Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 8 (2012) 214 – 219

Procedia Computer Science

New Challenges in Systems Engineering and Architecting
Conference on Systems Engineering Research (CSER)
2012 – St. Louis, MO
Cihan H. Dagli, Editor in Chief
Organized by Missouri University of Science and Technology

Software Patterns for Traceability of Requirements to Finite State Machine Behavior

Parastoo Delgoshaei and Mark Austin*

Institute for Systems Rearch, University of Maryland, College Park, MD 20742, USA
Institute for Systems Research and Despartment of Civil Engineering, University of Maryland, College Park, MD 20742, USA

Abstract

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 the engineering entities were restricted to elements of system structure. But, of course, real engineering systems also have behaviors. This paper will report on research to understand the role that software patterns (e.g., model-view-controller) 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). We will present a simple lamp example.

Keywords: system engineering; traceability; system behavior; finite state machine

1877-0509 © 2012 Published by Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.procs.2012.01.045

1. Introduction

One way in which modern model-based system engineering (MBSE) procedures deal with the wide range of design concerns in system development is through the use of multiple types of models (e.g., for requirements, system structure and system behavior) and multiple visual formalisms (e.g., diagram types in SysML). This, by itself, is not enough for systems engineers to take full advantage of MBSE. They also need: (1) An understanding for how and why various 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. In a first step toward providing this capability, 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 ontology. In our proof-of-concept work [1,2], engineering entities were restricted to elements of system structure in a rail transportation system (e.g., tracks, lines and stations). But, of course, real engineering systems also have behavior (e.g., the trains). This paper reports on research to understand the role that software patterns (e.g., model-view-controller, mediator, observer, adapter) 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).

2. Ontology-Enabled Traceability

2.1 Basic Model for Multiple Viewpoint Design 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.

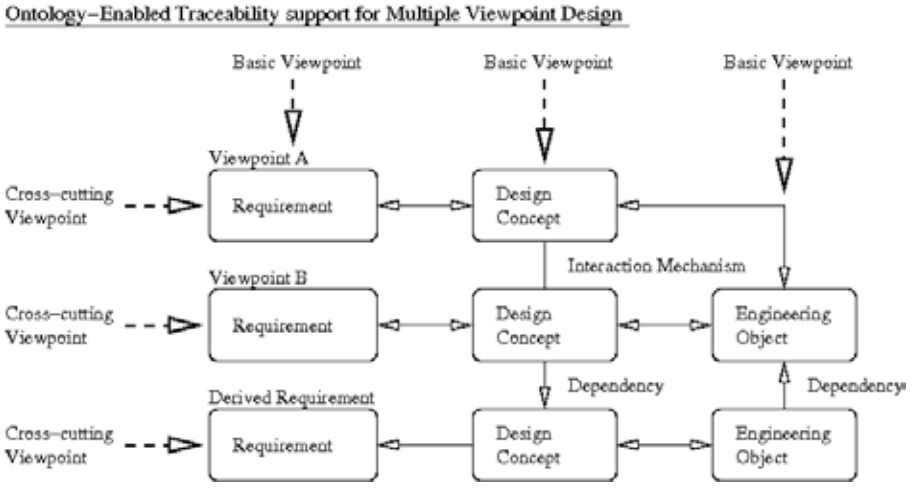

Fig. 1. Ontology-Enabled Traceability support for Multiple Viewpoint Design

Figure 1 shows the essential features of ontology-enabled traceability implemented in a setting for multiple viewpoint design. 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. Furthermore, because ontologies represent concepts for a problem domain, the ontologies are inherently reusable. From a validation and verification viewpoint, the 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. Moreover, if mechanisms can be created to dynamically load design concept modules into computer-based design environments, then rule checking can proceed even if the designer is not an expert in a particular domain. For an operational system that is being monitored, real-time evaluation rules can also contribute to system management.

3. Design Patterns for Requirements to Finite State Machine Behavior Traceability

Our prototype implementation of requirements to finite-state machine behavior traceability makes extensive use of the model-view-controller, mediator, and observer design patterns.

3.1 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 [3]. Gamma and co-workers [4] 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 [5].

3.2 Mediator, Observer and Model-View-Controller Design Patterns

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 simplifies communication between models and 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 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.

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 [6]), 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 change in the model.

3.3 System Architecture

Figure 2 shows the system architecture currently being implemented as a pyramid (i.e., two-level graph) of model-view-controllers.

