

Generating Synchronization Engines between Running Systems and Their Model-Based Views

Hui Song¹, Yingfei Xiong^{1,2}, Franck Chauvel¹, Gang Huang^{1,*}, Zhenjiang Hu³,
and Hong Mei¹

¹ Key Laboratory of High Confidence Software Technologies (Ministry of Education)
Peking University, Beijing, China

{songhui06,franck.chauvel,huanggang,meih}@sei.pku.edu.cn

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

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

Abstract. The key point to leverage model-based techniques on runtime system management is to ensure the correct synchronization between the running system and its model-based view. In this paper, we present a generative approach, and the supporting tool, to make systematic the development of synchronization engines between running systems and models. We require developers to specify “what kinds of elements to manage” as a MOF meta-model and “how to manipulate those elements using the system’s management API” as a so-called access model. From these two specifications, our SM@RT tool automatically generates the synchronization engine to reflect the running system as a MOF-compliant model. We have applied this approach on several practical systems, including the JOnAS JEE server.

1 Introduction

The increasing need of continuously available systems (IT systems, e-business, or critical systems) requires to perform management activities such as configuration, evolution or corrective maintenance *at runtime*.

Management activities (automated or not) are build on a loop [1]: *monitoring* the running system, *analyzing* the collected data, *planning* the needed reconfigurations, and *executing* those reconfigurations. For monitoring and execution, existing platforms such as JEE [2], Fractal [3], and Android [4] provide adequate facilities through devoted APIs, such as the JMX API [5] for JEE systems. For analysis and planning, researchers proposed many generic approaches, utilizing model-based techniques like architecture styles [6, 7], model checking [1], model-based self-repair [8], or model-based artificial intelligence [9], etc.

The key-point to leverage model-based analysis and planning at runtime is to obtain a model-based view of a running system and to ensure the proper synchronization between the system and its model-based view.

* Corresponding author.

However, despite their importance, such synchronization engines are still hand-crafted in a tedious and error-prone manner. Existing approaches [7,10,8,11] include hand-written synchronization engines. To do so, developers have to care about how to maintain a model, how to manipulate the system through the management API, and how to propagate the changes between model and system to ensure their consistency. All these functionalities have to be considered simultaneously.

The contribution of this paper is to make systematic the development of such synchronization engines between models and running systems. Our approach reflects a simple model-driven process: For a specific system, we require the developers to specify *what elements can be managed*, and *how to manipulate them* through the management API. From these two specifications, our approach automatically generates a synchronization engine that maintains a dynamic MOF-compliant model for the running system. This enables the standard model-based techniques (like OCL and QVT) to be used for runtime management. We implement this approach as a tool named SM@RT¹, and apply it on several practical systems.

The rest of this paper is organized as follows. Section 2 illustrates the difficulty for developing a synchronization engine by hand whereas Section 3 presents an overview of our generation process. Section 4 and Section 5 explain how developers specify the system and how to generate the synchronization engine. Section 6 describes and discusses our case studies. Finally, Section 7 presents some related approaches and Section 8 concludes the paper.

2 Motivating Example

This section illustrates the complexity of developing a synchronization engine (SE) between a running systems and its model view.

We choose the JOnAS [12] JEE application server as a running example. A JOnAS server contains a lot of manageable elements such as EJBs, data sources (proxies to databases), etc. Each data source maintains a pool of connections to the underlying database. If the number of cached connections tends to reach the capacity of the connection pool, the database access may be delayed and the pool capacity must be enlarged. In the same manner, if the number of cached connections is always zero, the data source can be removed to release resources.

JOnAS provides an interface (the JMX [5] management API) for the monitor and execution of manageable elements. But complex analysis and planning must still be performed by hand or by using external tools. Model-driven techniques and tools can help such analysis and control tasks. Take the above management scenario as an example, the administrators could use a model visualization tool (like GMF [13]) to help better understand the system, or use OCL constraints to automatically verify the server reconfiguration.

Like other model-based technologies, GMF and OCL can only be applied on MOF-compliant models (as shown in Figure 1), which are constituted by

¹ “SM@RT” for Supporting Models at Run-Time:

<http://code.google.com/p/smatrt>

standard model elements. But the JMX API represents the running system as a specific kind of Java objects, the Managed Beans (MBeans). The integration of model-based techniques thus requires an SE which reflects the running system into a MOF-compliant model, and ensures a bidirectional consistency between the system and the model. For instance, in our JEE scenario, the SE must build a model element for each data source on the JEE AS. When the management agent deletes a model element, the SE must detect this change, identify which data source this removed element represents, and finally invoke the JMX API to remove this data source.

