SM@RT: Supporting Model-Based Runtime Management

Hui Song\textsuperscript{1}, Yingfei Xiong\textsuperscript{2}, Gang Huang\textsuperscript{1}, Zhenjiang Hu\textsuperscript{1}, Franck Chauvel\textsuperscript{1}, Hong Mei\textsuperscript{1}

\textsuperscript{1}Key Laboratory of High Confidence Software Technologies (Ministry of Education), Peking University, Beijing, China
\{songhui06,huanggang,franck.chauvel,meih\}@sei.pku.edu.cn

\textsuperscript{2}Department of Mathematical Informatics, University of Tokyo, Tokyo, Japan
xiong@ipl.t.u-tokyo.ac.jp

\textsuperscript{3}GRACE Center, National Institute of Informatics, Tokyo, Japan
hu@nii.ac.jp

ABSTRACT

We present the SM@RT tool that supports model-based runtime system management. Developers provide meta-models describing both the target system and the desired architecture style, and the SM@RT tool automatically generates the synchronizer that reflects the running system as an architecture model.

1. INTRODUCTION

Continuously-running systems require dynamic management capabilities to enable system maintenance and evolution. Although existing execution platforms such as JEE now provide such facilities, their related management APIs remain too technical, and thus we require the integration of efficient high-level model-based management approaches on the running systems.

However, there are two key-challenges to achieve model-based runtime management:

1. Different systems provide different management APIs, and we cannot directly apply the model-based techniques on these low-level ad hoc management APIs.

2. Many runtime management approaches are designed upon generic architecture models that are heterogeneous with the specific system structures. It is not easy to maintain the causal link between the architecture model and the running system.

In this paper, we present the SM@RT\textsuperscript{1} tool to support model-based runtime management of existing running platforms. Given the meta-models of both the target system and the needed architecture style, SM@RT tool automatically generates a bi-directional causal link between the system and the model, enabling therefore the integration of model-based management approaches on the target system.

This paper is organized as follows: we first describe the tool in Section 2 and then illustrate the integration of the C2-style architecture-based management approach with the JOnAS platform, in Section 3.

2. TOOL OVERVIEW

This section gives an overview of the the SM@RT tool. Figure 1 illustrates the inputs and outputs of this tool, and we describe the two parts of this tool in the following paragraphs, as well as a brief introduction about our research work underpinning this tool. The detailed discussion could be found in our technical report [4].

In order to reflect the running systems into MOF-compliant models (for the first challenge), the SM@RT tool requires two inputs, namely the system meta-model and the access model. The former describes the different types of elements inside the running system [3], as well as their local states and relations. The latter one (the access model) describes how to manipulate those runtime elements through the management APIs. The generated adapter implements the standard MOF reflection interface. External programs (such as OCL engines) manipulate the running system through this standard interface (in the same way as manipulating an MOF compliant model), and the adapter forwards the external manipulation into proper invocations on the management API.

Our research contributions include a modeling language for specifying management APIs and the code generation mechanism.

The synchronization of the running system with architecture model (for the second challenge) requires an architecture meta-model describing the needed architecture style and a QVT transformation describing the relationship between the system and the architecture. SM@RT employs a QVT engine (the mediniQVT[1]) to propagate changes from architecture model to running system, and vice versa. But QVT cannot address the issues of change conflicts and API write failures.

Footnote 1: SM@RT (pronounced as “smart”) stands for “Supporting Models at Run-Time”. Project web site: http://code.google.com/p/smatrt

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXXX-XX-X/XX/XX ...$5.00.
We designed a synchronization algorithm handling those two issues: we use a three-way merge at the system side to filter out the conflict changes, and use another three-way merge at the architecture side to identify which architecture changes are not properly written to the system. Our contributions lie in the first attempt and a proper algorithm towards the generation of causal links between architecture models and running systems.

We implement the SM@RT tool as a set of Eclipse plugins. From the developers’ point of view, the SM@RT tool is constituted of some graphical editor for writing the specifications, as shown by the snapshot in Figure 2. It also provides a set of menu items that guide developers in launching the inside generators to produce adapters and synchronizers, which are all distributed as Java libraries (Jar files).

3. TOOL APPLICATIONS

The following case study illustrates how to apply the SM@RT tool to reflect JEE systems as C2-styled architecture models, realizing a typical model-based management approach (the architecture based runtime evolution by Oreizy et al.[2]) on a practical software platform (the JOnAS JEE application server).

Figure 2 illustrates the specification we wrote. The architecture meta-model (Figure 2(a)) defines the C2 style, where Architectures are made of Components linked by Connectors. The system meta-model (Figure 2(b)) specifies the structure of JEE systems. It defines the states and the relationships between EJBs, JDBCDataSources and WebModules running on a system. The access model (Figure 2(c)) specifies how to invoke the JMX compliant management API. The background editor shows the structure of this access model. The pop-up diagram shows a sample piece of code for getting the connection pool size of a data source. Finally, the excerpt of the QVT transformation (Figure 2(c)) specifies the relation between Components and JDBCDataSources.

