Modeling automatic train regulation systems

J.K. Tobin
Alcatel Canada Inc., Transport Automation Systems, Canada

Abstract

The increasing complexity of automatic train supervision and regulation systems (ATS/ATR) has placed more and more importance on the use of modeling to describe and understand system behaviour. Such a model should provide a description suitable for requirement review and approval by the rail operator as well as detailed views for use by designers, implementers and testers. The Unified Modeling Language (UML) is a visual modeling standard that has been adopted by Alcatel to capture ATS/ATR models. The choice of a standard notation was motivated by our need to exchange models with various stakeholders; the selection of UML in particular was influenced by its wide adoption in the software industry.

We have gained our experience with using the Unified Modeling Language during the development of our latest generation product for train regulation and supervision. During this development we have defined an approach for requirements modeling and selected a useful subset of UML notations for analysis and design capture. UML has proven to be very effective at mapping from the requirements model to the analysis model to the design model and finally to implementation. We have found that this natural correspondence between models has simplified the task of reusing our code base on new projects and we believe it will be beneficial in the areas of maintenance and new feature incorporation.

1 Introduction

In 1996, Alcatel Canada started a complete re-engineering of its train regulation and supervision product, the SELNET® Management Centre (SMC). The major goals of this effort included:
The creation of a framework from which all train regulation and supervision products delivered by Alcatel could be derived (fixed block and moving block automatic train control as well as centralized traffic control),

- A product that could be easily integrated with third-party solutions (E.G., System Control And Data Acquisition (SCADA) systems)
- A solution that could support data configuration, and
- A system that would scale to meet future control system needs.

The results of this re-engineering effort have been used in a revenue moving-block system for two years. We are now reusing this technology on three additional moving-block systems as well as a fixed-block system.

Besides the code base, important artifacts of this effort are the requirements, analysis and design models. These models are essential to describe the system to railway operators, to implementers and to testers. The maintenance of these models is also vital to the evolution of the new product: as new features are incorporated, these models must be kept consistent to be useful. The decision about how this information was to be captured was of strategic importance.

Figure 1 shows the typical components in an automatic train supervision/regulation system supplied by Alcatel Canada. A SELNET™ Management Centre (SMC) consists of several computers providing different functions connected by a Local Area Network (LAN). Workstations provide operators a user interface to examine system status and issue commands. Gateway computers act as protocol converters to enable communication with other systems (either connected directly to the network or by other interfaces). The main application which provides train routing, tracking and regulation logic is deployed on a computer called the schedule regulation system. Software that provides mission-critical functions is deployed redundantly.

Figure 1: SELNET™ Management Centre (SMC) block diagram
2 Modeling Overview

A model is a simplification of a system. We model systems so that we can better understand what they are supposed to do and how we are able to build them. Software-intensive systems provide a great deal of flexibility in terms of possible features, this adds complexity increasing the need to model.

A number of languages, both diagrammatic and textual, have been defined for modeling software-intensive systems such as Specification Description Language (SDL), Booch Notation, Object Modeling Technique (OMT), Z Language, and Entity-Relationship (ER) Modeling. We chose to capture our requirements, analysis and design models using the Unified Modeling Language (UML). It was able to satisfy the following needs:

- Understandability. We wanted the models to be readable by individuals without mathematical training.
- Object-orientation. It was important that the design model be easily mapped into our implementation language (C++).
- Tool support. UML has been widely adopted by tool vendors.

The UML is a graphical language for describing software-intensive systems. It is maintained by a consortium called the Object Management Group (OMG) and is currently at revision 1.3 (this can be found in [1]). The OMG is an approved Publicly Available Specification (PAS) submitter to the International Organization for Standardization (ISO) and is submitting UML for international standardization. This paper will only touch on a small subset of the UML. The intent is to provide examples to demonstrate how we were able to apply it to our domain of train control systems. Booch, one of the original specifiers of UML, provides a tutorial on using it in [2].

3 Requirements Model

The capture of requirements was performed using a technique called use case analysis as described in Jacobson [3]. Jacobson defines a use case as “a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.” In UML terms, an actor is a role played by a human, a device or another system when interacting with a use case. The functional requirements of the system are documented as a collection of use cases. UML provides a means to organize use cases by defining associations between them. These associations are documented in a use case diagram.