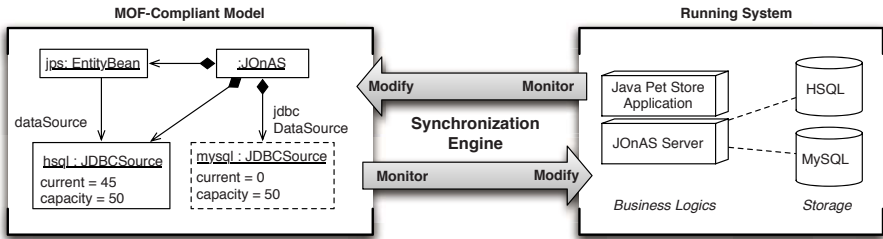


Fig. 1. A common structure of the synchronization engines

However, such an SE employs a complex mechanism and its development is therefore time consuming and error-prone. For the above scenario, the SE has to perform many functionalities: reading and writing models, monitoring and executing system changes, maintaining the mapping between model elements and system elements, handling conflicts between changes, and planning the proper subsequent changes to keep the model and system consistent. In addition, SEs share many commonalities, and developing the SE from scratch is a waste of time and labor. Actually, except for monitoring and executing system changes, all the other functionalities are independent to any specific systems, and thus it is possible to achieve common solutions for them.

3 Approach Overview

We provide a generative approach to assist the development of synchronization engines. As shown in Figure 2, the inputs of our approach include a *system meta-model* specifying what kinds of elements can be managed and an *Access Model* specifying how to use the API to monitor and modify those manageable elements. Our SM@RT tool generates a SE which reflects automatically the running system into a MOF-compliant model that conforms to the system meta-model.

Our approach is applicable under the following conditions. First, we require the target system to provide a management API: our tool does not instrument non-manageable systems, nor extends inadequate APIs. Second, we reflect a *direct model* for the system (that means the model is homogeneous with the

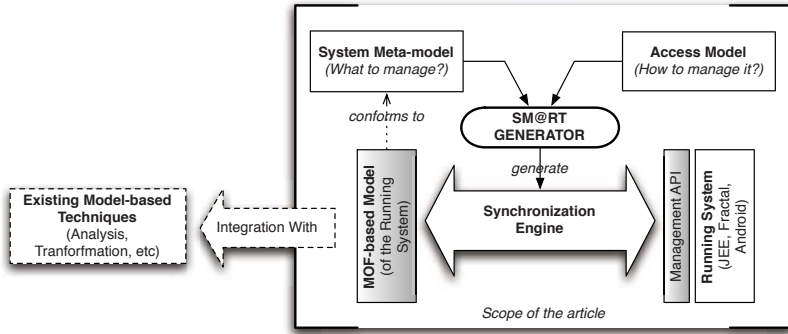


Fig. 2. Generation of Synchronization Engine: Approach Overview

system structure: each model element stands for one system element). If an abstract model is needed, a model transformation could be used to transform the abstract model to a direct model. Synchronisation with abstract models are considered beyond the scope of this paper.

4 Modeling Management APIs

In order to generate an SE for a specific system, we need to know *what can be managed in this system*, and *how to manage it*. In this section, we present how to specify these two kinds of information as models.

4.1 Modeling “What Can Be Managed”

According to Sicard et al. [8], a manageable running system is constituted of *managed elements* (like the JEE server and data sources). Managed elements have *local states* (like the pool sizes of data sources). They could be *composed* by other managed elements (a root JEE server contains several EJBs), and they could have *connections* between each other (an EJB requires data from a data source). These concepts can be described using the concepts of eMOF meta-models, as shown in Table 1.