We use a sample running system, a Java Pet Store application running on a JOnAS server, to illustrate the capability of the generated adapter and synchronizer. Figure 3 shows the architecture model reflected from this target system. The components (yellow rectangles) represent the EJBs, web modules and data sources in the system. The bottom “Properties” view lists the attribute values of the “EJB1” component. This architecture model is dynamic. That means, for example, if administrators launch the “synchronize” command, the “pool size” attribute will be refreshed according to the current connection pool of this data source. Administrators can also edit the “max pool” attribute on the architecture model, and then the underlying connection pool will be enlarged.

We use the synchronizer to reproduce the typical management scenario presented by Oreizy et al.[2]: We add new components for extracting pet information and representing it as RSS seed, as shown inside the red frame in Figure 3, and then launch “synchronize” command. The synchronizer automatically deploys the implementations (in the form of EJBs and Web Modules) of these components into the JOnAS server, enhancing the original system with RSS publish function. We can also change the connection between these components, and the contents of RSS seed will change correspondingly.

We also applied SM@RT tool to support several other management cases, e.g. reflecting a PLASTIC-based mobile system into a C/S styled architecture model, reflecting the class structure in a Jar file as a UML class diagram, reflecting the package structure in an Android mobile device as a tree-based model. These cases demonstrate the following properties of our tool. First, it reduces the development cost. Take the Jar/UML case as an example, the specification is only in the size of 91 model elements and 124 lines of code. In contrast, an equivalent, manually constructed case named Jar2UML2 costs 3108 lines of Java code. Second, its performance and applicability are acceptable. For the Android case, we successfully execute the generated adapter on the HTC G1 mobile phone.

Detailed descriptions about the cases, including the specifications and generated code, can be found in our website.

4. REFERENCES


http://ssel.vub.ac.be/ssel/research/adv/jar2uml
APPENDIX

In this part, we first describe the detailed process of the C2-JOnAS case study. We also use this case to evaluate the development efficiency and execution performance of SM@RT tool. After that, we briefly introduce the other case studies we have undertaken.

The C2-JOnAS Case

This case study is for reflecting the JOnAS systems as C2 styled architecture models, so as to realize the typical C2-based runtime management approach on the practical JOnAS-JEE systems.

Installing SM@RT Tool

The SM@RT tool is distributed as a set of Eclipse plug-ins, which can be downloaded from our project website. We deploy these plug-ins in an Eclipse 3.4 platform.

Writing the Specification

We use the model editors in SM@RT tool to write the specification. Figure 2 shows the snapshots of this specification in different editors.

We write the architecture meta-model (Figure 2(a)) according to the description of C2 style by Oreizy et al [2]: Components abstract the functional parts of a system, and carry some attributes revealing their runtime state. They interact with each other through the Connectors, forming a layered structure.

We write the system meta-model (Figure 2(b)) according to the JEE specification. We defined different types of components, including web modules, data sources and EJBs, each with different types of local states. For example, the JDBCDataSource class defines the type of data sources, and its currentOpened attribute reveals their connection pool sizes.

We write the access model (Figure 2(c)) by studying the reference of JOnAS JMX API, as well as the sample code using this API. This snapshot shows part of this access model: the class maps defining how to manipulate different kinds of components. Take the class map of JDBCDataSource as an example, we define the Logics for how to create and destroy a data source, and how to get and set its attributes. Under these logics, we define the Java code invoking the management API. For example, the pop-up dialog shows the code under the Get logic of JDBCMaxConnPool. These lines of code invoke the getAttribute method of JMX API.

Finally, we defined a QVT transformation to connect the architecture and system meta-models. We use five QVT relations to reflect all types of management elements to components. Figure 2(d) shows one of these relations, between data sources and components.

Table 1: Quantified work

<table>
<thead>
<tr>
<th>Items</th>
<th>Quantified work</th>
</tr>
</thead>
<tbody>
<tr>
<td>arch meta-model</td>
<td>29 model elements</td>
</tr>
<tr>
<td>sys meta-model</td>
<td>61 model elements</td>
</tr>
<tr>
<td>access model</td>
<td>102 model elements</td>
</tr>
<tr>
<td>wrapped code</td>
<td>237 lines of Java</td>
</tr>
<tr>
<td>relation</td>
<td>157 lines of QVT</td>
</tr>
</tbody>
</table>

Table 1 summarizes our work on this case study. By contrast, our previous implementation of architecture-based JEE management tool has 5,000 lines of code, a case study in Rainbow project costs 102,000 lines of manual code, and a Fractal implementation named Julia contains 28,000 lines of code. Of course, these contrast cases do not have equivalent capabilities as ours, but the code size at least demonstrates that SM@RT tool saves developers’ effort.