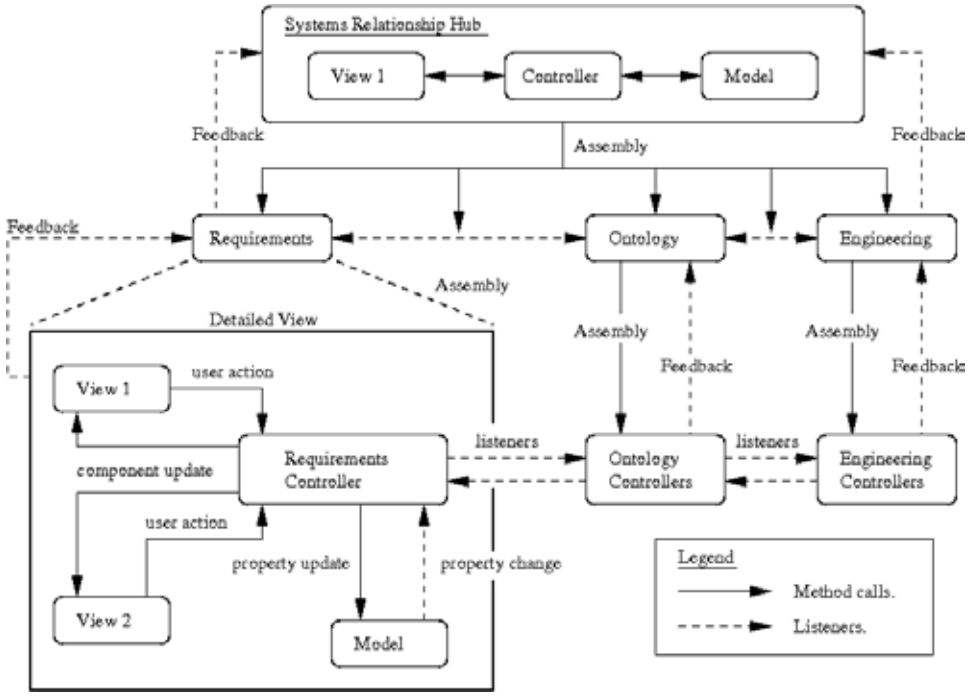

Fig. 2. Software architecture for the implementation of ontology-enabled traceability.

The fully developed system will consist of a requirements phase, an ontology phase, an engineering development phase. 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 Fig. 1. Each block will employ a combination of the mediator and model-view-controller design patterns. The requirements block is expanded to show the details of model, view and controller interaction

4. Finite State Machine Behavior and Requirements-to-Behavior Traceability

Our objective is to develop a software infrastructure that will allow for the modeling of system behaviors as networks of communicating finite state machines. To allow for scalability and the possibility of concurrent processes operating within a single system, finite state machine behavior models are built from an abstract model-view-controller assembly, plus extensions for statechart behaviors [8]. See Figure 3. Appropriate interfaces and abstract class definitions are added for the assembly of traceability models. Metadata is used to recognize the runtime-specific data used by the statechart (i.e., to keep a list of states, currently active states, transitions and guard conditions). The Metadata class fires propertychange events when the statechart enters a new state or starts a transition. Changes in state can also occur when events

are fired in the statechart model. Support for traceability includes state and transition classes, both of which fire propertychange events when their activity status is updated. Guard conditions are interfaces that verify the availability of a transition through the evaluation of evaluate boolean expressions in the statechart. Guard interfaces notify the controller when their status is evaluated to either true or false.

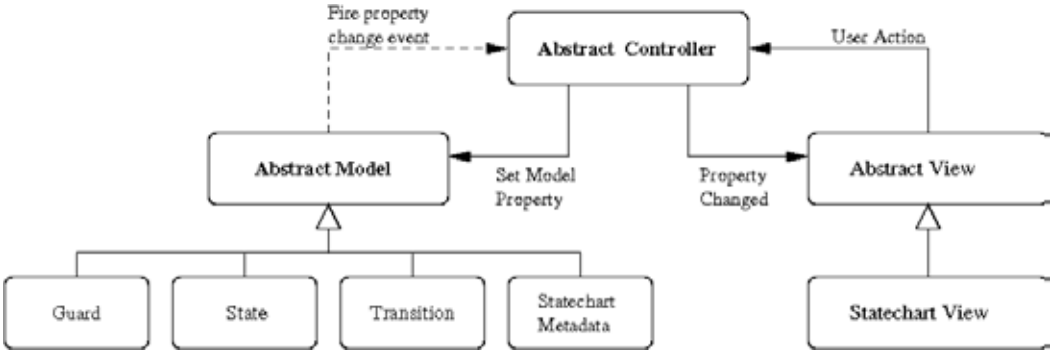

Fig. 3. Abstract models, controllers, and views supporting statechart behavior modeling.

5. Software Prototype: Behavior of a Simple Lamp

Computational support for requirements-to-behaviour traceability is being implemented within the framework of Figures 1 and 2, and through the systematic implementation of progressively complicated applications. Our first application is behaviour of a simple lamp having an on/off switch and a clock. Table 1 summarizes the system requirements and expected behaviour.

System Requirements	Expected Behaviour
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 1. Lamp requirements and expected behavior.