Following the above guidance, developers can specify the types of managed elements as an eMOF meta-model, and we refer to it as a “system meta-model”. For

Table 1. Concept mapping

<i>concepts of system structure</i>	<i>concepts of MOF Model</i>
Element type	Class
Local state	Attribute
Composition	Aggregation
Connection	Association

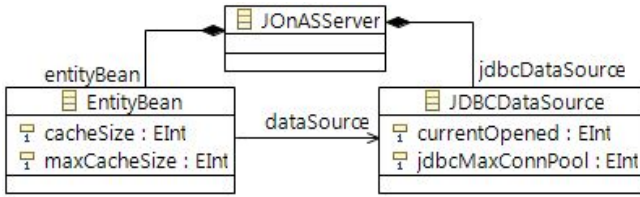


Fig. 3. The system meta-model of the JEE example

example, Figure 3 is an excerpt of the system meta-model we defined for JOnAS. The three classes specify the three types of managed elements. `JDBCDataSource` has two *attributes* representing the currently opened connections and the maximal pool sizes. `JOnASServer` have *aggregations* with the other classes. `EntityBean` have *association* with `JDBCDataSource`. Note that we only showed the necessary part for explaining our sample scenario, and the actual meta-model contains many other classes and properties, as shown in Section 6.

4.2 Modeling “How to Manage the Systems”

The manageable elements can be manipulated through the management API. For example, we can use the `getAttribute` method of JMX to fetch the current opened connections of a data source. For a system, we require developers to specify how to invoke its management API to manipulate each type of elements, and we name this as an “access model”. More formally, the access model is a partial function:

$$\text{access} : \text{MetaElement} \times \text{Manipulation} \longrightarrow \text{Code}$$

where *MetaElement* is the set of all the elements in the system meta-model (classes, attributes, etc.), *Manipulation* is the set of all types of manipulations, which are summarized in Table 2, and *Code* is a piece of Java code.

Figure 4 shows two sample items in the access model for JOnAS. The first item defines the common code for *getting* the values of *int-typed attributes*. We obtain an instance of an MEJB remote element (Line 4), and the logic is defined as an *Auxiliary* operation. Then we invoke the `getAttribute` method. The first parameter is the reference to the current management element. The second parameter is the property name. The second sample is used to add a new data source into a server, or “loading a data source according to a specific name” in the JEE language. We first find the model element added by management agents, and get the data source name (Line 11) from this element. Finally we use this name to invoke the `loadDataSource` operation (Lines 12-15).

When defining how to manipulate the systems, developers may need the following kinds of information: They need system information like “what is the current system element” (Line 5); They need system type information, like the property name, (Line 5). They also need the information inputted by the external management agent, like the appointed name for the to-be-created data source (Line 11), and such information is preserved in the corresponding model

Table 2. All kinds of manipulations. For each kind of operation, we list its *name*, the types of *meta elements* it could be applied, the *parameters* it requires for execution, and a brief *description*. In the table, *Property* standards for attribute, aggregation and association. The “1” or “*” marks following them means single-valued or multi-valued properties, respectively. The *Auxiliarys* are common operations defined by users, and can be used during the definition of code, as shown in the example.

name	meta element	parameter	description
Get	Property (1)	-	get the value of the property
Set	Property (1)	newValue	set the property as newValue
List	Property (*)	-	get a list of values of this property
Add	Property (*)	toAdd	add toAdd into the value list of this property
Remove	Property (*)	toRemove	remove toRemove from the list of this property
Lookfor	Class	condition	find an element according to condition
Identify	Class	other	check if this element equals to other
Auxiliary	Package	-	user-defined auxiliary operations

```

1 //Sample 1, get the value for any kind of attributes
2 MetaElement=AnyClass::AnyIntTypedSingleValuedAttribute,
3 Manipulation=Get, Code=BEGIN
4   Management mgmt=$sys::auxiliary.getMainEntry();
5   Integer res=(Integer) mgmt.getAttribute($sys::this,
6     $meta::prpt_name);
7   $sys::result=rest.intValue();
8 END
9 //Sample 2, add a new data source
10 MetaElement=JOnASServer::jdbcDataSource, Manipulation=Add
11 Code: BEGIN
12   String dbName=$model::newAdded.name;
13   Object[] para = {dbName, Boolean.TRUE};
14   String[] sig = {"java.lang.String", "java.lang.Boolean"};
15   Management mgmt=$model::auxiliary.getMainEntry();
16   $sys::newAdded=(ObjectName)mgmt.invoke(dbserver,
17     "loadDataSource", para, sig);
18 END

```

Fig. 4. Invoking JMX interface

element. We defined three kinds of *specification variables*, the *system*, *meta* and *model* variables, to stand for the above three kinds of information, in order to keep developers from the details about the generation and the SE.

