Debugging and Testing Behavioral UML Models

Dolev Dotan
IBM Haifa Research Lab
Haifa University, Mount Carmel
Haifa, 31905, Israel
+972-4-8281010
dotan@il.ibm.com

Andrei Kirshin
IBM Haifa Research Lab
Haifa University, Mount Carmel
Haifa, 31905, Israel
+972-4-8296581
kirshin@il.ibm.com

Abstract

As software complexity increases, the process of software development is shifting from being code-centric to model-centric. For this purpose, UML augments the object-oriented paradigm with powerful and flexible behavioral modeling capabilities. It allows the developer to describe the system's behavior in a higher level of abstraction by using state machines, activities, and interactions. To facilitate such model-driven development, we present a plug-in for IBM Rational's modeling tools, which enables the execution, debugging and testing of UML models. The presentation will show how to use our tools to discover defects early in the development cycle, thus preventing costly rework at later stages. We will highlight the innovative features of our tools, such as model-level debug control, interactive dynamic debugging, and the extensibility that allows developing support for UML profiles.

Categories and Subject Descriptors D.2.2 **[Design Tools and Techniques]**

General Terms: Design, Languages.

Keywords: UML, State Machines, Activities, Profiles, Debugging, Software Development.

1. Introduction

The Unified Modeling Language (UML) augments the object-oriented paradigm with powerful and flexible behavioral modeling capabilities. It allows the developer to describe the system's behavior in a higher level of abstraction by using state machines, activities, and interactions. In recent years, these capabilities are being increasingly utilized by developers, especially in areas such as Systems Engineering and Service-Oriented Architecture.

To facilitate model-driven development, developers must be empowered to debug and test UML models. In this paper we present how this can be achieved by using our plug-in to IBM Rational's Eclipse-based modeling tools.

In the next section, we give an overview of the behavioral modeling features found in UML 2.0. In Section 3 we introduce our UML Model Debugger.

2. UML Behavioral Modeling

In this section, we highlight the advanced features of UML activities and state machines, their integration with the object-oriented paradigm, and their extensibility with UML Profiles.

UML activity diagrams can be seen as sophisticated flowcharts, containing advanced features such as:

- Control and Data Flow
- Conditionals (choice node, guards)
- Concurrency (fork & join)
- Nested behavior calls
- Data stores

As a language for describing behaviors of object-oriented systems, activities have specialized actions (steps) for calling object operations, reading and modifying object attributes, sending signals, and more. In addition, actions and guard expressions (written using some textual action language) can reference class attributes, call class operations, etc.

UML state machines integrate Harel charts into the object-oriented paradigm. They include such features as:

- Behaviors for: state entry, do, exit, transition effect
- Composite states
- Nested state machines
- Concurrency (orthogonal states)
- Transition guards
- Conditionals (choice, junction)

State machines can also be tightly integrated with the object-oriented model. Operation calls and attribute changes can be specified as transition triggers. Behaviors and guard expressions can reference class attributes, call class operations, etc.

Action Language: Behaviors can also be defined using any textual action language (such as Java) – such behaviors are called Opaque Behaviors. An action language is also used to define the "atomic" behavior elements, such as activity actions, control flow guards, and transition effects.

Usages for Behaviors: The UML behavioral features described above can be used to specify the lifecycle of a class, the implementation of an operation, or can be stand alone model-elements (e.g. for representing a business process).

UML Profiles: Stereotypes can be applied to behavioral elements in order to modify or specialize their semantics to fit some domain-specific language (DSL).

3. The UML Model Debugger

Debugging at a higher level of abstraction: The UML Model Debugger is built as much as possible in the fashion of the Eclipse code debugger (see Figure 1). As such, it provides the following familiar views: Debug (for controlling the debug session), Variables (for observing the attributes of the current object), and Breakpoints. However, debugging at a higher level of abstraction entails some innovative conceptual differences.

