow a tree shape structure. The composite hierarchy design pattern provides a flexible way to create hierarchical tree structures (i.e., part-whole hierarchies) of arbitrary complexity, while enabling every element in the structure to operate with a uniform interface.

Figure 3.3 shows the class diagram for the composite design pattern. Implementations of this pattern employ component, node, and composite classes:

1. **Component.** The component interface defines methods available to all parts of the tree structure.

2. **Composite.** This class is defined by the components it contains. It supports a dynamic group of Components, and so it has methods to add and remove Component objects from its collection. The Composite class also provides concrete implementations of operational methods defined in the interface class Component.

3. **Node.** The node (or leaf) classes represent terminal behavior (i.e., parts of the composite that cannot contain other components). They implement the Component interface and provides an implementation for each of the component's

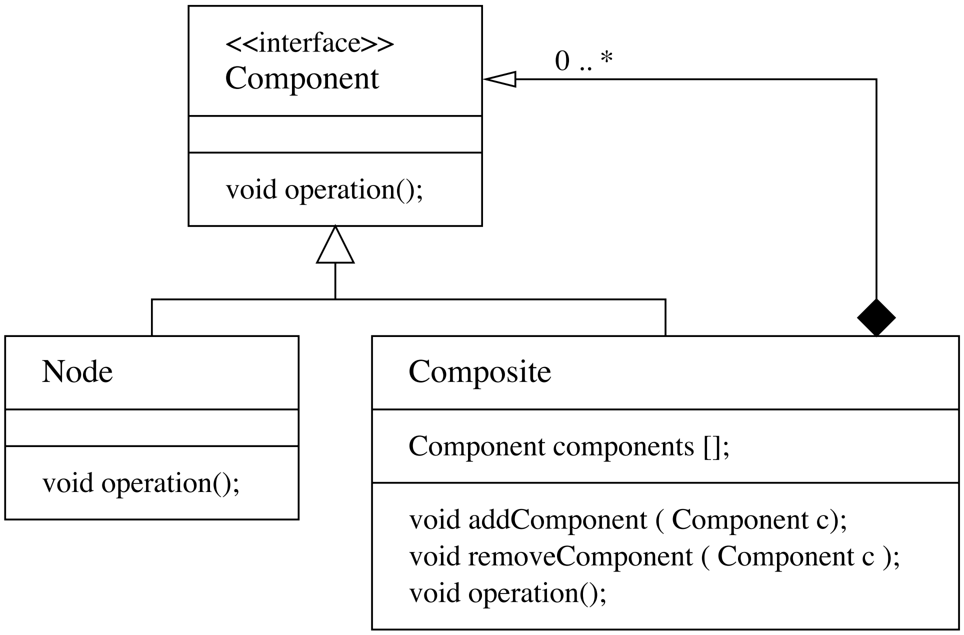


Figure 3.3: Composite class diagram.

operational methods.

The composite hierarchy pattern in very general – composites can contain components, each of which could be a composite. Traversal of the composite hierarchy occurs through recursion that continues to the lowest level where a node (leaf) is obtained.

3.2.4 Model-View-Controller (System) Design Pattern

The model-view-controller (MVC) design pattern divides a subsystem into three logical parts the model, view and controller and offers a systematic approach for modifying each part. As pointed out by Martin Fowler [17], this design pattern was one of the first attempts at creating systematic approaches to user-interface (UI development in large scales applications. At the heart of MVC is a clear division between domain objects that model the perception of the real world, and presentation objects that represents the GUI elements we see on the screen. Domain objects (model) should be completely self contained and independent of the presentation, and they should also be able to support multiple presentations at the same time. This is what is been used in current technology which allows many applications to be manipulated through both a graphical and command-line interface[36].

Figure 3.4 illustrates two approaches to implementation for MVC. In the most common implementation of this pattern (see, for example, the Java patterns in Stelting and Maasson [43], views register for their intent to be notified when changes to a model occur. Controllers register their interest in being notified of changes to a view. When a change occurs in the view, the view (graphical user interface) will query the model state and call the controller if the model needs to be modified. The controller then makes the modification. Finally the model notifies the view that an update is required, based on a on change in the model.

In the second approach to the implementation of MVC, the controller is positioned at the center of the pattern and the models and views communicate through the controller channels. For example, after a view has notified the controller of a user action, the controller will update the property in the model based upon that action. From other direction, the controller registers for the changes in the model and updates the view based on the notification triggered from the model. This approach is combined with the mediator where the controller plays the mediator role for model and view communications.

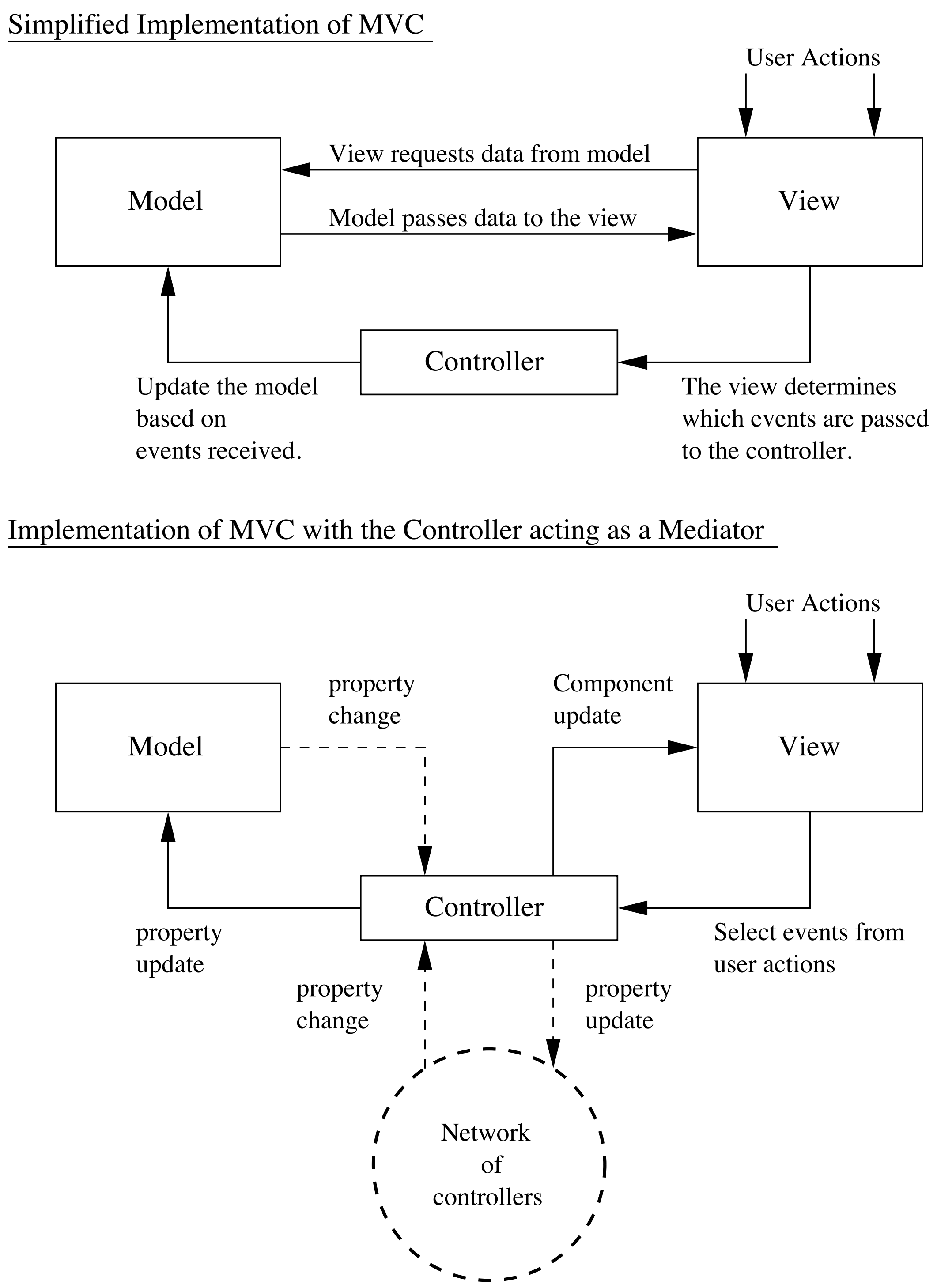

Simplified Implementation of MVC
User Actions
Model
View requests data from model
Model passes data to the view
View
Controller
Update the model based on events received.
The view determines which events are passed to the controller.
Implementation of MVC with the Controller acting as a Mediator
User Actions
Model
property change
Component update
View
Controller
property update
Select events from user actions
property change
property update
Network of controllers

Figure 3.4: Styles of model-view-controller implementation.

3.3 System Architecture Design

This section explains how a combination of software patterns, workspaces, and mechanisms for distributed computing will be used in the design and implementation of a systems architecture for ontology-enabled traceability. The fully developed system will have separate workspaces for the requirements, ontology and engineering phases of system development, plus a time workspace responsible for delivering temporal information to the system model via clocks and timers.

3.3.1 Requirements, Ontology, and Engineering Workspaces

The term “workspace” is most often used to describe a space allocated for work (e.g., such as an office) or an area reserved for some particular purpose. From a systems engineering standpoint, the concept of workspaces is attractive because it provides a way to develop work (i.e., data, information, models, products) independent of other activities occurring in a larger system view.

Workspaces for Ontology-Enabled Traceability. Figure 3.5 represents a complex system design, where low level (system) requirements are derived from various viewpoints constraints. These viewpoints can belong to different stakeholders ranged from users to design and manufacturing team. On the other hand, in ontology-enabled-traceability mechanisms, a path is required from a requirement to at least a concept or relationship in the ontology and ultimately to an engineering object that may have a certain behavior (statechart) at each viewpoint. In other worlds, this

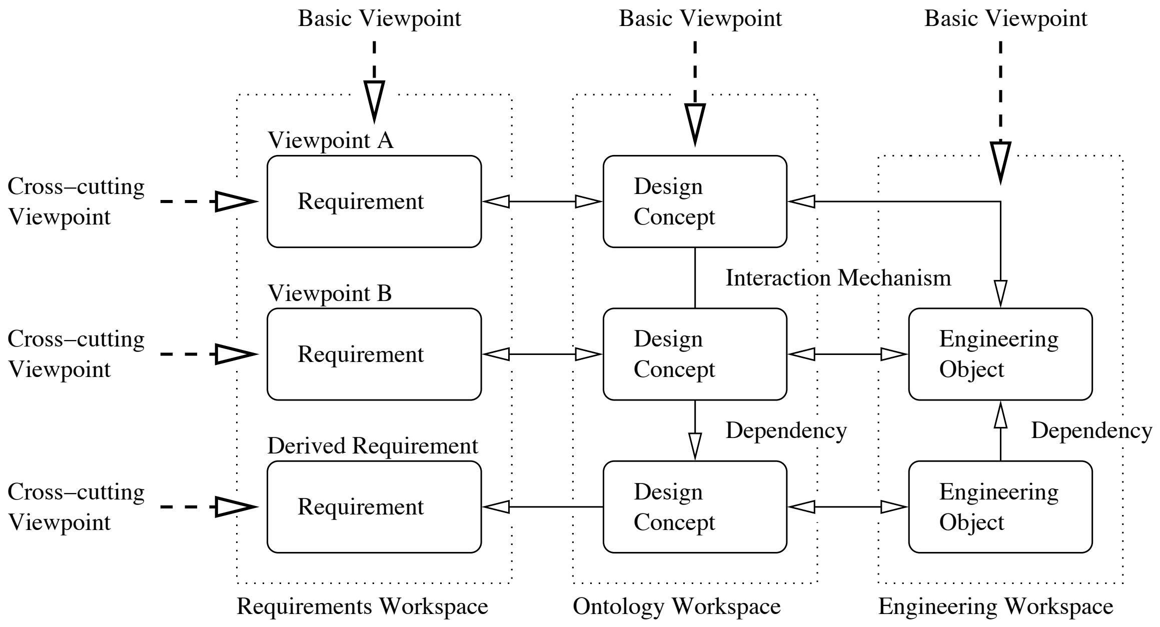


Figure 3.5: Multiple-viewpoint design support partitioned into requirements, ontology and engineering workspaces.

architecture is scalable to expand vertically by covering more viewpoints or more levels of details (system, subsystem, component). However, the horizontal axis is fixed and provide information about the requirements, ontology, engineering objects in the notion of workspaces and the connectivity between them. To achieve these goals, the requirement, ontology and engineering workspaces are created to store various viewpoints and their associated models. Moreover, the connection between the workspaces provides traceable threads from requirements to ontology and to engineering objects and vice versa.

3.3.2 Distributed Implementation

We expect that real-world implementations of ontology-enabled traceability will need to handle hundreds, and possibly thousands of requirements, dozens of ontologies, and engineering models containing thousands of components and connections among components. A practical way of handling and presenting information associated with each of these design concerns is to distribute them across multiple computers, with machines dedicated to supporting a particular phase of the systems engineering development. This is why the use of workspaces in Figure 3.5 makes sense.

Figure 3.6 shows the system architecture currently being implemented as a pyramid (i.e., a two-level graph) of model-view-controllers. The systems relationship hub (SRH) will be responsible for defining high-level system development entities and their initial connections, and then systematically assembling the graph infrastructure to mimic the graph structure and workspaces shown in Figure 3.5. Each block will employ a combination of the mediator and model-view-controller design patterns. The requirements block (Figure 3.6) is expanded to show the details of model, view and controller and how controller is being in place as a mediator between the model and views. The requirements, ontology and engineering workspaces will be implemented as networks of model-view-controllers (MVCs). This means that: (1) Models do not communicate with other models, (2) Views do not communicate with other views. Instead, models and views can only communicate with their controllers. Thus, workspaces communicate via interaction mechanisms between

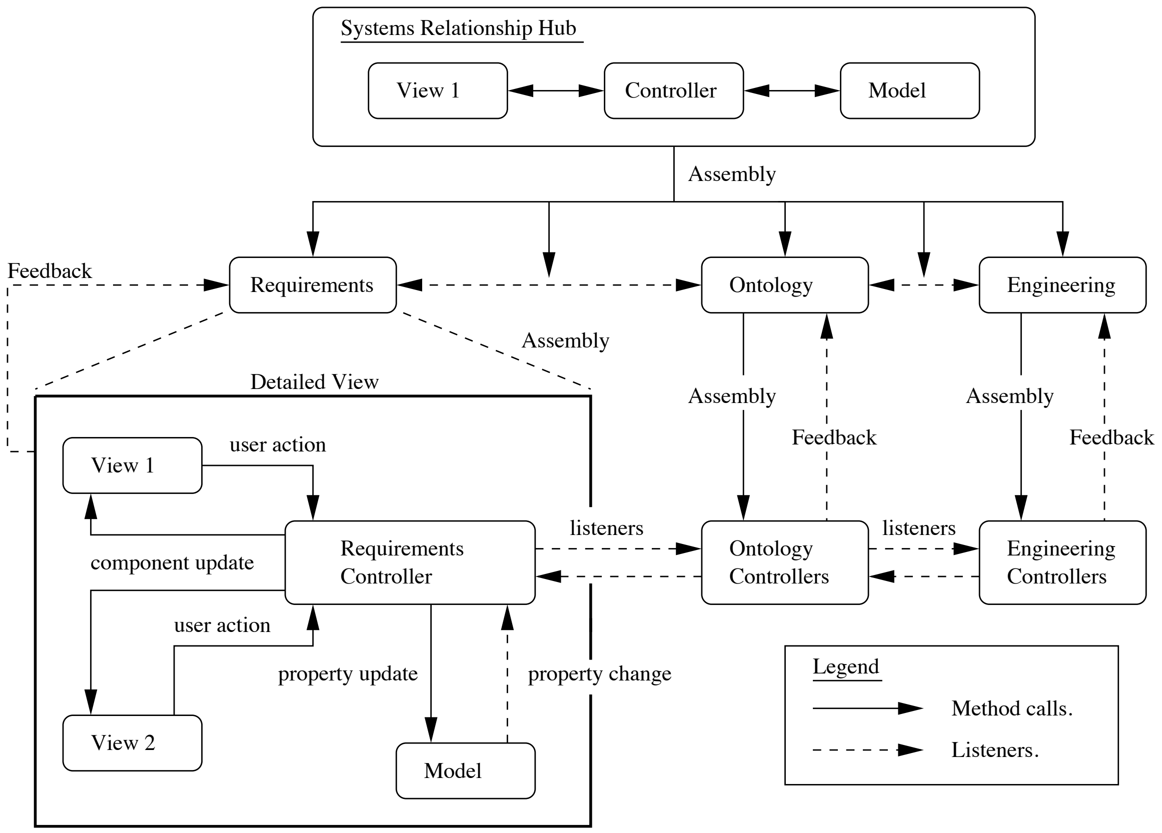


Figure 3.6: System architecture implemented as a network of model-view-controllers.

controllers. Therefore, the fully implemented system will be a network of MVCs.