Figure 4 is a schematic of the partially complete lamp prototype augmented by workspaces for the engineering, ontology and requirements models. Within each workspace, the model, view and controller classes are extensions of their abstract counterparts (e.g., AbstractModel). The engineering model has a system structure (i.e., defined by attributes for lamp geometry, color, style) plus a model for system behaviour implemented as small network of controller behaviours. Basic behaviour is handled by a lamp controller, which in turn, links to a statechart controller (not shown). A barebones clock would provide a switch for the lamp to be turned on/off, subject to the lamp being connected to a power supply (i.e., any transition to an On state will only occur when a guard check on power evaluates to true). However, in order to make the behaviour a bit more interesting, we add a clock and time model so that requirements 1 and 2 in Table 1 may be satisfied. The observer design pattern regulates communication among the controllers, both locally within a workspace, and globally throughout the traceability network. A user can interact with the engineering view by clicking the switch (displayed as a small black box) on and off.

Changes in the lamp state are automatically propagated to the statechart view, and when the model is complete, also to a requirements table view and ontology graph view.

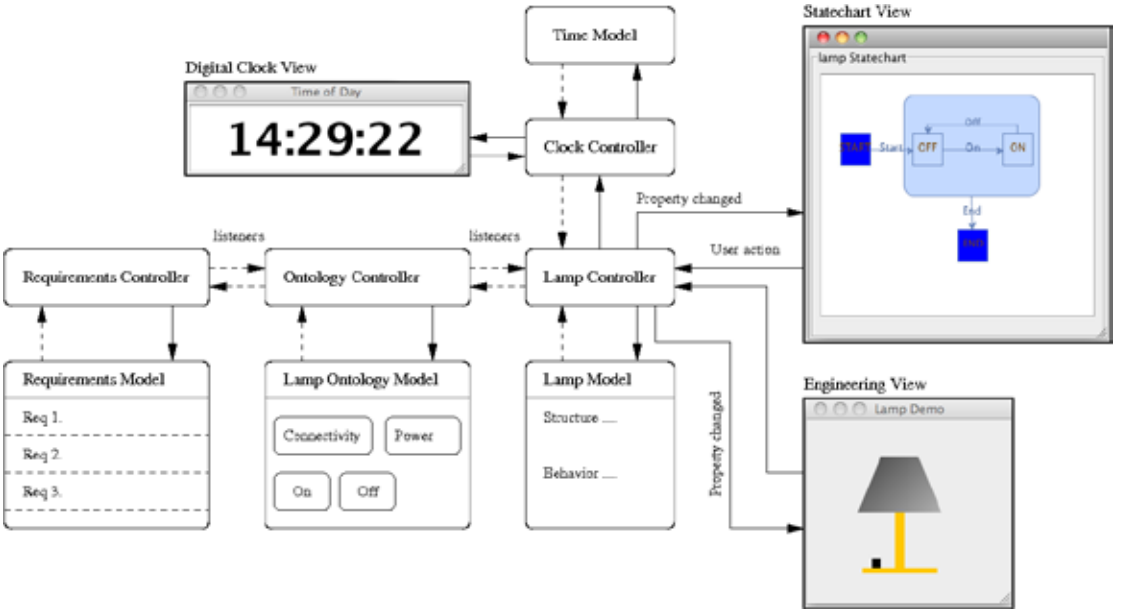

Fig. 4. Schematic of the lamp behavior software prototype

6. Conclusions and Future Work

Our program of research to understand the role that software patterns and mixtures of graph and tree visualization can play in the implementation of ontology-enabled traceability mechanisms is still in its infancy. When the lamp application is complete we will move onto trains and build a simplified model of the Washington D.C. Metro System, with a focus on timetables, schedules and train behaviors.

References

[1] Austin M.A. and Wojcik C.E., "Ontology-Enabled Traceability Mechanisms", *Twentieth Annual International Symposium of The International Council on Systems Engineering,* (Chicago, USA, July 12-15, 2010). Paper 562.

[2] Delgoshaei P. and Austin M.A. , Software Design Patterns for Ontology-Enabled Traceability, Ninth Annual Conference on Systems Engineering Research , Redondo Beach, CA, April 14-16, 2011.

[3] Alexander C., Ishikawa S., and Silverstein M., A Pattern Language: Towns, Buildings and Construction, Oxford University Press, 1977.

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

[5] Buschmann F., Henney K., and Schmidt D.C., Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, John-Wiley and Sons, 2007.

[6] Stelting S. and Maassen O., Applied JAVA Patterns, SUN Microsystems Press, Prentice-Hall, 2002

[7] Austin M.A., Mayank V., and Shmunis N., PaladinRM: "Graph-Based Visualization of Requirements Organized for Team-Based Design, Systems Engineering": *The Journal of the International Council on Systems Engineering*, Vol. 9, No. 2, May 2006, pp. 129-145.

[8] Harel D., On Visual Formalisms, Communications of the A.C.M., Vol. 31, 1988, pp. 514-530.