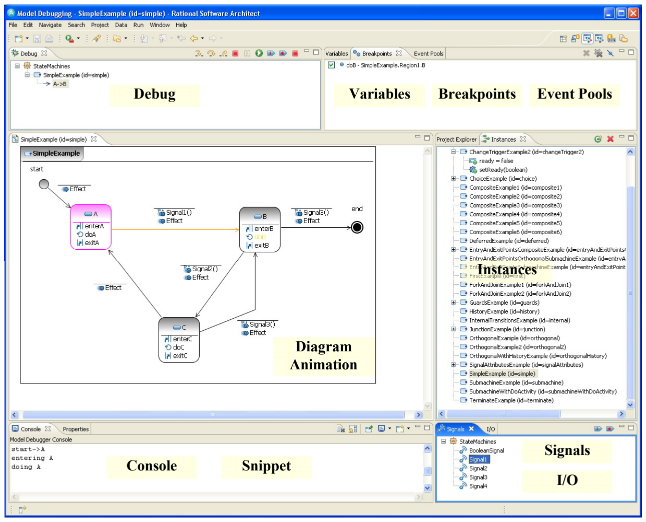

Figure 1: The Model Debugging Perspective

For example, instead of threads and operation call stacks, the debug view shows behavioral model elements – transitions, states, actions, etc. These are organized in a tree to allow representing high-level concurrency (e.g. orthogonal states, activity actions following a fork).

The debug session is also controlled at the behavioral model element level. Stepping into a transition, for instance, will take us through one or more exit behaviors, transition effect behaviors, and entry behaviors. Diagram animation (see Figure 2) is used to show the progress through these different model elements.

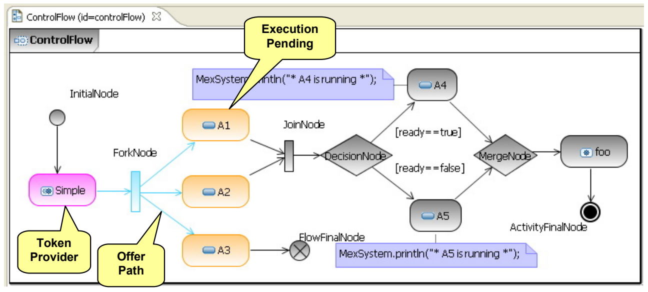

Figure 2: Diagram Animation of a UML Activity Diagram

UML's high level of abstraction makes explicit the notion of signals. Using the debugger's Signals and Debug views, the users can send signals to specific instances, thus simulating signal reception from other objects or the external environment. In addition, the Event Pools view provides support for UML's notion of active classes by listing each active object's pending events.

Novel debugging modes: Our debugger supports the usual fashion of debugging, in which the user starts in some entry point ("main") and runs until a breakpoint (placed on a behavioral element) is hit. However, we also support a unique feature we call "interactive debugging". In this debugging mode, the user can use the Instances view to interactively create instances of classes and invoke their operations and behaviors. This makes it possible to debug parts of the model without having to explicitly write entry points and set-up launch configurations.

Another unique feature our tool provides is the support for incomplete models. In cases when there is missing information regarding the next step, the debugger will ask the user what to do. For example, if guards are missing following a choice node, the debugger will let the user choose which path to take. This is particularly useful for debugging architectural models which sketch the behavior informally, for instance writing guards in a human language, and leaving the formal implementation details to a latter phase.

Extensibility: The Model Debugger is designed to be extensible to support modified semantics for different UML Profiles. For example, we have created an extension to support the semantics of the UML Profile for Voice [2].

Status: Currently, our debugger supports state machines and activities. Interactions (e.g. sequence diagrams) will be added soon. Java is supported as the action language. Most of the structural features of class, package and composite structure diagrams are also supported.

References

[1] Unified Modeling Language Version 2.1.1 Specifications, Object Management Group, http://www.omg.org/technology/documents/formal/uml.htm. Accessed on July 2nd, 2007.

[2] UML Profile for Voice final adopted specification, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/2006-10-02. Accessed on July 2nd, 2007