Engineering Workspace. Engineering workspace models will be implemented as composite hierarchies, a natural structure for subsystem organization (and network modeling therein). This design practice reduces the complexity of modeling connections between those pieces and provides a scalable, reusable architecture where new objects can be added, removed and altered at different levels of abstraction without affecting the rest of system. As a case, an engineering system can have a component (leaf level) and it can have a subsystem as another composite which itself has a number of components.

Synchronization of Workspaces. An event-based model (observer design pattern) will be used for synchronization of states and data in views and models (Observer design pattern). The expanded requirements box in Figure 3.6 shows how this will work. User actions detected in views (e.g., a user highlights a row in a table) will be propagated to the controller, which in turn, will forward the updates to related views within the same workspace and to the controllers of connected workspaces. Similarly, property changes to a model will be sent to the controller for distribution to related viewpoints and to the controllers to connected workspaces.

Note: Create a connection between ontologies to the concept of "body of knowledge" – summarize key points from the three papers given to us by Srini at NIST.

Chapter 4

Modeling Behavior with Statecharts

4.1 Overview

In model-based systems engineering, state machines (statecharts) are used to describe the state-dependent behavior of a system component throughout its life cycle. Common applications of statecharts include the real-time behavior modeling of computer systems (e.g., computer software; operating systems; telecommunication protocols) and embedded systems (e.g., vending machines; elevators in buildings; electronic systems in automobiles). Statecharts are widely used in SysML to describe systems having complexity ranging from switches in a simple lamp to very complex control units in aerospace applications.

This chapter describes the use of statecharts as a formalism for modeling system behavior that is described in terms of responses to internal and external events. It presents a software implementation of statecharts that is cast into the model-view-controller design pattern format. The chapter concludes with four standalone working examples (e.g., a countdown timer) that illustrate the various features of statechart behavior and serve as a stepping-stone to railway transportation system case study covered in Chapter 5.

4.2 Finite State Machines

A finite-state machine (FSM) is a mathematical model of computation for an abstract machine defined in terms of a finite number of states and transitions, and sequences of input events that will be consumed during the machine's operation. A state is a description of the status of a system that is waiting to execute a transition. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received. At any point in time, the machine can only be in one state (called the current state). When the machine receives an appropriate input event, condition or message, the state machine can transition to a new current state. In some finite-state machine representations, entry and exit actions are performed when entering/exiting a state.

4.2.1 State Transition Diagrams

A state transition diagram is a graphic representation of the real-time (or on-line) behavior of a system.

Figure 4.1 show, for example, a simple finite state machine model for a sensor testing procedure. The rectangular nodes with rounded edges represent states – states are commonly labeled with a name to represent the action that will occur while the state is active. The beginning and end states are drawn as a large black dot and black dot with a surrounding circle, respectively. They are examples pseudo states – that is, a special type of state exists only for the purposes of modeling.

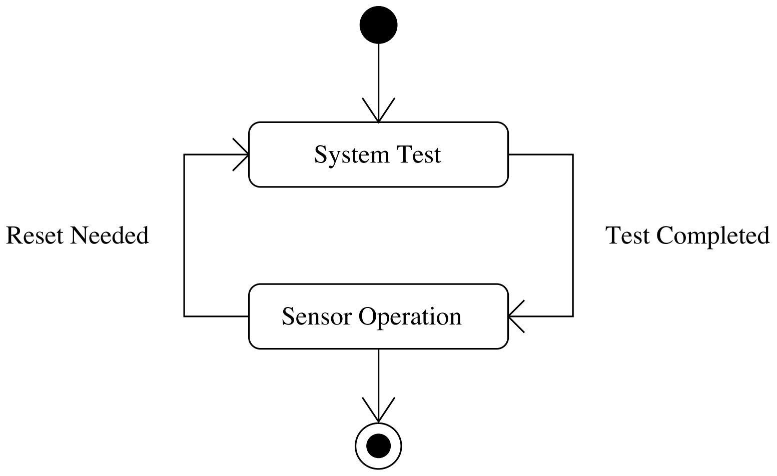


Figure 4.1: Sensor testing procedure.

State transitions are shown as a solid-line arrows (directed edges) originating from the source state and terminating by an arrow at the target state. A state state is a pseudostate that only has outgoing transitions, and the final-state only has incoming transitions. Annotations along the transition edges represent the events, actions and guard conditions that will affect the machine behavior. The sensor testing procedure is very simple – `Reset Needed` and `Test Completed` are the events that transition the machine between the `System Test` and `Sensor Operation` states.

4.2.2 Events, Actions, and Guard Conditions

The transitional behavior of finite state machines can be refined with the association of events with actions and guard conditions.

Events. An event is a kind-of stimulus that can be presented to an object or system, and then the object or system can decide how it wishes to respond. The main categories of events are [18]:

- Signal Event: A signal which arrives from the surrounding environment. It may contain a number of arguments that will be used in execution of the transition.
- Time Event: This event indicates that a specific instance of time (relative) or, timeout interval has reached (absolute).
- Change Event: Indicate that a status of an object in the system has changed and a set of attribute values hold.
- Call Event: Notifies that an operation is been requested. It is very similar to method calls and is possible to be accompanied by number of arguments.

Actions. An action (also known as an effect) is the response given to stimuli in a transition, and will normally correspond to an activity that is performed during a transition. States many also have entry/exit behaviors and "do" activities that will start immediately after the completion of an entry action. Hence, the synopsis of the transition is as follows:

1. The transition is triggered by an event and the guard expression is evaluated. If the guard is evaluated as "true":

2. The exit behavior of the current (source) state is executed.

3. The transition effect (action) is executed.

4. The entry action of the new (target) state is executed

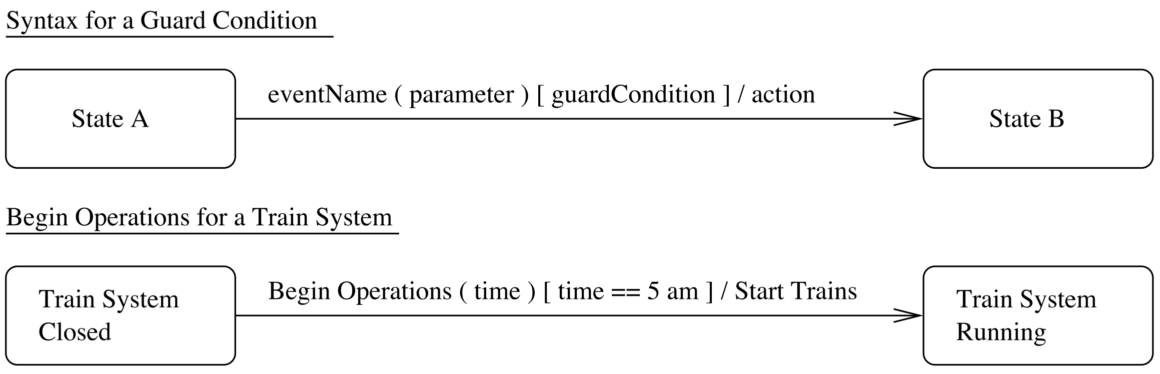
5. While in that state perform "Do Activity."

Every state can have a completion-transition, which is a transition without an event. If the state has no such transition, and all transitions are triggered by events, then a system will stay in a state until an appropriate event can be handled by an outgoing transition.

Guard Conditions. A guard condition is a predicate expression associated with an event. Guard conditions are used to make sure that a transition can only trigger if the evaluation of the guard is true. The syntax for representing cases where an action is triggered by an event is as follows:

```
    eventName ( parameters ) [ guardCondition ] / action
```

From left-to-right, parameters is a comma-separated list of parameters supplied by the event, and guardCondition is a predicate expression in square brackets. A forward slash precedes action, the transition effect.



Syntax for a Guard Condition
State A
eventName (parameter) [guardCondition] / action
State B
Begin Operations for a Train System
Train System Closed
Begin Operations (time) [time == 5 am] / Start Trains
Train System Running

Figure 4.2: Guard conditions: Top. Graphical syntax. Bottom. Use of a guard condition to trigger a change in operations for a train system.

Figure 4.2 shows the graphical counterpart of a guard condition along with a simple example that uses a guard condition to trigger opening operations in a train system.

4.3 Benefits and Limitations of the Basic FSM Model

In computer science circles, many basic implementation of finite state machines have their origins in the consumption of characters or tokens. The benefits of basic state machine models are as follows:

1. Easy to use graphical languages (e.g., in UML and SysML).

2. Powerful mathematical algorithms for synthesis of hardware and software and verification.

However, basic state machine models are limited in several respects:

1. They do not scale well. Even for small- to- moderate sized engineering problems, the number of states can quickly become unmanageable.

2. Basic state machine models only support a single thread of concurrency – that is, at any point in time, only one external event can be recognized, and only a single state may be active.

3. A single state machine cannot directly represent the aggregate behavior of two or more independent processes running concurrently.

Together, these limitations are significant impediments to the behavior modeling of real-world systems. As such, we need a formalism that can allow for complex system behaviors to be decomposed into process hierarchies and as communication networks of simpler finite state machine behaviors.

4.4 Statecharts

Statecharts were developed for the graphical modeling of control requirements in complex reactive systems [23, 24, 25], and to overcome the limitations of basic state machine models discussed in the previous section. Formally, statecharts are a higraph-based extension of standard state-transition diagrams, where:

Statecharts = state transition diagrams + depth + orthogonality + broadcast communication.

Statecharts incorporate all of the semantics of diagrams for basic finite state machine models. Depth refers to the simplification of models achieved by the hierarchical nesting of states (i.e., each state encloses a FSM). Significant reductions in diagram complexity will occur for applications where a system may transition to a new state from many state locations. Orthogonality refers to the modeling of two or more independent control strategies and/or independent behaviors at the same time. Broadcast means that all machines are visible to other. In practical terms, this means that an output action of any process may be sent to and consumed by any another process.

4.4.1 Statechart Model Components

This section describes the statechart model components including the different types of states, regions, transitions and control nodes. To facilitate this process, Figure 4.3 is a big-picture view of the a generic statechart containing the range

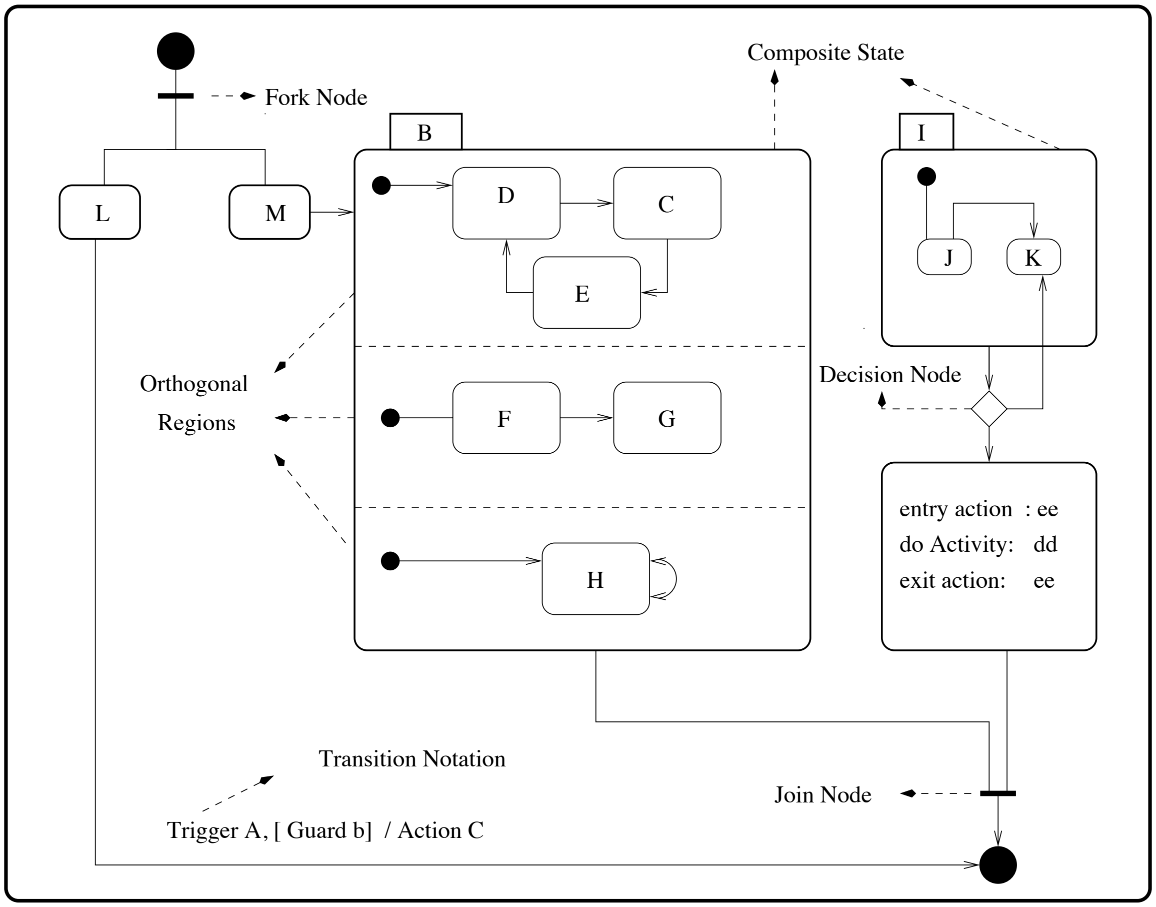


Figure 4.3: Schematic of different types of components in a statechart.

components that will be supported.

