Software Patterns for Traceability of Requirements to Finite State Machine Behavior: Application to Rail Transit Systems Design and Management

Parastoo Delgoshaei  
Institute for Systems Research  
University of Maryland,  
College Park, MD 20742, USA.

Mark Austin  
Institute for Systems Research  
University of Maryland,  
College Park, MD 20742, USA.

Abstract. This paper describes a software design pattern (e.g., hierarchical graphs of model-view-controllers) and models of visualization for ontology-enabled traceability, where requirements are traced to elements of finite-state machine behavior (e.g., actions, states, transitions and guard conditions). These ontology-enabled traceability mechanisms will play an important role in the team-based development of network-enabled platforms for analysis, design, and early validation and verification of information-age engineering systems. The application of these concepts is illustrated through the study of a simplified rail transit system.

Introduction

Problem Statement. There is a growing class of engineering 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 for assessing the correctness of system operations, estimating system performance, and understanding trade spaces involving competing design criteria. To address these concerns, during the past few years we have developed methodologies and tools for ontology-enabled traceability; that is, traceability mechanisms where requirements are connected to models of engineering entities by threading the traceability connection through one or more ontologies. In our proof-of-concept work (Austin and Wojcik 2010) the engineering entities were elements of system structure (e.g., tracks, lines and stations) in a rail transportation system. But, of course, real engineering systems also have behaviors (e.g., schedulers and train behaviors).

Scope and Objectives. To address this extension, this paper reports on research to understand the role that software patterns (e.g., model-view-controller, mediator, observer, adapter), ontology technologies, and mixtures of graph and tree visualization can play in the implementation of traceability mechanisms from requirements to elements of finite-state machine behavior (e.g., actions, states, transitions and guard conditions). These ontology-enabled traceability mechanism will play an important role in the team-based development of network-enabled platforms for analysis, design, and early validation and verification of information-age engineering systems (Delgoshaei and Austin 2011). The application of these concepts is illustrated through the study of a simplified rail transit system.
**Ontology-Enabled Traceability**

**Basic Model.** In state-of-the-art traceability mechanisms design requirements are connected directly to design solutions (i.e., engineering objects).

![State-of-the-Art Traceability Model](image)

**Proposed Traceability Model**

![Proposed Traceability Model](image)

**Figure 1. State-of-the-Art and Proposed Models of Ontology-Enabled Traceability**

As illustrated along the lower half of Figure 1, 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. Two more questions to ask are: Why make the traceability mechanism more complicated? What are the benefits? We observe that ontologies carry with them a conceptual representation and understanding of a particular domain. By explicitly connecting requirements to engineering system representations through ontologies we are indicating “how and why” requirements satisfaction is taking place. From an efficiency standpoint, 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. And because ontologies represent concepts for a problem domain, they are inherently reusable across families of similar projects. From a validation and verification viewpoint, relationships and dependencies among ontology concepts allow for notions of system completeness to be formally assessed. A second 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. For an operational system that is being monitored, real-time evaluation rules can also contribute to system management.

**Extensions for Multiple Viewpoint Design.** Figure 1 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. Real-world engineering systems have multiple stakeholders, many requirements, and multiple design criteria that will sometimes be in conflict. Therefore, instead of a simple requirements-to-ontology-to-engineering chain of connectivity, ontology-enabled traceability mechanisms will correspond to graphs of requirements, domain/ontology concepts and engineering objects.
Figure 2 shows the essential features of ontology-enabled traceability implemented in a setting for team-based design.

The basic model is extended from a chain to a graph of design concept entities. 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 in a team. This is one basic viewpoint. Similar relationships exist for the ontology engineer and the engineering of the system itself. 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. 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. Even though a group of ontologies may only provide a partial prescription for the functions and services that a system entity may need to provide it is still useful since a design can be checked with respect to the rules associated with the design concepts included in the model.

**Software Design Patterns**

While it is relatively straightforward to prototype a small-scale system that implements much of the functionality illustrated in Figures 1 and 2 (see, for example, the paper and technical report by Austin and Wojcik, 2010), 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. 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.