5 Generating the Synchronization Engine

This section presents the SEs we generated to maintain the causal links between model and system. We first explain how the generated SEs work, and then introduce how we generate the engines.

5.1 The Synchronization Engine

The first question for a synchronization mechanism is “*when* and *where* to synchronize”. Since the model is the interaction point between the system and the management agent (MA), the synchronization should be triggered *before MA read the model* and *after they write the model*. In addition, for each reading or writing, the MA only cares about parts of the model. And thus, we only synchronize the involved part of the model with the running system.

Figure 5 shows the structure of our SE, implementing the on-demand synchronization approach we discussed before.

The model we provide is in an in-memory form conforming with Ecore [13]. Each model element is represented by a Java object in the type of `EObject`. External management agents read or write this model by invoking the standard get or set methods on these model elements.

The **Model Listener** listens to these model operations. For a reading operation, the listener interrupts the operation, asks the planner to perform the synchronization, and finally resumes the original operation with the refreshed model. For a writing operation, it waits until the operation is finished, and asks the planner to synchronize this modified model with the system.

The **Mapping pool** maintains a one-to-one mapping between the model elements and the system elements, as a reference for the synchronization. This mapping is implemented as a hash map between model elements and the references to the system elements, and is dynamically maintained as the model and system evolves.

The **Model and System proxies** are used to read the current model and system, and write the required changes (i.e. the synchronization result) back. The model proxy is implemented by invoking the standard get or set methods on the model elements, while the system proxy is implemented by the API invocations specified in the access model.

The **Exception Catcher** implements a simple conflict handling strategy, i.e. when a conflict causes failures during the model or system manipulation, it catches the thrown exceptions and warns the management agent. Developers could implement more sophisticated strategies by defining complex code in the access model, but currently we do not provide special support on that.

Based on these auxiliary parts, the central **Planner** execute a set of synchronization strategies:

$$SynchStrategy : ModOp \times MOFElem \rightarrow (ModOp \cup SysOp \cup MapOp)^*$$

Each strategy defines that when a specific kind of model operations (get, set, etc.) happened on a specific part of the model (model elements, single-valued attributes, etc.), the engine will execute a *sequence of operations*. These operations manipulate the model, the system, and the mapping pool, in order to make them consistent.

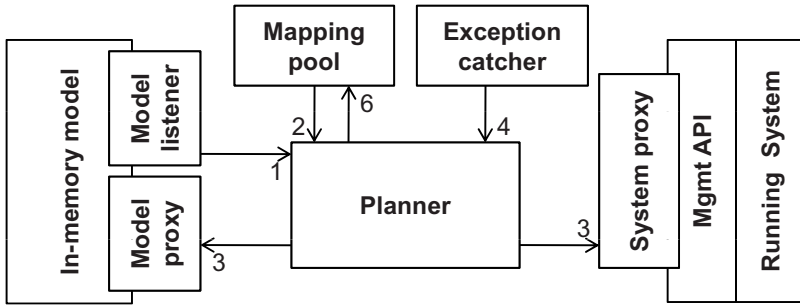


Fig. 5. Structure of the generated SE

```

ModOp : Get
MOFElem: Aggregation(upper-bond>1)
Parameter: mself: the current model element
             pname: the name of the aggregation

sself ← ImagePool.getSys(mself)
schildren ← sList(sself, pname)
foreach schild ∈ schildren do
  if ImagePool.getModel(schild) = Null then
    | mclass ← GetClass(mself.class,pname,sself)
    | mchild ← mCreate(mclass, mself, schild)
    | ImagePool.put(mchild, schild)
  end
  else
    | mchild ← ImagePool.getModel(schild)
  end
  mAdd(mself, pname, mchild)
end

```

Algorithm 1. Sync strategy for getting multi-valued aggregations

Algorithm 1 illustrates one of the synchronization strategies² we have defined and hard-coded inside the planner. This strategy is corresponding to the “Get” operations on the multi-valued aggregations in the model. The variable starting with “m” are elements from the model while the ones starting with “s” are elements in the system. Similarly, the functions starting with “m” (like **mCreate**) are standard model manipulations executed through the **model proxy**, while the ones with “s” are the manipulations on systems executed through the **system proxy**, which are exactly the ones defined in the access model (see Table 2). When informed by the model listener that someone wants to get the children of a model element *mself*, the planner first gets the corresponding system elements *sself*, and then gets the children of this *sself*. For each of the obtained children

² Strictly speaking, such strategies are not “algorithms”, because they do not have inputs and outputs: They just tell the planner what to do on a model operation.