Hierarchical, Composite, and Pseudo States

The statecharts formalism supports both hierarchical states and simple states. Hierarchical states (also known as or-state) are used to decompose a composite state into graphs of lower-level substates and orthogonal regions. Each substate (or region) will be defined in terms of states and pseudostates and the transitions between these states, and will operate under the assumption of independent contexts. When the execution of an object reaches a composite state, exactly one sub-state will be automatically activated. Substates in orthogonal regions are concurrent, which means that while the superstate is active, the statechart can be in exactly one substate from each orthogonal region during each statechart iteration. Some implementations will allow for concurrent execution of substates and others will not. Like their finite state machine counterparts, initial and final pseudostates are active upon arrival and/or termination from a statechart. When a final state become active in a superstate, then the region to which it belongs will be completed.

State Transitions

The semantics for handling events and transitions within and across statechart hierarchies is considerably more complicated than for basic finite state machines. Several sources of complication exist. As a case in point, software implementations need to make sure than when need to make sure that when an event-triggered transition occurs from an or-state, every substate will have the opportunity to handle this event as well. To realize this semantics, outgoing transitions from hierarchical-state are inherited by all substates, and this requires modeling of event-triggered transitions be organized into a hierarchy tree.

Control Nodes

Control nodes route the control and data along paths in the statechart. Decision nodes are used to route input to one of several alternative outgoing paths. Merge nodes recombine the data from different paths into one. Fork nodes transfer the data into several parallel output paths. Join nodes synchronize the data coming from different paths into one output.

4.5 Statechart Software Package

Our work on modeling and implementation of event-driven behavior with finite state machines has been adapted from the open source package UML Statechart Framework for Java [45]. Modeling support is provide for: (1) Simple, hierarchical and concurrent states, start and final states, (2) History and deep-history pseudostates in hierarchical states, (3) Fork- and join pseudostates for concurrent states, (4) Segmented transitions using junction points, and (5) Events, guards and actions for transitions. All elements are real objects. Transitions can cross borders of composite states (implicit entry/exit). Unfortunately, support for the visual modeling of statechart behavior is lacking.

This project employed the model-view-controller design pattern discussed in Section 3.2.4, and the mxGraphics package to provide a visualization of statechart behavior. Figure 4.4 shows the high-level software architecture consisting of an abstract controller, abstract models, and abstract views. The primary purpose of these abstract (super) classes is to encapsulate the common functionalities of controllers, views and models, respectively. For example, methods are provided to add and remove listeners in a model, to add and remove models and views in the controller, for notifying the controller of events occurring through interactions in the views, and to notify the controller when the value of a property is changed. The State and Transition classes – detailed below – are extensions of AbstractModel. They send property change events to the statechart controller when the state is entered/exited or a transition is executed, respectively. The controller plays the mediator role by

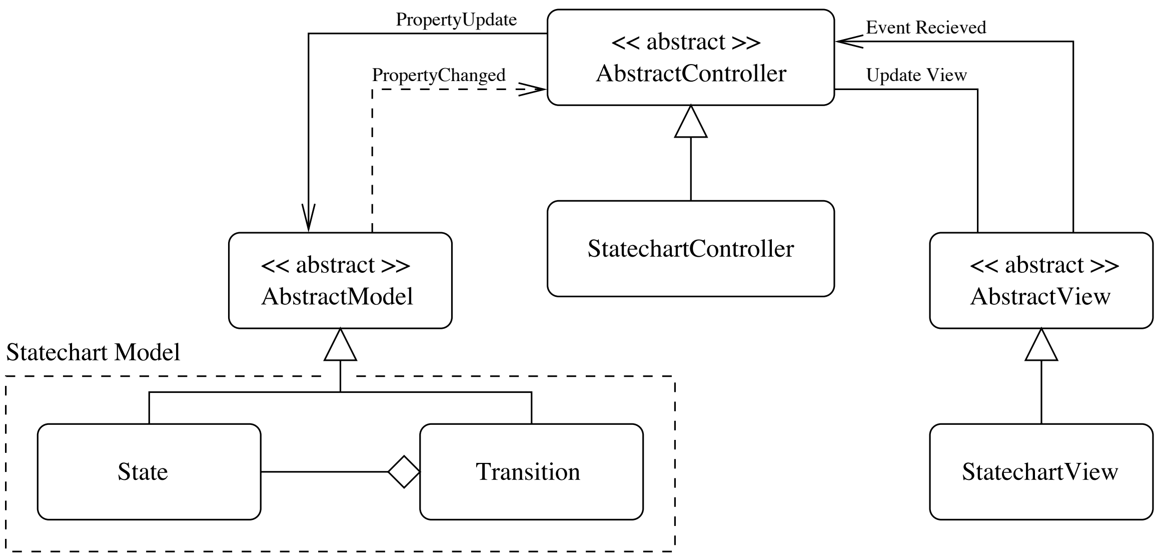


Figure 4.4: Class structure in the UML Statechart Framework modified to work with the MVC design pattern.

passing the messages between the model and views. The abstract view provides mechanisms for the translation of user interactions to the model domain. This is achieved through the use of the observer design pattern and features in Java to support listeners for Swing components. From other direction, the controller calls all the registered views when a model property update has occurred. The statechart view is a graph that displays the states as nodes and the transitions as vertices.

4.5.1 Software Framework Components

This section briefly explains the main classes and components of the statechart package shown in Figure 4.5.

Action and Guard Interfaces

Application-specific statechart behaviors are specified through the use of

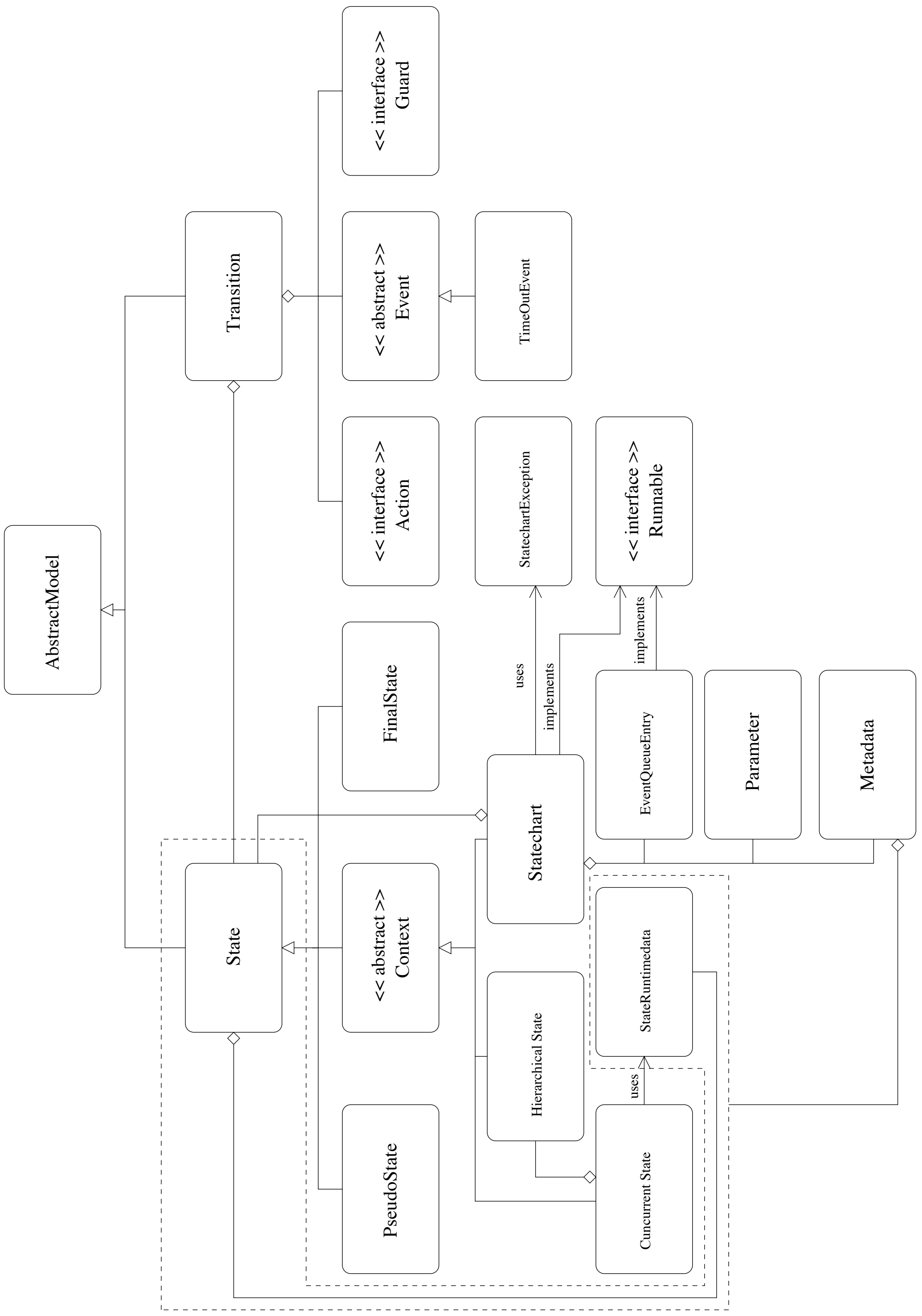


Figure 4.5: Class structure in the UML Statechart Framework modified to work with the MVC design pattern.

classes that implement the Action and Guard interface specifications, i.e.,

```
public interface Action {
   void  execute( Metadata data, Parameter param);
}

public interface Guard {
   boolean check( Metadata data, Parameter param);
}
```

The arguments of execute() and check(), data and param, represent a runtime data object and parameter for this action, respectively.

StateRuntimedata

StateRuntimedata describes runtime specific data of the statechart. The main data is the currently active state, or in general all actives when using hierarchy. For every active state a StateMetadata-Object (Stateruntimedata) is created. It stores runtime specific data for the state (e.g., the timeset since entering the state). This object is disposed of when the state becomes inactive.

Metadata

Metadata objects describe the run-time specific data of the statechart. They are created and passed to the statechart when an event is dispatched, and are deleted when the state becomes inactive. The value field in this class allows active states to have access to and modify the most recent data value.

State

This class is a concrete class of AbstractModel. Thus, It sends a property change message to the controller when the statechart enters a new state. The fragment of code:

```
public class State extends AbstractModel {
   protected Action entryAction = null;  // The entry action to execute.
   protected Action doAction    = null;  // The do action to execute.
   protected Action exitAction  = null;  // The exit action to execute.

   protected Vector<Transition> transitions = new Vector<Transition>();

   Context context = null;               // The context of this state.
   Statechart statechart = null;         // The statechart this state belongs to.
   protected String name = null;         // The name of the state.
   boolean   status, oldStatus;          // To keep the activation status

   public State( String name, Context parent, Action entryAction,
                 Action doAction, Action exitAction) throws StatechartException {

      ... details of constructor removed ...
   }
}
```

shows the essential details of the State class definition. The context of a state is simply the parent state. Support is provided for entry, do, and exit Action, as previously discussed. Each state stores a list of references to permissible transitions away from the state.

Transition

The transition class extends AbstractModel and is defined by event, guard and action objects. The fragment of code:

```
public class Transition extends AbstractModel {
   Event event   = null;      // The triggering event or 0 if no event is used.
   Guard guard   = null;      // The guard watching if the transition can trigger.
   Action action = null;      // The action to execute when the transition triggers.
   String name;               // Name of transition
   boolean status, oldStatus; // Status of transition

   // List of all states which must be deactivated/activated when triggering.

   Vector<State> deactivate = new Vector<State>();
   Vector<State>   activate = new Vector<State>();

   // Constructor methods ...
   public Transition(State start, State end) {
      init(start, end, null, null, null);
   }

   // Initialize the transition ...

   private void init(State start, State end, Event event, Guard guard, Action action) {
      this.event  = event;
      this.guard  = guard;
      this.action = action;
      Transition.calculateStateSet( start, end, deactivate, activate );
      start.addTransition(this);

      ... details of initialization removed ...
   }
}
```

shows the essential details of the Transition class definition. When a transition executes the sequence of activities is as follows:

1. Check that the event to be handled matches the event associated with the state transition.

2. Check that the guard evaluates to true.

3. Deactivate all states.

4. Execute the exit-action.

5. Execute all of the new states.

6. Fire a propertyChange event to the statechart controller.

Statechart

The statechart model employs the composite hierarchy design pattern to store the statechart components (i.e., it holds the states, exit and entry actions to those states, events and guard expressions). It operates as a pool of executable lightweight processes – in fact, at least two threads for asynchronous and timeout events

```
public class Statechart extends Context implements Runnable {
   private ExecutorService threadpool = null;
   DelayQueue<EventQueueEntry> timeoutEventQueue = new DelayQueue<EventQueueEntry>();
   HashMap<String, State> states = new HashMap<String, State>();

   ... details removed ...
}
```

The main entry point for using the statechart framework. Contains all necessary methods for delegating incoming events to the substates. When deleting the statechart all substates, actions, events, guards and transition will be deleted automatically. Methods are provided for adding events to the event queue, dispatching events, and dequeuing elements from the timeout queue and dispatches them.

4.5.2 Evaluation of Guard Expressions

This project employees jEval for the symbolic representation and evaluation of guard expressions. JEval is a library for adding math, string, boolean and functional expression parsing and evaluation in Java applications. For example, the expression:

```
    ((2 < 3) || ((1 == 1) && (3 < 3)))
```

evaluates to 1.0 (i.e., true). Support is provided for custom functions, nested functions and variables.

4.5.3 Statechart Animation

Execution of the test examples considered in this chapter is so quick that if the model and visual portions of the statechart package are perfectly synchronized, then the human eye cannot detect and follow the sequence of events. The statechart package solve this problem through use of the StatechartAnimation class and a mechanism to delay – slow down – the period of an active state or transition (current statechart status) being highlighted, while preserving the exact sequence of activities in the statechart behavior.

The animation class receives the state entrance/exit and transition execution events in a queue. It constantly checks for new element in the queue and adjusts the time count period if new item is added to the queue. Time count is used as a counter to display a transition or state hilighted. For those cases where there a no new entries in the queue, the current state/transition is highlighed for a longer time. However, when a new entry is received in the queue, the count down is performed faster.

4.6 Standalone Statechart Behavior Modeling

In this section we present details of behavior modeling with statecharts for four applications: (1) Behavior modeling for a lamp switch, (2) Behavior modeling for a countdown timer, (3) Behavior modeling for a surveillance system, and (4) Behavior modeling for lamp workspace with statechart and engineering views. The standalone lamp switch will be used in the lamp workspace example. The countdown timer illustrates control of flow that depends on evaluated guard conditions. The surveillance system illustrates the use of hierarchical states, decision nodes and timeout guard conditions. The lamp workspace example will illustrate synchronization of engineering and statechart views that update in response to user-driven events.

4.6.1 Example 1. Statechart Behavior for Lamp Switch

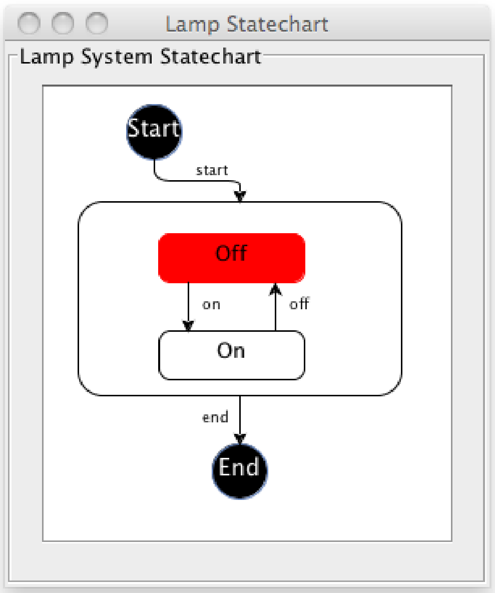
Figure 4.6 shows a sequence of screen captures for the operation of simple lamp. The lamp behavior model has two states (i.e., On and Off) two transitions (i.e., on and off), no state-dependent data, and no guard conditions. We exercise the lamp behavior through a sequence of four transition events: (1) start, (2) on, (3) off, and (4) end. In each case, the currently active states are shown in red.