**Definition and Benefits.** 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. 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. 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 (Alexander et al. 1977). Gamma and co-workers (Gamma et al. 2002) point out that patterns facilitate reuse -- one person’s pattern can be another person’s fundamental building block. Software design patterns are particularly beneficial in the development of architectures for distributed systems (Buschmann et al. 2002).

**Mediator Design Pattern.** The mediator pattern defines 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. This strategy of development: (1) Simplifies communication between models and views because they do not need to implement the specific details of communication with each other, and (2) Provides maximum flexibility for expansion because the logic for the communication is contained within the mediator.

**Observer Design Patterns.** 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.

**Model-View-Controller 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. In the most common implementation of this pattern (see, for example, the Java patterns in Stelting and Maasen 2002), views register for their intent to be notified when changes to a model occur (Observer pattern). Controllers register their interest in being notified of changes to a view (Observer pattern). 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. Figure 3 illustrates the implementation of MVCs employed in our work. In a departure from standard approaches, the controller is positioned at the center of the pattern and the models and views communicate through the controller channels (i.e., the controller now acts as a mediator). After a view has notified the controller of a user action, the controller will update the property in the model based upon that action. In the left-to-
right direction, the controller becomes a registered listener of changes in the model and subsequently updates the view based on the notification triggered from the model.

![Figure 3. Model-View-Controller Pattern with Controller also acting as a Mediator](image)

Controllers also register their intent to receive updates from other controllers – for example, the ontology workspace controller will want to listen for property changes propagated from the requirements and engineering model workspace controllers.

**Architecture Design and Implementation**

Our objective is to develop a system/software architecture that will allow for the modeling of system behaviors as networks of communicating finite state machines (including hierarchical state machines and statecharts), and forward and backwards traceability of requirements to ontologies and engineering models.

**System-Level Architecture.** Figure 4 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 shown in Figure 2. Each block will employ a combination of the mediator and model-view-controller design patterns. Groups of related blocks are organized into workspaces. The requirements block is expanded to show the details of model, view and controller interaction.

The fully developed system will have work spaces corresponding to the requirements, ontology and engineering phases of system development (see Figure 2), plus a time workspace responsible for delivering temporal information to the system model via clocks and timers.
**Statechart Behavior Modeling, plus support for Traceability.** State machine behavior models describe the state transitions and actions that a system or its parts will perform in response to events. To allow for scalability of behavior models, and to provide support for concurrent processes operating within a single system, in our research, finite state machine behavior models are implemented as statechart extensions to an abstract model-view-controller assembly. This formulation is consistent with the statechart definitions provided by Harel (1988) and event-based behavior modeling provided by the OMG SysML specification (Friedenthal et al., 2008). The basic finite state machine (statechart) elements are modeled as collections of states, series of transitions, and guard condition objects, organized into the class hierarchy shown in Figure 5. Within a statechart, transition behaviors are defined in terms of time, change, and signal event triggers, guards and actions. Guard conditions are interfaces that verify the availability of a transition through the evaluation of Boolean expressions. Each state may have its associated entry action, exit action and a list of do-activities. Support for scalability of behavior models and modeling of concurrent behaviors is provided through nested sequential (i.e., composite states) and concurrent states.
Statechart views are graph-based extensions to an Abstract View Model. Metadata is used to recognize the runtime-specific data used by the state machine (e.g., current time, \( t \), and other variables used in the evaluation of guard expressions). Support for bi-directional traceability of requirements to elements of statechart behavior, and vice versa, is provided through the firing of property change events associated with the states, transitions and guard activity conditions (e.g., the behavior enters a new state, a transition is completed, the status of a guard condition evaluation changes). In the model-view-controller implementation, synchronization of the statechart models and views is handled by the controller.