From the above specification, SM@RT tool generates the adapter and synchronizer. This generation work is fully automated. We only need to launch a menu item and indicate the files storing this specification. The generated adapter and synchronizer are both in the form of jar files. We can either use the adapter alone to reflect the system as a low-level model, or use the adapter and the synchronizer together to reflect the system as a C2 styled architecture model.

Using the generated adapter

The generated adapter enables us to use standard model-based techniques for managing the systems. For example, we can use the following OCL query to monitor the JOnAS system. This query checks if the total opened connections of all data sources are more than 50.

```
context MBeanServer inv:
  self.jdbcDataSource ->
  sum(e|e.currentOpened) < 50
```

Figure 5 shows how to execute the above OCL rule on JOnAS system, with the help of our adapter. We first initialize an object for the JOnAS server (Lines 1-6), and this object conforms to the MBeanServer class in the system meta-model(2(b)). Then we initialize the Eclipse OCL engine (Lines 8-11), and evaluate the OCL rule (Lines 13-20). Since server conforms to the EMF common root class,EObject (according to the class hierarchy in the generated adapter), the Eclipse OCL engine could directly accept server as a parameter for query evaluation.

Note that during this process, we only initialize an object for the JOnAS server, and during the query evaluation, the adapter automatically constructs the model from the running system.

Using the generated synchronizer

The synchronizer enables us to apply a typical architecture-based management approach on this JOnAS system. Different from the usage of adapter, we deploy the synchronizer as an Eclipse plug-in, and use a GUI based way to initialize and launch it.

Our target system is a Java Pet Store (JPS) application running on a JOnAS server, and Figure 6(a) shows the home page of this web-based application. To illustrate the scenario more intuitively, we employed a graphical editor for C2 styled architecture models, as shown in Figure 3. Before performing the management scenario, we first configure the synchronizer, using the dialog as shown in Figure 7. We provide an Eclipse project to store the relevant files, and indicate the meta-models and the QVT transformation.

After the above preparation, we execute the "synchronize" command for the first time. The synchronizer constructs an
Following the typical architecture-based evolution approach, we add RSS capability into JPS system to support subscription of pet information. We perform this scenario in the following steps. 1) We add ProductArtist and ItemArtist components for organizing the raw data as products (a product represents a pet breed) and items (same breed of pets from different sellers are regarded as different items), respectively, and add the //localhost/rss component for formatting the data as a RSS seed. These new components are shown inside the red box in Figure 3. All these components have their implementations in the form of EJB or web module. 2) We launch the synchronizer, which automatically deploys the EJ Bs and the web module onto the JOnAS server. Now we can subscribe a RSS seed with all items via the same URL (Figure 6(c)).

The performance of this synchronizer is not ideal, but enough for human participant runtime management. We run the above scenarios on a laptop computer with an Intel Pentium 1.6GHz process and 1.5GB memory. All synchronization processes lasts between 2 to 5 seconds. In particular, adding the RSS components takes 3.32 seconds in average. Since the default web-based JOnAS management tool also does not show the RSS component in the architecture model, and synchronizes it with the running JOnAS system, the performance decreases. The response time of this synchronizer is tolerable by JEE administrators.

Other case studies

We have also performed several other case studies, and Table 2 summaries the system, architecture style (for case 4 and 5, we only generated the adapter, and thus did not employ architecture styles) and specification size of each case. For the first two cases, we also list the code size of contrast cases developed by ourselves (case 1) or other researchers (case 2).

<table>
<thead>
<tr>
<th>#</th>
<th>system</th>
<th>arch style</th>
<th>model (elements)</th>
<th>code (LOC)</th>
<th>contrast (LOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JOnAS</td>
<td>Client/Server</td>
<td>192</td>
<td>237</td>
<td>5294</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>class diagram</td>
<td>91</td>
<td>124</td>
<td>3108</td>
</tr>
<tr>
<td>3</td>
<td>PLASTIC</td>
<td></td>
<td>67</td>
<td>70</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>Eclipse GUI</td>
<td></td>
<td>-</td>
<td>86</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>Android</td>
<td></td>
<td>-</td>
<td>84</td>
<td>47</td>
</tr>
</tbody>
</table>

Case 2 is a reproduction of an existing tool named Jar2UML. The generated adapter reflects the class structure in a Jar file as a UML model. Case 3 is undertaken by our colleague, Xingrui Shen. He wrapped the PLASTIC management functions and reflect the mobile system as a Client/Server styled architecture model. Case 4 is an illustrated example used in our tutorial. We provide a tree-based model editor to achieve dynamic configuration of an Eclipse window, like changing a button’s caption or a label’s background color. For case 5, we defined and generated an adapter to reflect the system state and package structure of Google Android platform. We linked the adapter with an extended OCL engine, and write a set of extended OCL rules (with the ability for assignment) to achieve self-adaptation on the mobile phone. For example, one of the rules specified that “if there is no available Wi-Fi service, then disable the Wi-Fi related components”. We successfully deployed and executed this case on the HTC G1 mobile phone.