The abbreviated fragment of code:

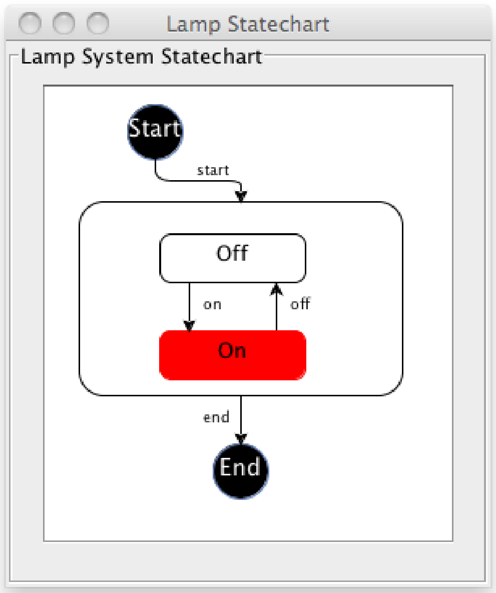
```
Output generated by the lamp statechart model ...

     [java] *** Enter Transition( start ).execute() ...
     [java] *** State.deactivate(): Start deactivated
     [java] *** Transition.execute(): start executed
```

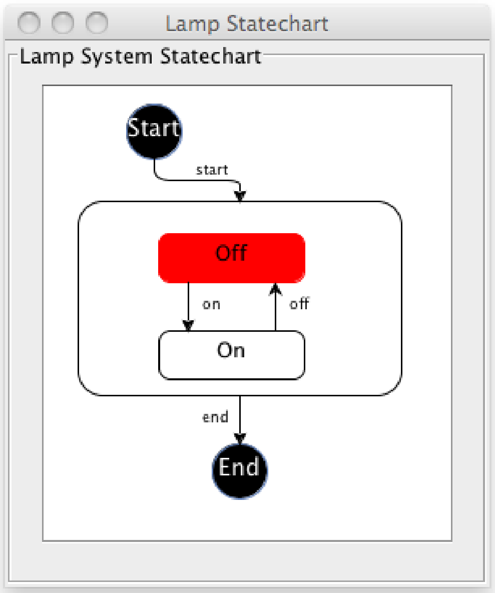
Step 1: start action.



Step 2: on action.



Step 3: off action.



Step 4: end action.

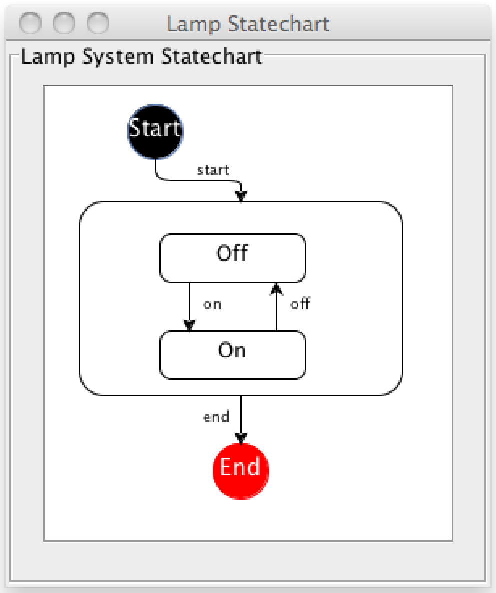


Figure 4.6: Statechart sequence for start, on, off, and end states in a simple lamp.

```
     [java] *** State.activate(): Operating activated
     [java] *** Transition.execute(): start executed
     [java] *** State.activate(): Off activated
     [java] *** Transition( on ).execute() ...
     [java] *** Transition( end ).execute() ...
     [java] *** Transition( on ).execute() ...
     [java] *** State.deactivate(): Off deactivated
     [java] *** Transition.execute(): on executed
     [java] *** State.activate(): On activated
     [java] *** Transition( off ).execute() ...
     [java] *** Transition( end ).execute() ...
     [java] *** Transition( off ).execute() ...
     [java] *** State.deactivate(): On deactivated
     [java] *** Transition.execute(): off executed
     [java] *** State.activate(): Off activated

     [java] *** Transition( on ).execute() ...
     [java] *** Transition( end ).execute() ...
     [java] *** Transition( on ).execute() ...

     [java] *** Transition( end ).execute() ...
     [java] *** State.deactivate(): Off deactivated
     [java] *** State.deactivate(): Operating deactivated
     [java] *** Transition.execute(): end executed
     [java] *** State.activate(): End activated
     [java] *** Enter State( End ).dispatch() ...
     [java] *** Leave State( End ).dispatch() return false ...

Response of the lamp statechart view ...

     [java] *** LampStatechartView.highlight() nodeId = start
     [java] *** LampStatechartView.highlight() nodeId = Operating
     [java] *** LampStatechartView.highlight() nodeId = start
     [java] *** LampStatechartView.highlight() nodeId = Off
     [java] *** LampStatechartView.highlight() nodeId = on
     [java] *** LampStatechartView.highlight() nodeId = On
     [java] *** LampStatechartView.highlight() nodeId = off
     [java] *** LampStatechartView.highlight() nodeId = Off
     [java] *** LampStatechartView.highlight() nodeId = end
     [java] *** LampStatechartView.highlight() nodeId = End
```

is partitioned into two sections: (1) Output generated by the lamp statechart model, and (2) Response of the lamp statechart view. Details on the propagation of model property changes through the abstract controller to the statechart view have been removed. At each step in the statechart execution, the statechart checks each of the local transitions to see if it is a valid exit. In this example, the statechart can transition from states On and Off to state End. Also notice that the output generated by the statechart view occurs after the model has essentially completed its execution. This happens because the model and view are separate threads of execution, and because the timer associated with the processing of events in the statechart view is deliberately slowed down for human visualization.

4.6.2 Example 2. Statechart Behavior for a Countdown Timer

The next level of complexity in statechart behavior occurs with the addition of state-dependent data and guard conditions.

To see how this works in practice, Figure 4.7 is a snapshot of a statechart behavior for a countdown timer that counts down from 5 to 1 and then expires. Support for state-dependent data and guard conditions is provided through user-defined classes that implement the Action and Guard interfaces. Figure 4.8 summarizes the essential details for the countdown timer application. Count decrements the counter by one. Reset sets the counter to zero. SetValue initialize the counter value to a user-defined value (in this case 5). Expired evaluates the guard condition (i.e., is the counter value equal to 1?).

Scripted Transition Events and System Response. The abbreviated fragment of code:

```
Thread thread = new Thread(stv,"statechartdisplay");
thread.start();
metaData = new TimerMetaData();
chart.start(metaData);

TransitionEvent te = new TransitionEvent("Count");
```

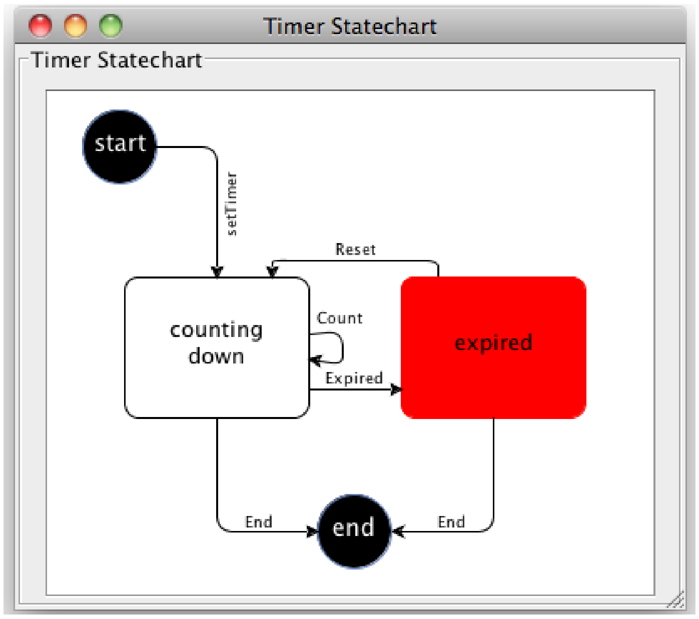


Figure 4.7: Statechart for a countdown timer.

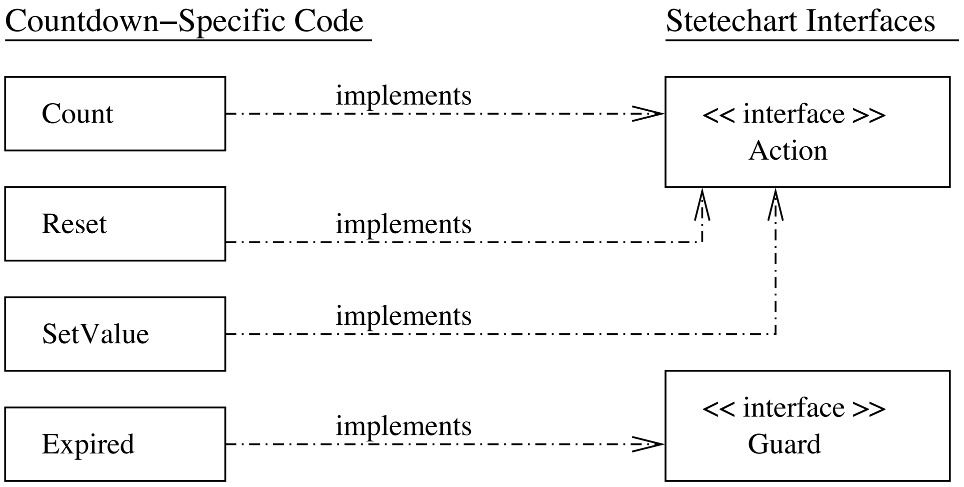


Figure 4.8: Implementation of statechart interfaces for the countdown application.

```
for ( int i = 1; i <= 5; i = i + 1 ) {
    System.out.println("*** Command: chart.dispatch( ...... )";
    chart.dispatch(metaData, te );
}
```

initiates execution of the statechart thread and then systematically dispatches five instances of the transition event "Count". The abbreviated script of code shows flow of control among the statechart, state and transition classes for the counter countdown.

```
[java] *** Enter TimerStatechart.create() ...
[java] *** Enter TimerStatechart.startStatechart() ...

... details of output removed ...

[java] ***
[java] *** Command: chart.dispatch(metaData, new TransitionEvent("Count")); ...
[java] ***
[java] *** Enter State( counting ).dispatch() ...
[java] *** Transition( Expired ).execute() ...
[java] *** Enter Expired.check(): expression: is 1 == 5?
[java] *** Leave Expired.check(): bResult = false

... details of output removed ...

[java] ***
[java] *** Command: chart.dispatch(metaData, new TransitionEvent("Count")); ...
[java] ***
[java] *** Enter State( counting ).dispatch() ...
[java] *** Transition( Expired ).execute() ...
[java] *** Enter Expired.check(): expression: is 1 == 4?
[java] *** Leave Expired.check(): bResult = false

... details of output removed ...

[java] *** Enter Expired.check(): expression: is 1 == 1?
[java] *** Leave Expired.check(): bResult = true

[java] *** TimerStatechartView.highlight() nodeId = counting
[java] *** TimerStatechartView.highlight() nodeId = Expired
[java] *** TimerStatechartView.highlight() nodeId = expired
```

Details of the modelPropertychange() calls to the statechart view and flows of propertyChange values through the abstract controller have been omitted. With each iteration, the counter value is decremented by one, and the iterates of statechart behavior will continue until the boolean expression associated with Expired evaluates to true.

4.6.3 Example 3. Statechart Behavior for a Surveillance System

The statechart package provides modeling support for decision nodes, timeout guard conditions, hierarchal, composite, and nested states.

Figure 4.9 shows, for example, a snippet of statechart behavior for the surveillance system case study described in the SysML text by Fridenthal, Moore and Steiner [18]. The top-level surveillance model has three pseudostates (i.e., start, end, and junction), four regular states (i.e., Idle, Initializing, Diagnosing, and ShuttingDown) and one hierarchal state (i.e., Operating). The second-level states (i.e., within Operating) are LoggedOn and LoggedOff. LoggedOn contains two third-level states, Normal and Alerted.

Statechart behavior emanates from sequences of events that trigger transitions connecting the states. Transitions occur through a mixture of externally and internally generated events. Examples of the former are the events startup and turnoff. The statechart will transition from Initializing to Operational when the internal attribute for initializing evaluates to true.

Scripted Transition Events and System Response. The prescribed sequence of transition events:

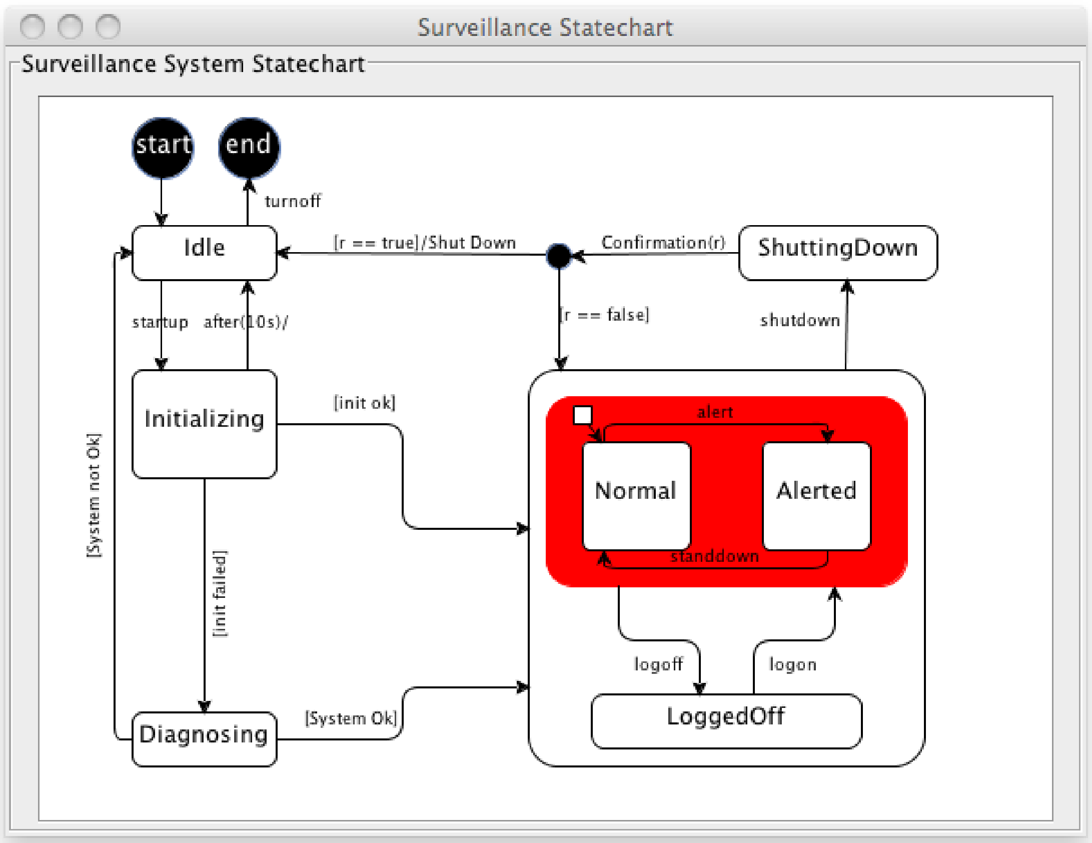


Figure 4.9: Statechart for behavior of a surveillance system. The highlighted region is the LoggedOn state.

```
  Thread thread = new Thread(stv,"statechartdisplay");
  thread.start();
  metaData = new SurveillanceMetadata();

  chart.start(metaData);
  chart.dispatch(metaData, new TransitionEvent( "startup"  ));
  chart.dispatch(metaData, new TransitionEvent( "alert"));
  chart.dispatch(metaData, new TransitionEvent( "standdown"));
  chart.dispatch(metaData, new TransitionEvent( "logoff" ));
  chart.dispatch(metaData, new TransitionEvent( "logon" ));
  chart.dispatch(metaData, new TransitionEvent( "alert"    ));
  chart.dispatch(metaData, new TransitionEvent( "standdown"));
  chart.dispatch(metaData, new TransitionEvent( "logoff" ));
  chart.dispatch(metaData, new TransitionEvent( "shutdown" ));
  chart.dispatch(metaData, new TransitionEvent( "turnoff"  ));
```

is handled by the statechart beginning with the start state. A high-level view of the statechart behavior can be summarized as follows:

```
Sequence of States ....
-------------------------------------------------------------------------
start -> Idle -> Initializing -> Operation -> ShuttingDown -> Idle -> end
=========================================================================
```