**Application to Rail Transit Systems Design and Management**

The underlying tenet of our research is that software support for ontology-enabled traceability requires the use of (possibly new) software design patterns, plus multiple models of visualization. To experimentally assess this assertion, we are designing and building a series of progressively complicated software prototypes for requirements to finite state machine traceability. Our first application modeled the operation of a simple lamp that accepts user clicks as input, but also has clock and time model for automatic switching of the lamp state at prescribed times (Delgoshaei and Austin 2012). This section describes our second application, requirements-to-behavior traceability for trains operating in a simplified of the Washington DC Metro, the second largest rail transit system in the US.

**Systems Architecture.** 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 minimize delays, subject to cost and performance constraints.
Figure 6 is a schematic of the partially complete requirements-ontology-engineering system architecture as applied to behavior modeling of trains and schedulers in a small fragment of the Washington DC Metro System. When fully implemented, the system architecture will correspond to a network of communicating model-view-controllers (MVCs) spanning the requirements, ontology and engineering model workspaces.

The general pathway of communication among entities is as follows: the time controller notifies the scheduler, a controller having behavior, about a change in time. The scheduler triggers the train controllers to send/stop a train. A change to the train model will result in a call to statechart controller to update the state of the train. Finally, changes in the statechart model will be synchronized with appropriate views/labels in the requirements model/workspace (e.g., the requirements table view will highlight the requirement or requirements affected by the fragment of system behavior).

The lower half of Figure 6 shows a chain of loosely coupled workspaces for requirements, ontology, and engineering development. Within each workspace, the model, view, and controller classes are extensions of their abstract counterparts (e.g., Abstract Model in Figure 5). Individual workspaces register with and are connected to other workspaces via their controller. Changes in state that occur within a workspace are considered to be local. When a workspace controller receives notification of a change to the state of a model, this changes is propagated to the neighboring workspaces who have registered their interest in such events. Additional features
have been added to prevent the propagation of messages from entering loops. When the recipient controller receives a change, it updates its own model, and notifies the view and other listener controllers about changes to the model.

**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, 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.

**Requirements 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 metro system scheduler and trains. See Tables 1 and 2. Our goal for this prototype is to keep the model of behavior simple. The metro system will open at 5 am and close at 2 am. Trains will be dispatched every 10 minutes. No provision will be made for holiday or weekend schedules. Similarly, we assume that all trains will be parked at the end of the line when they are not in use. Beginning at 5 am, the scheduler will delegate a departure time to each train, which, in turn, will lead to the automatic generation of a timetable. Trains may (or may not) shop at every station along the line. This requirement opens the possibility of express trains. When a train stops at a station it must remain stationary for at least one minute.

<table>
<thead>
<tr>
<th><strong>Table 1 Scheduler Requirements</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scheduler Requirements</strong></td>
</tr>
<tr>
<td>1. The metro system will open at 5 am.</td>
</tr>
<tr>
<td>2. The metro system will close at 2 am.</td>
</tr>
<tr>
<td>3. Trains will be dispatched every 10 minute</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Table 2 Train Behavior Requirements</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Train Requirements</strong></td>
</tr>
<tr>
<td>1. Trains park at the end of the line when not in use</td>
</tr>
<tr>
<td>2. Trains need not to stop at every station</td>
</tr>
<tr>
<td>3. Train listen to the scheduler for dispatch time</td>
</tr>
<tr>
<td>4. Train creates its own timetable</td>
</tr>
<tr>
<td>5. Minimum time to go is 1 minute</td>
</tr>
</tbody>
</table>
From an implementation standpoint, textual requirements are translated into equivalent statements for expected behavior, which, in turn, can be translated into one or more mathematical constraints. This pathway of development is shown on the left- and right-hand sides of Tables 1 and 2. Some of these constraints will be evaluated in terms of attribute values in the system model. Many of the behavior requirements will be evaluated via guard conditions in the statecharts. The requirements controller mediates two different conversations between its model and view; Update the requirement table view as a result of user interaction (i.e. local change) and to display the changes imposed from other workspaces, state-chart (i.e. global change).