Figure 2 shows an example use case diagram. The information contained in this diagram can be summarized by the following assertions:

1) The use case description for performing a “Launch train” involves depot controller interaction.
2) The “Launch train” use case depends upon the behaviour defined in “Regulate to run” and “Validate user command” to accomplish its goals.
3) The “Validate user command” use case makes use of the description in “Log system data” to perform its actions.
4) The use case description for "Log system data" involves an external "Log Database" device.
5) The "Launch train" use case description is extended by "Automatic launch allocation" to provide additional behaviour for systems that have an automated yard.

Figure 2: Train Launch use case diagram

Since actors represent entities that are external to the system being modeled, the collection of all use case diagrams establishes the context of the system. Figure 3 shows another use case diagram that serves as the context diagram for an SMC. The UML package notation, drawn as a folder with a tab, is shown in the diagram. The package "SMC" contains all of the use cases of the system. Examples of three types of users are shown: staff supervisor, depot controller and central operator. Even though these roles could be filled by the same person, they are distinguished since the goals of interaction is different.

Figure 3: SMC context diagram

UML does not define how use case descriptions are formatted or specified. They may take the form of informal structured text, formal text or even pseu-
docode. Our goal was to make our use case descriptions readable by the user, not just the developers specifying and implementing the system. For this reason, we chose informal text to describe use cases. Table 1 shows an example of our use case format. In the interest of space, the use case has been simplified; however, the example provides a good indication of how our descriptions are structured. Each use case identifies the preconditions required before it can be performed as well as the starting condition. Use case flow descriptions are separated into normal flow and exceptional flows. These descriptions are labeled and may contain nested sub-flow descriptions. Flow nesting is documented by our numbering convention. Enclosing a use case name in square brackets, as shown in the “Allocation” flow, demonstrates our convention for referring to other use cases.

Table 1. Launch train use case

<table>
<thead>
<tr>
<th></th>
<th>Title</th>
<th>Launch train</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Description</td>
<td>This use case describes how the SMC launches trains into service.</td>
</tr>
<tr>
<td>1.1</td>
<td>Preconditions</td>
<td>A schedule has been selected.</td>
</tr>
<tr>
<td>1.2</td>
<td>Normal flow</td>
<td>This use case starts when the next build up time for a schedule occurs.</td>
</tr>
<tr>
<td>1.3</td>
<td>Allocation</td>
<td>The SMC displays the launch list for the build up. The launch list specifies train type and run start time. The depot controller assigns a train to each run requiring launch. Each command is validated by [Validate user] before execution.</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Launch</td>
<td>When a train arrives at a disposition point and it has been associated with a run, and the train has been automated, the train is assigned the run which is then handled by [Regulate to run].</td>
</tr>
<tr>
<td>1.3.2</td>
<td>Launch completion</td>
<td>When all of the trains in the launch list have been launched, an indication of launch completion is provided. The use case ends.</td>
</tr>
<tr>
<td>1.4</td>
<td>Exceptional flows</td>
<td></td>
</tr>
<tr>
<td>1.4.1</td>
<td>Train failure</td>
<td>If a train which has been associated with a run for launch fails before it arrives at the disposition point, the depot controller can remove the association and associate another train. If no other train is available, the depot controller may cancel the launch ending the use case. Each command is validated by [Validate user] before execution.</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Incorrect train type</td>
<td>The depot controller associates a train of incorrect type with a run. The SMC asks for confirmation. If the depot controller grants permission, the train is associated and will be launched, otherwise the command is ignored.</td>
</tr>
</tbody>
</table>
Table 2 shows an example of a use case that extends another. We have found this is a powerful technique for organizing use cases. Using an extending use case allows the specification of additional or optional behaviour without having to change the original use case. The extension points or flows that are replaced by the extending use case are identified in the use case text. Note that the use case is not complete outside of the context of the use case it extends.

Table 2. Automatic launch allocation

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Title</td>
</tr>
<tr>
<td>2.1</td>
<td>Description</td>
</tr>
<tr>
<td>2.1</td>
<td>Preconditions</td>
</tr>
<tr>
<td>2.2</td>
<td>Automatic Allocation</td>
</tr>
<tr>
<td>2.3</td>
<td>Exceptional flows</td>
</tr>
</tbody>
</table>

SMC use cases necessarily include a significant amount of interaction with users. When documenting user interaction we describe the essential purposes of the interaction. Constantine [4] describes such use cases as essential use cases. This approach allows us to keep our use cases from becoming complicated by user interface details. Remaining technology-neutral has the additional benefit of permitting multiple user interface solutions.