When the statechart reaches the hierarchal state, Operation, the lower-level states move through the following sequence

```
-------------------------------------------------------------------------
start2 -> Normal -> Alerted -> Normal -> LoggedOff -> Operation -> ...
=========================================================================
```

The abbreviated system response is as follows:

```
     [java] *** Transition( startidle ).execute() ...
     [java] *** In State.deactivate(): start deactivated

     [java] *** Transition.execute(): startidle executed
     [java] *** Transition.execute(): startup executed
     [java] *** Transition.execute(): initializingoperating executed
     [java] *** Transition.execute(): start2normal executed
     [java] *** Transition.execute(): start2normal executed
     [java] *** Transition.execute(): alert executed
     [java] *** Transition.execute(): logoff executed
     [java] *** Transition.execute(): logon executed
     [java] *** Transition.execute(): logoff executed
```

```
[java] *** Transition.execute(): shutdown executed
[java] *** Transition.execute(): confirmation executed
[java] *** Transition.execute(): junctionidle executed
[java] *** Transition.execute(): turnoff executed

[java] *** Enter SurveillanceViewAnimator.actionPerformed() ...

... details of output removed ...

[java] *** Enter SurveillanceViewAnimator.actionPerformed() ...
```

One the statechart has entered a state, it will systematically test each of the transition arcs to see if their guard conditions evaluate to true. Notice that some of the statechart behavior occurs without a transition event – for example, once the statechart reaches state Initializing it can transition to state Operating without any input.

4.6.4 Example 4. Lamp with Timer and Scheduled Behavior

Our first real networked application is behavior of a simple lamp having an on/off switch and a clock. The statechart is composed of two states, on and off, representing the high level functionality of a lamp. The clock operates as a separate process and send time events to the lamp controller.

Table 4.1 summarizes the system requirements and expected behavior of such system. In a more realistic application, the requirements would be refined to detailed subsystem and component requirements describing the physics and safety aspects from the electrical, or reliability engineering perspective, e.g., the maximum wattage of the lamp, range of acceptable voltage, and so forth.

System Requirements	Expected Behavior
1. The lamp shall be switched to on when time is 8:00 pm.	When the time is 8 pm, the statechart will transition to the On state if it is not already in that state.
2. The lamp shall be switched to off when time is 7:00 am.	When the time is 7 am, the statechart will transition to the Off state if it is not already in that state.
3. The user shall be able to switch the lamp off at any given time.	When the user clicks the switch button (small black box) in the lamp view, the lamp will turn off if it was on.
4. The user shall be able to switch the lamp on at any given time.	When the user clicks the switch button (small black box) in the lamp view, the lamp will turn on if it is off.

Table 4.1: Lamp System Requirements and Expected Behavior

Lamp Network Topology and Local/Global Behavior

Figure 4.10 is a schematic of the simple lamp architecture with timer and scheduled behavior. Due to simplicity of lamp prototype, a communication channel was created between the controllers in the MVC network. Controllers register their interest to be notified of events occurring throughout the network. For example, the Light controller is registered to Clock controller, and the the statechart controller is registered to Light controller.

This arrangement implies local and global types of system change in the system. Local changes are initially triggered through the interaction of a user with a view (e.g., pressing a button in the lamp view).. The controller, updates the model accordingly and notifies the view to display the recent change. But it also notifies the listener controllers in the list about a global change, and they also follow the same rule to update their own component and/or notify other components. As

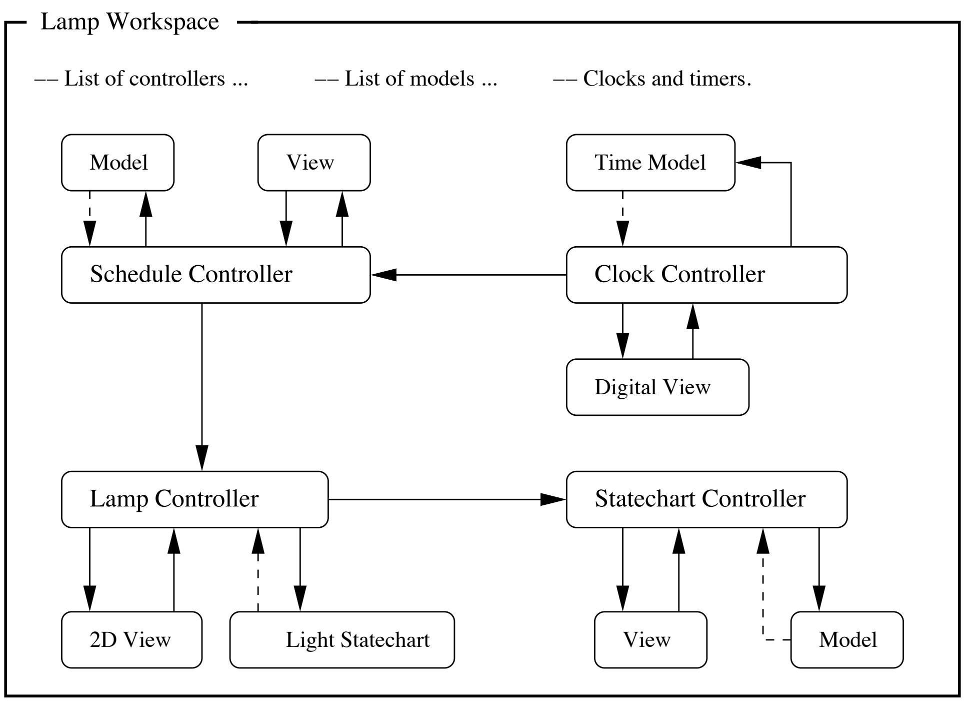


Figure 4.10: Architecture for simple lamp with timer and scheduled behavior.

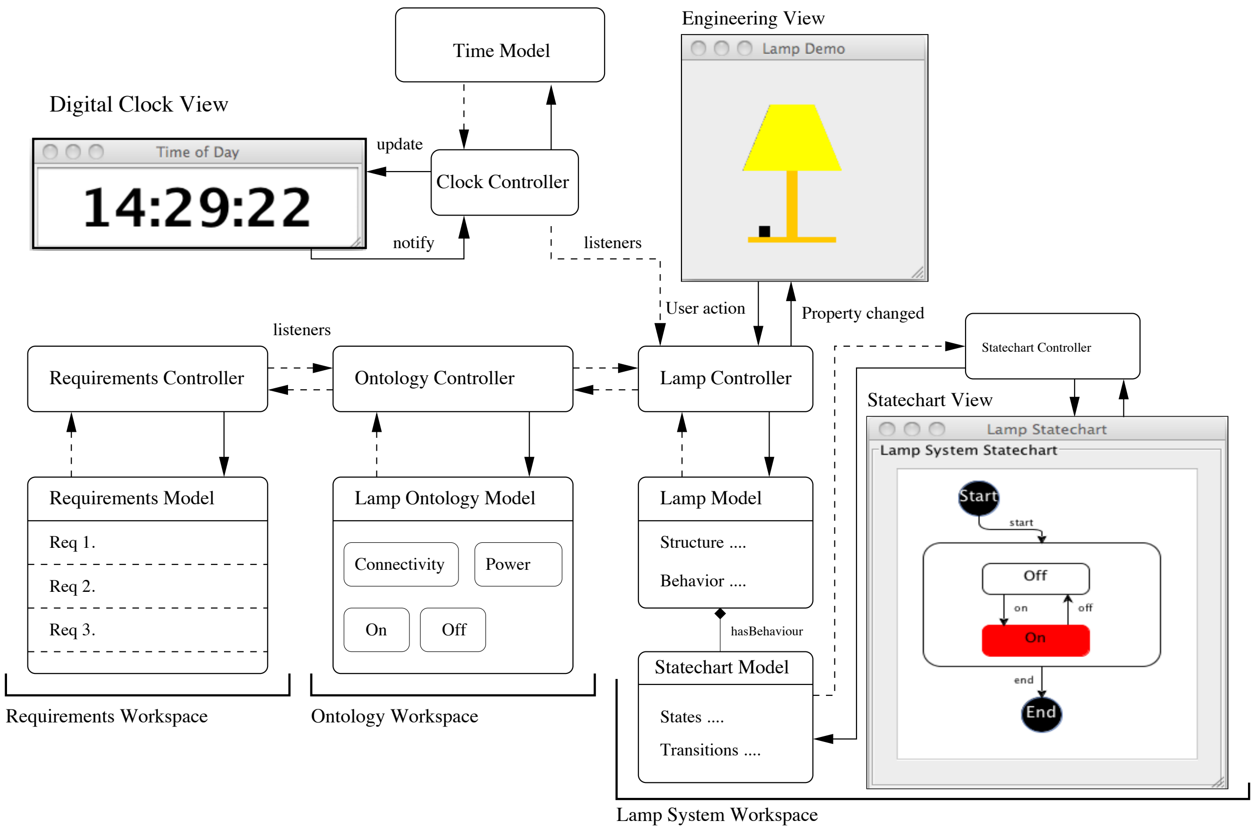


Figure 4.11: Schematic for lamp behavior prototype.

a case in point, the clock controller receives a time change event from the time model, and it passes it to the scheduler controller. For the purpose of simplicity, no (statechart) views or model are attached to scheduler controller. Finally, the engineering system, lamp, receives this global change and update the lamp status (on/off) and model behavior (statechart model). Accordingly, statechart controller calls for an update in the lamp statechart view to represent the recent status of the lamp.

Chapter 5

Rail Transit Systems Management Case Study

5.1 Case Study Problem Statement

This chapter describes a prototype implementation of ontology-enabled traceability models and mechanisms, targeted toward rail transit systems design and management. The motivating application is a simplified fragment of trains operating in the Washington D.C. Metro System.

Figure 5.1 sets the stage for descriptions of the railway transit management architecture organized into a network of communicating requirements, ontology, engineering model, and time workspaces (for details, see Chapters 3 and 4), and implemented through repeated application of the MVC design pattern. Emphasis is placed on the implementation of mechanisms (i.e., observer and MVC design patterns) that provide well maintained and scalable visualization in traceability. We will describe how these mechanisms work across different workspaces, and how changes the view or model of a workspace are propagated to all of the registered models and workspaces. We exercise the behavioral aspects of the transit system through train behaviors that are modeled and visualized using statecharts. Finally, we implement real-time rule checking to monitor and control train behavior to satisfy safety requirements.

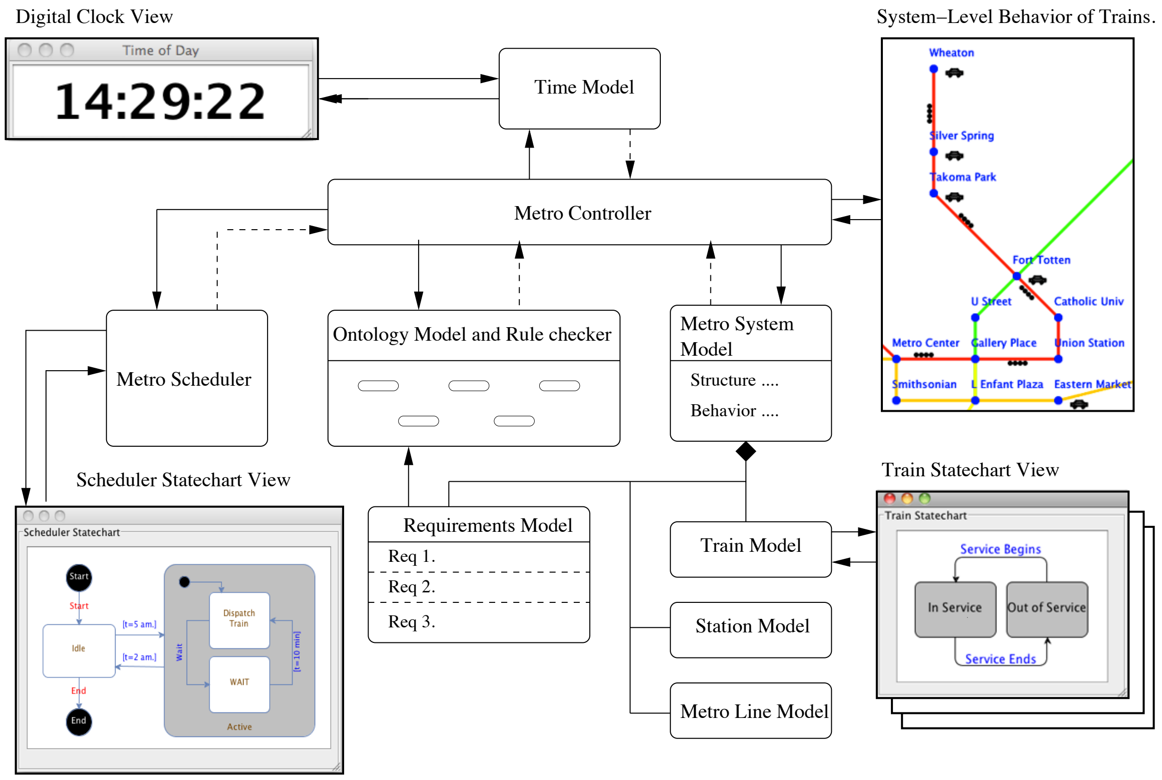


Figure 5.1: Railway Management System Architecture with Model View Controller Approach.