**Ontology Workspace.** The ontology workspace (see Figure 2) contains ontological descriptions of knowledge relevant to areas of expertise needed to design and manage a metro system (e.g., in the design of metro stations, schedulers, train behaviors), association and dependency relationships among these concepts, and design rules associated with these concepts. For example, the semantics of behavior specify the properties, operations, relationships, sequencing and coordination of behavior occurrences in models of behavior. Operations on behavior manage the details of process execution, such as when to start the process, when to stop the process, and the sequencing and coordination of lower-level processes and tasks. Properties include things like how long the process takes to complete its execution. Constraints can exist on the ordering of occurrences and multiplicity of relationships in ordered occurrences. Visual modeling languages such as UML and SysML provide concrete syntaxes for examining behavior from multiple perspectives; activity diagrams focus on the control of flow between actions or tasks; state machines model behavior as transitions between states.

We are experimenting with an ontology workspace implementation that involves the integration of scripting languages such as Python and Semantic Web, and the idea that ontologies can be introduced directly into the programming context so that OWL classes are usable alongside normally defined classes (Babik and Hluchy, 2006). In this approach, rule-enabled sets are described with OWL Description Logic construct. These sets can be defined by using N3 notation, e.g., ":Metro Station a owl:Class; rdfs:subClassOf :Node". This simply states that Metro Station is also a Node. Now let us introduce two instances, e.g., ":College Park a :Metro Station and :Transfer Point a :Node", the instances of Node are both College Park and Transfer Point. A metaclass is a class whose instances are themselves classes. This approach to implementation employs a metaclass-based representation of the rule-enabled sets based on the core metaclass, Thing. The constructor accepts two main parameters, i.e., N3 description of the i-set default namespace. Moreover, properties can be extended from Property class and be mapped to an OWL object property. With the Metro Station/Node relationship in place, the script of code:

```
Metro Station = Thing.new('Metro Station',' a owl:Class .')
Parking       = Thing.new ('Parking',' a owl:Class .')
hasParking   = Property.new('hasParking,' a owl:ObjectProperty .')
CollegePark  = Metro Station()
cpParking    = Parking()
hasParking = (College Park, cpParking)
```
establishes class definitions for Metro Station and Parking. It then creates CollegePark is an instance of Metro Station() with parking. Finally, it is possible to ask triple like queries and apply inference on the existing triples to obtain new ones. For example, hasParking.query(CollegePark, cpParking) evaluates to true. As Figure 7 shows the rules associated to ontological concepts can be implemented as inference rules, i.e. Design Rule 01, applied to OWL instances.

![Figure 7. Tree and Graph Views of a Simplified Metro System Ontology](image)

There are many interesting possibilities for the use of visualization techniques for ontologies and by extension, requirements and engineering model attributes needed for team-based design. Figure 7 shows, for example, 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 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 graph and tree visualizations serve complimentary purposes. While the former highlight the connectivity relationships among entities (e.g., a metro system has lines, metro stations and tracks), tree models and visualizations provide easy-to-navigate hierarchical structures for organizing and interacting with information in the metro system. Graphs can be converted into sets of tree structures. Conversely, sets of tree structures can be converted into graphs. We propose that a designer be 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 will transform the requirements-ontology-engineering workspaces into spreadsheet-like support for engineering design and systems management.
**Engineering Workspace.** This workspace holds the data and its representation associated with system structure (i.e., defined by attributes for track geometry, line color, station location, train size) and various aspects of the system behavior (i.e., scheduler behavior and individual train behaviors). Train functionality corresponds to a collection of methods for simple operations, e.g., `run()`, `stop()`, `park()`, `beginService()`, `endService()`. The details of how and when a train has to function depends on scheduler routine, and will be discussed in a moment. This engineering workspace controller mediates conversation between the engineering view and the engineering model (mediator design pattern). User interactions with the engineering views are implemented with mouse listeners attached to objects and collections of objects displayed in the view. To see how this works in practice, let us suppose that 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, this happens because the implementation in the `mouseMoved()` method in view causes the engineering controller to update its model. That is, the controller searches through the collections (trains, tracks and stations) to retrieve the model based on the coordinates of the selection and set it 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. Finally, the view display highlights the component that was initially selected.