4 Analysis Model

The purpose of analysis is to define how a system behaves without specifying implementation details to avoid constraining the design. In a sense this activity is used to determine the “true” or detailed requirements of the system. The use cases provide a complete description of the system in operational terms. The analysis model provides additional behavioral details as well as documentation of system organization by specifying how model elements relate to each other.

The analysis model provides both a structural and behavioral view of the system in terms of problem domain objects. An object is an entity that can accept requests to do something (messages) and maintains the state needed to perform these requests. The model structure describes how objects are associated with each other; the behavioral description shows how they interact. Most of our documentation of these interactions is done with UML sequence diagrams. Sequence diagrams that form a part of the analysis model show interactions of objects representing a particular scenario taken from a use case. Figure 4 shows an example sequence diagram. Objects that participate in the interaction are placed at the top of the diagram on the X axis. Messages that are sent and received are placed along the Y axis with time increasing from top to bottom. Textual anno-
tations are provided on the left-hand side of the diagram to provide additional documentation.

![Sequence diagram](image)

Figure 4: Sequence diagram

Modeling the organization or structure of a system is done using *class diagrams*. Class diagrams show model abstractions called *classes*. A class is a description of a collection of objects that all share the same structure and behaviour. In the sequence diagram we saw an object named “42” that belonged to the class “Train”. As such, we would expect this object to behave like all other trains and to associate with other objects like all other trains. Figure 5 shows an example class diagram. It can be summarized by the following statements:

1. A schedule is composed of one or more runs.
2. A run includes an attribute called “start time”.
3. A schedule creates a launch list which may be associated with zero or more trains. A launch list will accept an “arrival” message.
4. A train may be associated with a run called its assignment. No other train may be associated with the same run.
5. A train is may be associated with a launch list and will accept an “assign run” message.

The note box expresses the constraint found in statement 4) using the *object constraint language* (OCL) which is a part of the UML standard (see [5] for a description of OCL). This is an example of the concision that is possible when using UML.

Note that the class diagram falls completely within the domain of analysis and may be read by individuals unfamiliar with software design or implementation. It is constructed solely from the use case; however, important details that are required for design, such as the constraints on associations, have been added.
The design model provides solutions to realize the system behaviour documented in the analysis model. The most strategic aspect of the design model is the software architecture. Software architecture defines how the software is structured into independent elements and how these elements interact. We have taken a component-based approach to our software development: software is partitioned into components, physical units of software, which are accessed solely through published interfaces. UML provides a variety of choices for modeling components as pointed out by Kobryn in [6]. As shown in Figure 6, we decided to use UML subsystems to model our components. Subsystems provide a grouping mechanism (they are packages) and represent a behavioral unit (they can participate in sequence diagrams). In the example, the following is asserted:

1) The component Signaling System Handler has an interface called ITrain.
2) The component Schedule Server has interfaces ISchedule and ILaunchList.
3) The component Train Regulation uses the interfaces ITrain and ISchedule.

**Figure 5: Class diagram**

**Figure 6: System components**
We created the design model from the analysis model through a process of adding details to classes and adding additional design classes to support our solution. This process works very naturally since classes are used to represent both analysis abstractions as well as design abstractions. The challenge is to avoid “losing” the analysis model. We did not want to separate the two models since maintaining consistency would become very difficult. We used the UML concept that a diagram is only a view of the underlying model by maintaining separate UML diagrams representing the analysis view as well as the design view. We believe that this practice was the best option available; however, we were unable to avoid some corruption of the analysis model by certain design forces (E.G., the need to split a class across two components).

The most detailed level of software design is arguably the code. UML classes correspond to native features in object-oriented languages such as C++, our implementation language. Through tool use we are able to maintain consistency between our code and design model.

6 Conclusion

The decision to use a modeling language that naturally supports the transitions between requirements, analysis and design models and ultimately to code has proven effective for the development and reuse of a modern train supervision and regulation system. We have been able to involve rail operators in the review of UML requirements and analysis models. Our software architecture and design is well documented using UML. The close correspondence between UML model elements and implementation constructs has ensured design and code consistency.

Using UML will not ensure good requirements are written or flexible software designs are created; however, it does provide a common syntax for their description. We would like to see its continued adoption in the area of rail transport for the exchange of such information.

7 References