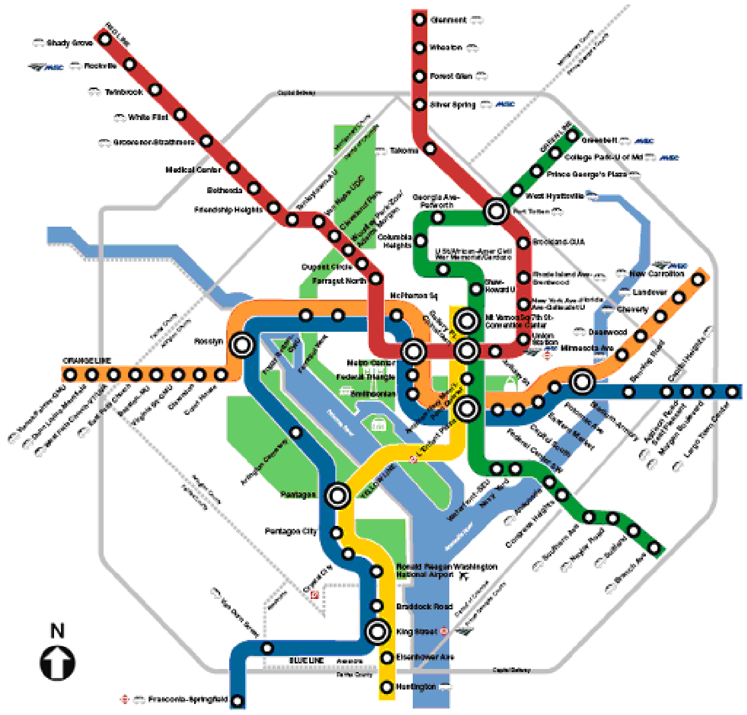


Figure 5.2: Map of the Washington D.C. Metro System

5.2 The Washington D.C. Metro System

The Washington D.C. Metro System is the second largest rail transit system in the United States. It serves a population of 3.5 million people with more than 200 million passenger rides per year. As of 2006, there were 86 metro stations in service and 106.3 miles of track.

Figure 5.2 shows the map of the Metro System. The five Metro System lines cover the District of Columbia; the suburban Maryland counties of Montgomery and Prince George's; the Northern Virginia counties of Arlington, Fairfax and Loudoun; and the Virginia cities of Alexandria, Fairfax and Falls Church [47].

5.3 Framework for Rail Transit Systems Design and Management

Modern railway systems are a complex intermingling of traditional infrastructure with electronics and telematics (i.e., GIS and GPS) [39]. To keep the complexity of design concerns in check, railway system design procedures strive to separate the underlying infrastructure (e.g., track profile and layout) from operational (e.g., schedule and capacity) and control (e.g., sequencing of switching and crossings) concerns. In systems engineering terminology the track infrastructure and railway vehicles define the systems structure. System behavior is defined by the operations and control. The first and most important priority is to ensure that all operations are completely safe. Then with safety concerns satisfied, schedules, capacity and switching operations are designed to maximize available capacity and

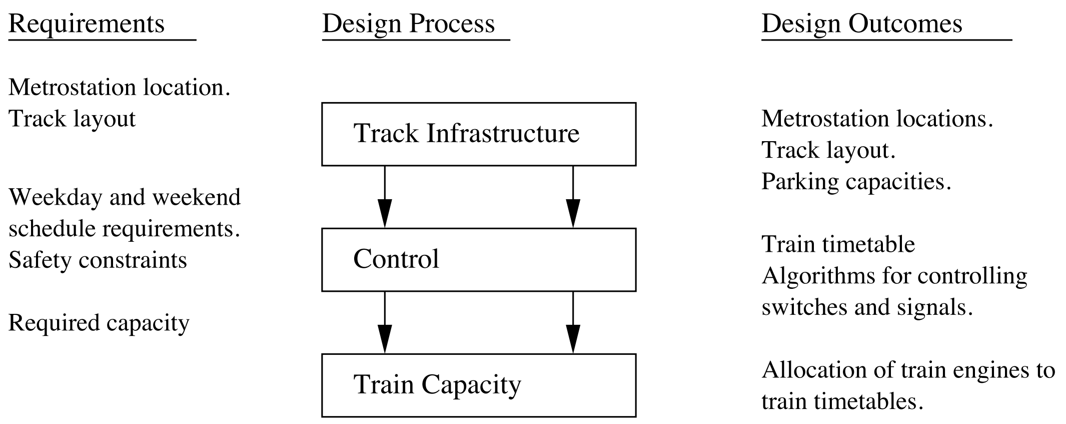


Figure 5.3: Flowdown of requirements and design outcomes in a top-down down development process.

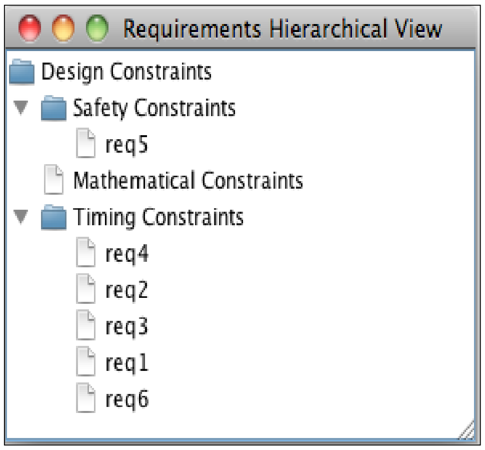
minimize delays, subject to cost and performance constraints.

Figure 5.3 shows the sequence of developments and flowdown of requirements in a (simplified) top-down development process. The development process begins with decisions on track infrastructure (e.g., positioning of metro stations; track layout; transfer stations) which deals with static and structural aspect of the system. Moreover, issues of scheduling (e.g., weekday and weekend departure and arrival times for trains) and train control (e.g., routing trains through railway stations) have to be addressed. This phase defines system dynamics and behavior. The primary purpose of a railway control system is to prevent events from happening that could lead to an unsafe system state. Generation of a “train timetable” is often complicated by highly utilized and intertwined railway networks with many connections between trains [49]. Since many sections of the track will operate as a shared resource (meaning that different trains will use the same section of track at different times), strategies for scheduling and control must guarantee that all safety constraints are met. The most straightforward approach to achieving this objective is to implement centralized control algorithms that: (1) have access to the global state of the system, and (2) verify correctness of system operations through formal analysis. Finally, decisions are made on train selection to satisfy requirements on scheduling and passenger capacity. A complete study would also generate a mix of best-engine allocations for a number of fleet alternatives [16].

5.4 Railway System Architecture and Workspaces

In the best practice of ontology-enabled traceability architectures, individual workspaces are represented as a MVC node with a controller and number of attached registered models and views. For the purposes of simplifying the railway system implementation, a shared controller (metro controller) is responsible for setting up the communication channel between different models and views across the network (i.e., the different models and views are connected to one controller). Figure 5.1 shows the architectural layout of this system. MetroController serves as the main communication hub for time, ontology and engineering workspaces. And since MetroController extends AbstractController (for details see Chapter 3 and 4) it can have a number of registered views and models.

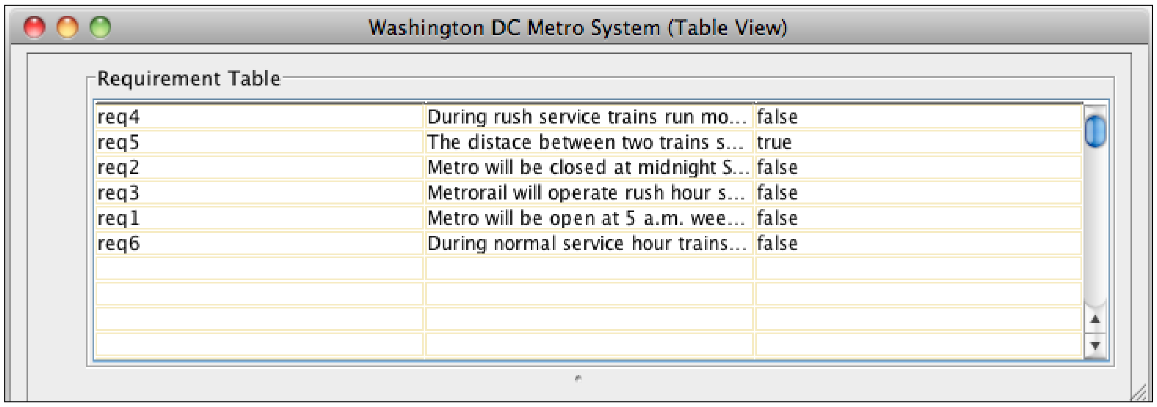
Requirements Workspace. This workspace stores information about the safety (i.e., maximum capacity and operational spacing of trains), regulations (i.e., location of the stations and parking spots), scheduling (i.e., train timetables) and design requirements (i.e., size and specification of track, train car specification).



Requirements Hierarchical View

- Design Constraints
- Safety Constraints
  - req5
- Mathematical Constraints
- Timing Constraints
  - req4
  - req2
  - req3
  - req1
  - req6

Figure 5.4: Tree view of requirements.



Washington DC Metro System (Table View)

Requirement Table

req4	During rush service trains run mo...	false
req5	The distace between two trains s...	true
req2	Metro will be closed at midnight S...	false
req3	Metrorail will operate rush hour s...	false
req1	Metro will be open at 5 a.m. wee...	false
req6	During normal service hour trains...	false

Figure 5.5: Tabular view of requirements.

Procedurally, the requirements are stored in an XML file format, read and converted into a RequirementModel (i.e., a hashtable collection), and are then adapted to a variety of visualization formats. Figure 5.4, shows, for example, a tree view of the requirements. In a tree view, emphasizes is placed on categories (e.g., safety rules and design rules) and hierarchal organization. Figure 5.5 shows that the requirements model can also be adapted and displayed in a tabular view. Now, emphasis is placed on describing the textual details of each requirement and their operational status. Status "true" refers to a situation when a requirement is active or satisfied.

In both view representations, only one model (i.e., RequirementModel) is used, and this is how multiple view points are addressed in this framework. We will soon see how the separate view are connected to one controller and how traceability visually shows the connection between them (i.e., when one requirement is true in the table view, the same requirement will also be selected in the tree view. Moreover, we will see that requirements (or derived constraints) are correspondent with rules maintained in the ontology workspace. While the metro system needs to satisfy safety, behavioral, and non-functional engineering requirements, among others, our discussion here is restricted to behavioral requirements for the trains. In this test case the dynamics of the trains, the safety rule in terms of minimum distance of two following trains were examined. This requirement become active or not in a real time manner as trains move along the tracks.

Safety Requirements. These requirements are with respect to timing and scheduling of the trains. As a case in point, the distance between the back and the front of two following trains, can not be less than a quantity. If it is violated, the train following on the back will be stopped until the one in the front moves and satisfy the minimum distance.

Ontology Workspace. Figure 5.6 builds upon Figures 1.12 through 1.15, and displays a mockup of a metro system ontology that takes into account the design concerns of transportation and mathematical analysis stakeholders. A mathematician will view the metro network as a graph of nodes and edges. Many algorithms

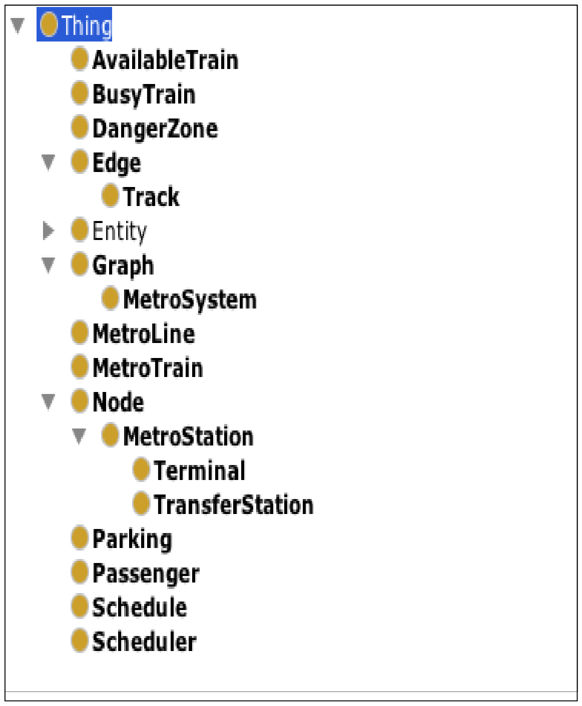
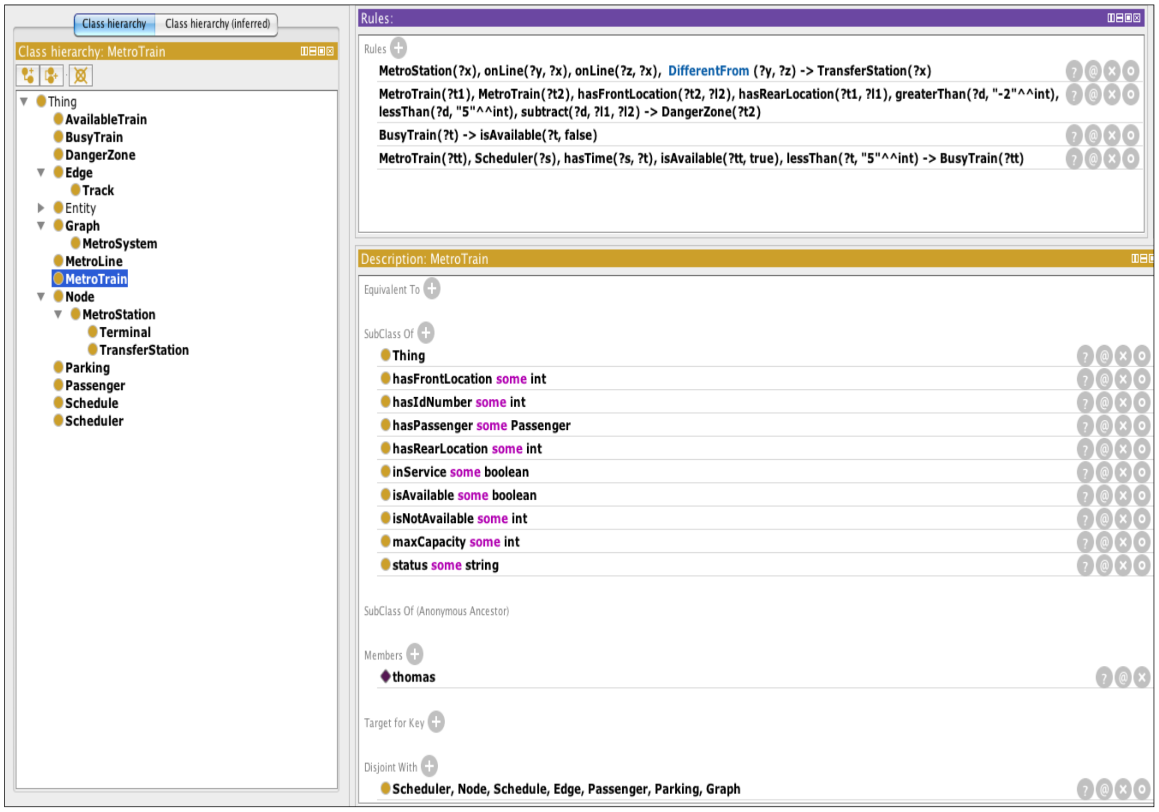

Thing
AvailableTrain
BusyTrain
DangerZone
Edge
Track
Entity
Graph
MetroSystem
MetroLine
MetroTrain
Node
MetroStation
Terminal
TransferStation
Parking
Passenger
Schedule
Scheduler

Figure 5.6: Mockup of a Metro System Ontology

now exist to analyze the properties of a graph. Transportation engineers look at the same network in terms of metro stations (nodes) and tracks (edges) and groups of track elements are organized into lines (e.g., the green line, the red line) to facilitate passenger navigation from a source to a destination. The collection of concepts in each of these perspectives will have well defined purposes and will be constrained by design rules and constraints.