![Scheduler Statechart](image.png)

**Figure 8. Statechart Behavior Model for the Metro System Scheduler Process**

**Scheduler Behavior (Model-View-Controller).** The metro system scheduler is responsible for assigning trains to specific tracks, based on their availability and satisfaction of other operational constraints. As indicated in Figure 8, behavior of the scheduler is modeled as a two-layer statechart. Either the scheduler is idle (its status during the night) of it is active. When the scheduler is active it is either dispatching a train (once every 10 minutes) or it is waiting to dispatch a train.
Train Behavior (Model-View-Controller). Individual train behaviors are also modeled as a statechart hierarchy. As shown in Figures 6 and 9, there are two main super states for the trains, namely, “in service” and “out of service”. Transitions between these super-states will be coordinated with activities in the scheduler. Our simple model assumes that train services will begin at a station at the end of the line. Services will begin with the doors being “opened” to board the passengers. Then once the doors have been closed, the train behavior will transition to “Departing Station”. When the train passes a certain distance from the station, it transitions to “Moving Along” state. Before reaching the station the train will transition to “Approaching Station” state. At each station, trains can either stop or go onto the next station. Under normal operations, a train will stop, open the doors to allow the passengers to depart/enter, close the doors, and then move onto the next stations. However, the model also allows for a train to stop at a station without opening the doors. This scenario will be activated, for example, as a result of delays in the system. This cycle will repeat until the train reaches the end of the line, becomes out of service, and goes to the end of the line to park.
Traceability of Requirements to Finite State Machine Behavior. As noted in the introduction, we propose that traceability mechanisms connect requirements to ontologies and ontologies to elements of the engineering model. The requirements, ontology, and engineering models will be displayed in a variety of views. Collectively, these mechanisms allow for an understanding for how a change in one entity will affect connected elements. All types of requirements (e.g. behavioral, safety and engineering) will be connected to ontologies and engineering models by a traceability pathway. Behavioral requirements will be quantitatively evaluated via the attribute values of states and guard condition expressions in the transitions of finite state machines. As a case in point, consider the requirement “Metro will be open at 5 am on weekdays.” There is a corresponding guard condition $t=5$ am in the scheduler statechart that satisfies this property (see Figure 8). The traceability mechanism may be activated in a number of ways. One example occurs when a user positions the mouse over the related requirement in the requirement workspace view. The propagation of traceability dependencies will result in the corresponding guard condition being highlighted in the statechart view. This traceability thread can be also followed from system behavior to 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.

Conclusions and 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. Future work will include development of a formalism to model design rules within the context of ontologies – see Figures 6 and 7. This will result in better verification procedures. Within the engineering workspace the engineering model can provide more detailed information relating to graph theory problems, e.g., shortest path for train travel between 2 points. This information can be used to perform trade-space analyses for performance and economics. 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. When the Washington DC 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.

References


Gamma, E., and R. Helm, R. Johnson, J. Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.


**Biography**

Parastoo Delgoshaei is a systems engineering graduate student in the Master of Science in Systems Engineering (MSSE) Program at the University of Maryland College Park. She has a bachelor of Computer Engineering (Software) from Shahid Beheshti University (SBU) Tehran, Iran. Her research in Ontology-Enabled Traceability Mechanisms is supported in part by a grant from the National Institute for Standard and Technology (NIST).

Mark Austin is an Associate Professor of Civil and Environmental Engineering at the University of Maryland, College Park, with a joint appointment in the Institute for Systems Research (ISR). Mark is Director of the Master of Science in Systems Engineering (MSSE) Program at ISR. He has taught short courses in Systems Engineering at US companies, and in Europe and South America. Mark has a Bachelor of Civil Engineering (First Class Honors) from the University of Canterbury, Christchurch, New Zealand, and M.S. and Ph.D. degrees in Structural Engineering from the University of California, Berkeley.