The Railway system ontology is created in Protege and stored in an OWL file. Figure 5.7 is a schematic the metro ontology and and SWRL rule repository as modeled in the Protege framework. And Figure 5.8 shows the relationship between the ontology model and ontology classes and properties. To provide application programs with the ability to search, query and modify the ontology data, the Jena framework creates a one-to-one mapping from the ontology classes to Java attributes.



Class hierarchy | Class hierarchy (inferred)

Class hierarchy: MetroTrain

```
▼ Thing
    AvailableTrain
    BusyTrain
    DangerZone
  ▼ Edge
      Track
  ▶ Entity
  ▼ Graph
      MetroSystem
    MetroLine
    MetroTrain
  ▼ Node
    ▼ MetroStation
        Terminal
        TransferStation
    Parking
    Passenger
    Schedule
    Scheduler
```

Rules:

Rules

```
MetroStation(?x), onLine(?y, ?x), onLine(?z, ?x),  DifferentFrom (?y, ?z) -> TransferStation(?x)
MetroTrain(?t1), MetroTrain(?t2), hasFrontLocation(?t2, ?l2), hasRearLocation(?t1, ?l1), greaterThan(?d, "-2"^^int),
lessThan(?d, "5"^^int), subtract(?d, ?l1, ?l2) -> DangerZone(?t2)
BusyTrain(?t) -> isAvailable(?t, false)
MetroTrain(?tt), Scheduler(?s), hasTime(?s, ?t), isAvailable(?tt, true), lessThan(?t, "5"^^int) -> BusyTrain(?tt)
```

Description: MetroTrain

Equivalent To

SubClass Of
- Thing
- hasFrontLocation some int
- hasIdNumber some int
- hasPassenger some Passenger
- hasRearLocation some int
- inService some boolean
- isAvailable some boolean
- isNotAvailable some int
- maxCapacity some int
- status some string

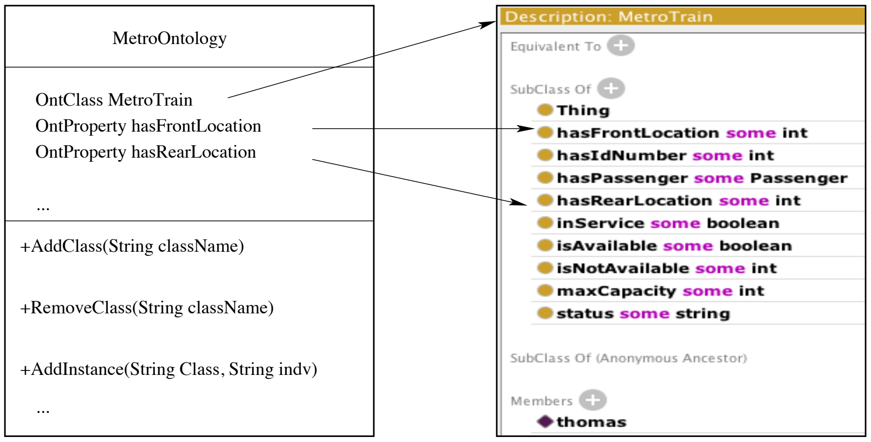
SubClass Of (Anonymous Ancestor)

Members
- thomas

Target for Key

Disjoint With
- Scheduler, Node, Schedule, Edge, Passenger, Parking, Graph

Figure 5.7: Schematic of Metro Ontology and SWRL rule repository in Protege Framework



Safety Rule:

MetroTrain(?t1), MetroTrain(?t2), hasFronLocation(?t2,?l2),hasRearLocation(?t1,?l1), subtract(?d,?l1,?l2), lessThan(?d,"2")–> DangerZone(?t2)

Figure 5.8: Relationship between the Ontology Model and Ontology Classes and Properties

For the railway case study the ontology model is a Java class that stores the ontology classes (e.g., train, station) and their associated properties (e.g., hasLocation, hasCapacity) as attributes. End-users are provided with the means to not only modify the ontology by adding new individuals (e.g., a new train) and/or remove existing individuals (e.g., removing a train from operations) from the ontology model, but property values such as the location of a train can be updated at each time-step of a simulation. A reasoner attached to the Jena framework will perform rule checking in response to changes in the ontology model and, if required, in real-time. The reasoner is used to evaluate safety requirements associated with the minimum allowable spacing between trains – when the rule fails, the train at the back will be labeled as entering a the danger zone, and a message will be sent to the controller to stop the train.

Engineering Workspace. The engineering workspace models the system structure and system behavior and, as such, can be visualized with multiple abstractions (e.g., CAD-like views of the system layout; metro map views that highlight connectivity but simplify geometry; statechart views of train behavior). Due to the geometric nature of the transportation system layout, nested data structures and collections store models of the stations, trains and tracks. Each object type will have an assortment of attribute values (e.g., station location, train capacity, source and destination of track, line color, station location, train capacity size and location).

Figure 5.9 is a top-level view for part of the engineering model. The metro system model holds a collection (i.e., HashSet) of different components or subsys-

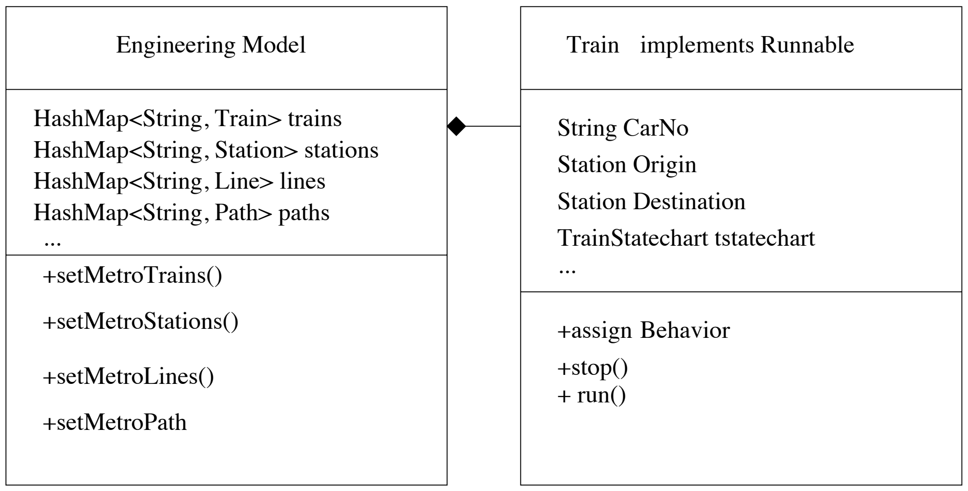

Engineering Model
HashMap<String, Train> trains
HashMap<String, Station> stations
HashMap<String, Line> lines
HashMap<String, Path> paths
...
+setMetroTrains()
+setMetroStations()
+setMetroLines()
+setMetroPath
Train implements Runnable
String CarNo
Station Origin
Station Destination
TrainStatechart tstatechart
...
+assign Behavior
+stop()
+ run()

Figure 5.9: A part of the engineering model class diagram.

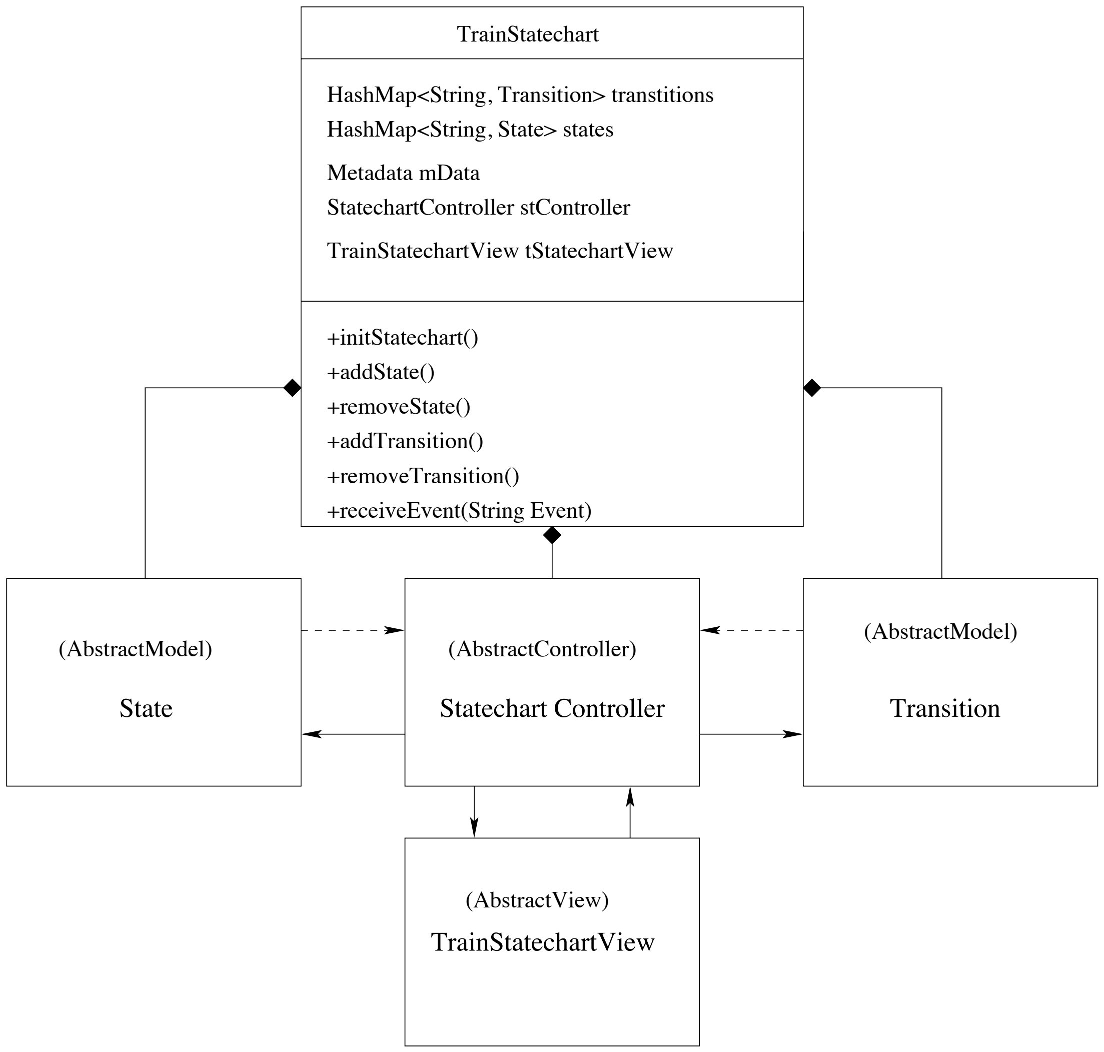


Figure 5.10: MVC design pattern for implementing the train statechart.

tems. For those components that also have assigned behavior, information on the associated statechart – states, transitions, guard conditions – is attached to the component and operates as a separate thread of execution. Figure 5.10 shows the associated train statechart class diagram that contains states and transitions representing that statechart and creates MVC between transitions, states, statechart controller and statechart view.

High-level train functionality corresponds to a collection of methods for simple operations, e.g., run(), stop(). The details of how and when a train has to function depends on the timing rules and are stored in the rule set. When a statechart model enters a new state or completes a transition, the associated controller will notify all of the registers views that an update to the statechart view is required. Figures 5.11 and 5.12 show the two train states employed in this simplified model – state "run" is connected to "stop" with transition "Stop()" and guard condition "[safety]". When an active state changes or a transition occurs the views will updates themselves in response to a call from the controller.

Time Workspace. The time workspace provides time to the scheduler to notify the trains when to run, stop, park, end/begin a service. Changes in the time model are due to the timer event (every second, minute, and hour), which triggers the controller to update the clock view. The time model contains a timer that notifies the controller in case of a second, minute and hour change. The time display view is implemented as a digital view, but analog views are also possible.

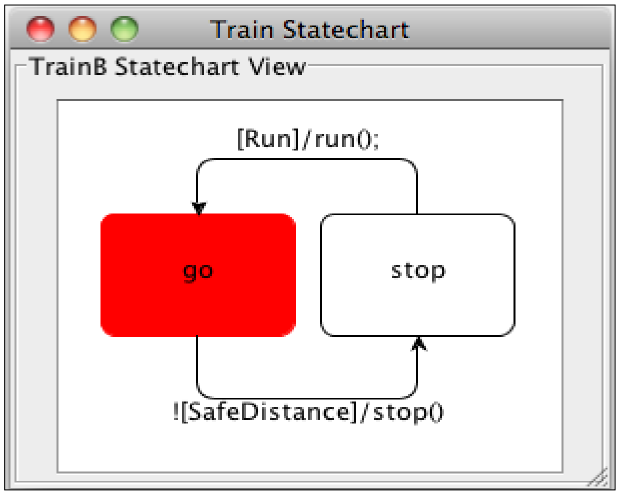


Figure 5.11: State run is active in statechart view.

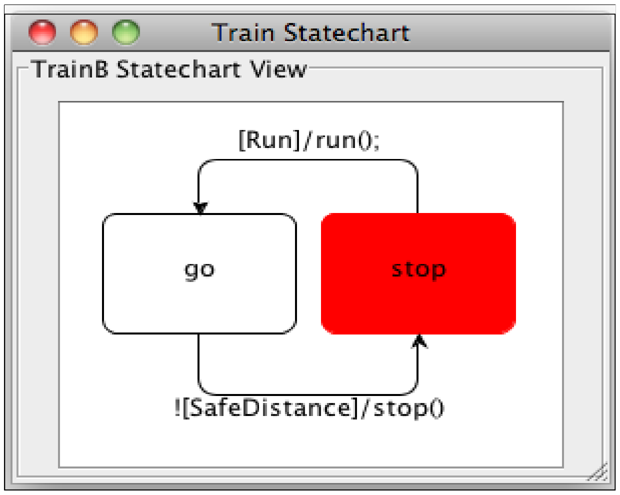


Figure 5.12: State stop is active in statechartview.

5.5 Traceability

Recent research in creating, maintaining, and using traceability mechanisms has identified the following challenges [22]:

- Exploring topics related to automating the traceability process
- Developing strategies for achieving cost-effective traceability
- Visualizing traceability and advancing its effectiveness for end users
- Developing traceability practices that apply across a wide range of domains

For the purposes of this project, the term traceability refers to an ability to link system requirements forward to design concept models and rules, and to engineering system components and simulations. Starting from the engineering workspace, traceability also implies that an event in the engineering workspace can be traced back to dependencies in the ontology and requirements workspaces. These mechanisms provide end-users with the ability to conduct impact analyses and requirements validation. Requirements traceability can demonstrate that a system meets the different stakeholder viewpoints, stated requirements, and complies with a sets of regulations.

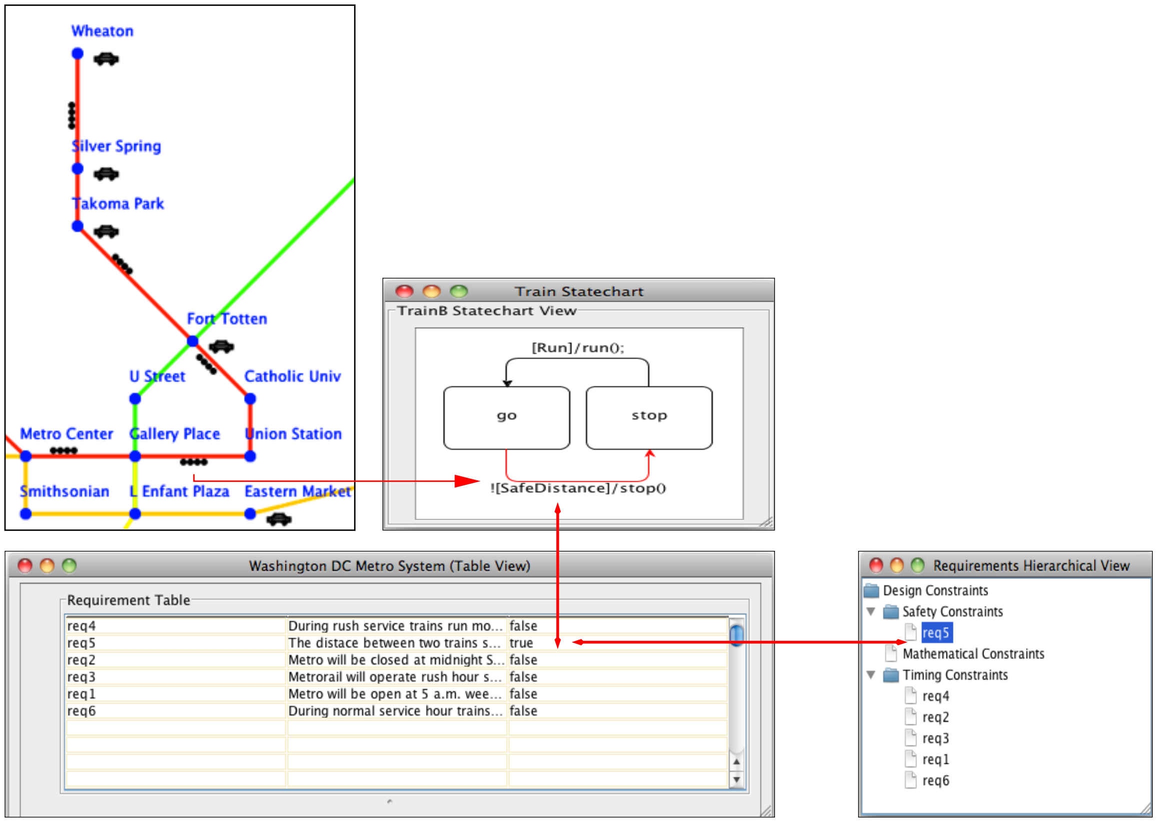
This case study focuses on the capture of two types of traceability mechanism: (1) Traceability within a single workspace (e.g., selecting on requirement tree view will select the corresponding requirement in table view), and (2) Traceability across workspaces (e.g., requirements and engineering) which shows how a change in one workspace is related to a change in another workspace.

5.6 Traceability of Requirements to Statechart Behavior Modeling

The traceability of requirements to statechart behavior modeling occurs across the requirements, ontology and engineering workspaces, and will be visualized through a variety of views. Their quantitative evaluation involves the attribute values of states and guard condition expressions in the transitions of finite state machines.

Consider, for example, the requirement: The distance between two trains shall not be less than 5 meters. The associated traceability mechanism may be activated in a number of ways. First, a user positions the cursor over a train spacing requirement in either a tabular or tree view. The propagation of traceability dependencies will result in the corresponding guard condition being highlighted in the statechart view. A second possibility occurs when simulations are setup with (different) train speeds that will lead to a spacing violation. Pathways of traceability can also be followed from the system behavior views back to the requirements, as well as from a statechart view to engineering view. Suppose, for example, that a user mouses over a statechart describing train behavior the propagation of traceability relationships will result in the owner train being highlighted in the engineering view.

User Initiated Changes. To see how this works in practice, let us suppose that



req4	During rush service trains run mo...	false
req5	The distace between two trains s...	true
req2	Metro will be closed at midnight S...	false
req3	Metrorail will operate rush hour s...	false
req1	Metro will be open at 5 a.m. wee...	false
req6	During normal service hour trains...	false

Figure 5.13: Schematic of visual traceability across workspaces.

a user positions the cursor/mouse over a track, train or a station. The visual representation will indicate selection by becoming highlighted. Behind the scenes, the mouseMoved() method attached to a view will cause the engineering controller to update its model – the controller will search through collections (trains, tracks and stations) to retrieve the model based on the coordinates of the selection and set it is status to true or active. The change of the status in the model side will trigger the controller to update the view side based on the recent changes from the model. The controller will propagate a change-event to each of the registered views, which in turn, will decide whether or not they wish to respond (e.g., by also highlighing the selected item).

System Dynamic Changes. System dynamic changes are triggered by changes in the internal state or behavior of an engineering component. During simulation of the rail transit system, the train velocities and locations are constantly being updated. When a change in behavior occurs due to evaluation of a guard condition, a traceability pathway will also exist back to a source requirement. Figure 5.13 shows, for example, a scenario where a transition from “go” to “stop” is active in the statechart view; the corresponding source requirement in both the tree and table views are active as well.

Chapter 6

Conclusions and Future Work

6.1 Summary of Work and Contributions

The most important contribution of this thesis is development of a software framework to support ontology-enabled traceability from requirements to ontologies and elements of system structure and behavior. The framework design is based upon a multitude of software design patterns which work together to provide mechanisms for modeling systems as networks of controllers, multiple models (i.e., for requirements, ontologies, and engineering developments), and multiple visualizations (e.g., plan views, tree views, table views).

In state-of-the-art approaches to traceability, requirements are mapped directly to engineering objects without making any reference to how or why the mapping makes sense. The proposed model improves upon this practice by providing engineers with the mechanisms to: (1) Understand why different components of the system exist and what requirement(s) led to that design, and (2) Obtain information on cause-and-effect relationships among requirements, design concepts, and models in the engineering domain. Previous contributions to this problem area at the University of Maryland [2] have focused on traceability of requirements to elements of the system structure. This project has taken the next step and provides mechanisms for evaluating requirements associated with fragments of component and system behavior. This is achieved through the tracing of requirements to guard expressions in statecharts and evaluation of relationships among components (e.g., spacing between trains).

Although statechart diagrams in UML and SysMl provide information about the different states and transitions in a system, they do not represent real-time behavior of the system components. This capability has been implemented by combining a SourceForge statechart package (on the modeling side) with a the Java-based graph visualization toolkit JGraphX. The modeling and visualization aspects of statecharts are synchronized through use of the MVC software design pattern. The MVC statechart package has been tested on a number of standalone applications. The key benefits of this capability are visual feedback on the current operating state(s) of a system, and enhanced understanding for how and why a system may have failed.

In state-of-the-art approaches to systems engineering with SysML, parametric relationships describe dependencies among constraints, and constraints verify requirements. Generally speaking, a key weakness of SysML is a lack of semantic support to represent and reason with data and knowledge in engineering domains. To address this problem, in this project Semantic Web technologies (e.g., OWL and SWRL) have been employed for the capture of rules and rule checking at the design stage and during the operation of a system. With this approach it is convenient to think of an ontology as a block (or class diagram) and the rules as the constraints that are checked and verified with a reasoner. The result is improved support for system verification. In the railway management system case study, the Protege framework was used to create a metro ontology. A safety requirement rule: "The distance between two trains shall be less than 2 m" was created and stored in the SWRL rule repository. During a simulation of train behavior, data on the train locations are streamed to the ontology in real-time. The rule checker evaluates the safety requirement and takes action (i.e., stops a train) when this rule is not satisfied.

6.2 Future Work

Our program of research to understand the role that software patterns, ontology technologies, and mixtures of graph and tree visualization can play in the implementation of ontology-enabled traceability mechanisms is still in its infancy. Further work is needed to create software capability for the vision conveyed in Figures 1.16 and 1.17, for the modeling and display of requirements and ontologies as graphs, and to incorporate real-time measurement of performance into the system assessment. The latter can occur in a number of ways. Within an engineering workspace, for example, it is sometimes possible to cast engineering models as graph problems and use well-devloped algorithms for operational planning (e.g., shortest path for train travel between 2 points). This information can to perform trade-space analyses for performance versus economics, an so forth. To date, the train scheduler has been very rudimentary. Improvements will include the ability to generate timetables for each train based on models of passenger demand. We plan to replace SWRL with the Jena Framework, a Java framework for the development of Semantic Web applications. We also wish to move toward a model of one controller per workspace and, thus, networks of controllers connected together. This capability will give us the freedom to add and remove workspaces.

When the Washington D.C. Metro System example is complete (i.e., including schedules, timetables, requirements, ontologies and animated train behaviors) our plans are to move onto energy efficient buildings.

Bibliography

[1] Alexander C. *A Pattern Language: Towns, Buildings and Construction*. Oxford Press, 1977.

[2] Austin M.A. and Wojcik C.E. Methodology and System for Ontology-Enabled Traceability: Pilot Application to Design and Management of the Washington D.C. Metro System. *ISR Technical Report 2010-21*, December 2010.

[3] Austin M.A. and Wojcik C.E. Ontology-Enabled Traceability Mechanisms. In *20th Annual International Symposium of The International Council on Systems Engineering (INCOSE 2012)*, Chicago, USA, July 12-15 2012.

[4] Balasubramaniam R., Jarke M.,. Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, 27(1), January 2001.

[5] Bechhofer S., van Harmelen F., Hendler J., Horrocks I., McGuinness D.L., Patel-Schneider P.F.., and Stein L.A. OWL web ontology language reference W3C Recommendation. 2004.

[6] Berners-Lee T., Hendler J., and Lassila O. The Semantic Web. *Scientific American*, 2001.

[7] Berners-Lee T., Hendler J., Lassa O. The Semantic Web. *Scientific American*, pages 35–43, May 2001.

[8] Bever C.E. (now Wojcik C.E.). Requirement-to-UML-to-Engineering Model and Drawing Mappings, 2006. M.S. Thesis in Systems Engineering, Institute for Systems Research, University of Maryland, College Park, MD 20742.

[9] CORE. See http://www.vitechcorp.com/productline.html. 2009.

[10] Davis A.M. *Software Requirements: Objects, Functions, and States* . Prentice Hall, 1993.

[11] Delgoshaei P., and Austin M.A. Software Design Patterns for Ontology-Enabled Traceability. In *Conference on Systems Engineering Research (CSER 2011)*, Redondo Beach, Los Angeles, April 15-16 2011.

[12] Delgoshaei P., and Austin M.A. Software Patterns for Traceability of Requirements to Finite-State Machine Behavior. In *Tenth Annual Conference on Systems Engineering Research (CSER 2012)*, St. Louis, Missouri, March 19-22 2012.

[13] Delgoshaei P., and Austin M.A. Software Patterns for Traceability of Requirements to Finite-State Machine Behavior: Application to Rail Transit Systems Design and Management. In *22nd Annual International Symposium of The International Council on Systems Engineering (INCOSE 2012)*, Rome, Italy, 2012.

[14] Dynamic Object Oriented Requirements System (DOORS). See http://www.telelogic.com/products/doorsers/doors/. 2009.

[15] Eeles P. and Cripps P. *The Process of Software Architecting*. Addison-Wesley, 2010.

[16] Florian M., Bushell G., Ferland J., Guerin G., and Nastansky L. The Engine Scheduling Problem in Railway Network. *INFOR*, 14(2), June 1976.

[17] Fowler M. See http://martinfowler.com/eaaDev/uiArchs.html.

[18] Fridenthal S., Moore A., and Steiner R. *A Practical Guide to SysML*. MK-OMG, 2008.

[19] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[20] Geroimenko V., and Chen C. (Eds). *Visualizing the Semantic Web: XML-based Internet and Information Visualization*. Springer, 2003.

[21] Gomez-Perez A., Fernandez-Lopez M., and Corcho O. *Ontological Engineering*. Springer, 2004.

[22] Gote O., Cleland-Huang J., Zisman A. . *Software and Systems Traceability* . Springer, 2012.

[23] Grossman, Ornit. Harel, David. On the Algorithmics of Higraphs. Technical report, Rehovot, Israel, 1997.

[24] Harel D. Statecharts: A Visual Formalism for Complex Systems. *Science of. Computer. Programming*, 8:231–274, 1987.

[25] Harel D. On Visual Formalisms. *Communications of the ACM*, 31:514–530, 1988.

[26] Hendler J. Agents and the Semantic Web. *IEEE Intelligent Systems*, pages 30–37, March/April 2001.

[27] Holsapple C., and Joshi K. D. *A Knowledge Management Ontology*. Springer-Verlag, NY, 2003.

[28] Horrocks A., Parsia B., Schneider P., and Hendler J. Semantic Web Architecture: Stack or Two Towers? *Bell Labs Technical Journal*, 2005.

[29] 2009. IBM Telelogic SLATE: See http://www.craiglarman.com.

[30] IEEE 1471, Recommended Practice for Architectural Description of Software Intensive Systems, IEEE Std 1471-2000. For details, see http://standards.ieee.org/reading/ieee/ std_public/ description/se/1471-2000_desc.html (Accessed April 17, 2010), 2000.

[31] Jacobson I., Booch G. *Unified Software Development Process. Reading.* Reading, Mass.: Addison Wesley., 1999.

[32] Liang V.C., and Paredis C.J.J. A Port Ontology for Conceptual Design of Systems. *Transaction of the ASME*, 4, September 2004.

[33] Mahmoud Q.H. Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications, 2005. Sun Microsystems. For more information, see http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html (Accessed, March 10, 2008).

[34] Maier, M.W. Reconciling System and Software Architectures. *The Aerospace Coorporation*, 1998.

[35] Muller D. Requirements Engineering Knowledge Management based on STEP AP233. 2003.

[36] GUI Architectures, See http://www.oracle.com/technetwork/articles/javase/index-142890.html.

[37] Oliver D. AP233 - INCOSE Status Report. *INCOSE INSIGHT*, 5(3), October 2002.

[38] OracleMVC. See http://www.oracle.com/technetwork/articles/javase/index-142890.html.

[39] Profillidis V.A. *Railway Engineering: Second Edition.* Ashgate, 2000.

[40] Protege see http://protege.stanford.edu/.

[41] Rudolf G. Some Guidelines For Deciding Whether To Use A Rules Engine, 2003. Sandia National Labs. For more information see http://herzberg.ca.sandia.gov/guidelines.shtml (Accessed, March 10, 2008).

[42] Staab S., and Maedche A. Ontology Engineering beyond the Modeling of Concepts and Relations. In Benjamins R.V., Gomez-Perez A., Uschold M., editor, *Proceedings of 14th European Conference on Artificial Intelligence: Workshop on Applications of Ontologies and Problem-Solving Methods*, 2000.

[43] Stelting S. and Maassen O. *Applied Java Patterns.* SUN Microsystems Press, Prentice-Hall, 2002.

[44] Tidwell D. *XSLT.* O'Reilly and Associates, Sebastopol, California, 2001.

[45] UML Statechart Framework for Java. For details, see https://github.com/klangfarbe/UML-Statechart-Framework-for-Java, (Accessed 2011).

[46] w3 See http://www.w3.org/TR/owl-features/.

[47] 2006. WMATA Facts. See http://www.wmata.com/about/metrofacts.pdf. Accessed, December 2006.

[48] XML Stylesheet Transformation Language (XSLT). See http://www.w3.org/Style/XSL. 2002.

[49] Zwaneveld P.J., Kroon L.G., Stan P.M., and van Hoesel S.P.M. Theory and Methodology: Routing Trains through a Railway Station based on a Node Packing Model. *European Journal of Operations Research*, 128:14–33, 2001.