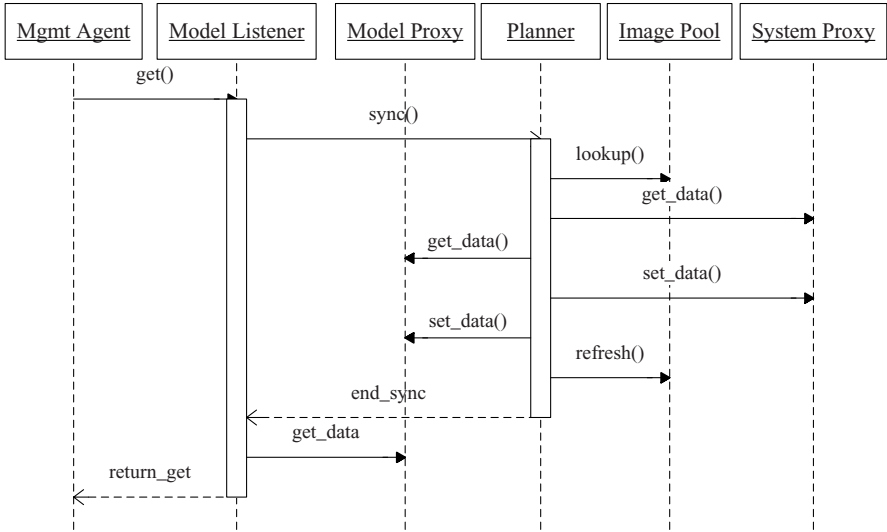


Fig. 6. The collaboration inside an generation engine

schildren, it creates a new model element as its image, and refresh the model with these images as the original *mself*'s children.

Figure 6 illustrates how the parts inside an SE work together. We do not show the “Exception Catcher” here because it only appears when this sequence fails. For the JOnAS sample, in the beginning, the model contains only one element standing for the JOnAS server. The management agent invokes `get` method on this root element to see its data sources. The model listener interrupts this `get` operation, and informs the planner. Follow the synchronization strategy for `get` operations on *multi-valued aggregations*, the planner performs the following operations: It first checks the mapping pool to see that `root` stands for the JOnAS server, and then invokes `list` on this server (See Table 2), which returns a set of `ObjectNames` pointing to the current data sources. The planner then invokes the `create` operation on the model proxy to create a new model element for each of these data sources, and refreshes the mapping pool for these new model elements. Finally, the original `get` operation continues, and returns a set of newly created model elements.

5.2 The Generation Tool for the Synchronization Engines

Our SM@RT tool automatically generates the above SEs. The tool has two parts, a common library and a code generation engine. The common library implements the mapping pool, the exception catcher, and the planner, with the synchronization strategies hard-coded inside. The code generation engine is an extension of the Eclipse Modeling Framework (EMF), and it generates the model listener, model proxy, and system proxy specific to the target system. Specifically,

it generates a Java class for each of the MOF classes in the system meta-model, implementing the `EObject` interface defined by Ecore. Then it overrides the model processing methods in `EObject`, inserting the logic for listening operations and launching the synchronization planner. Finally, it wraps the pieces of API invocation code in the access model into a set of system manipulation methods, which constitutes the system proxy.

6 Case Studies

We applied our SM@RT tool to generate SEs for several practical systems, and performed several runtime management scenarios on these models, utilizing existing MOF-based model-driven techniques.

6.1 Reflecting JOnAS JEE Systems

Our first case study is the full version of the running example we used before. We reflect all the 21 kinds of JEE manageable elements (including applications, EJBs, data sources, transaction services, etc.) as a MOF-compliant model, and visualize it to provide a graphical management tool for JOnAS administrators.

We first define the system meta-model and the access model for JOnAS as explained in the previous sections. The resulting system meta-model contains 26 classes, 249 attributes, 21 aggregations and 9 associations. The resulting access model defines 28 pieces of code like the sample in Figure 4.

From the system meta-model and the access model, the SM@RT tool automatically generates the SE for JOnAS as a Java library. We connected this library with GMF to visualize the reflected model (just in the same way as visualizing any common Ecore models), as shown in Figure 7.

In this snapshot, the rectangles stand for the JOnAS manageable elements and the lines stand for the association between these elements. From this diagram, we see that there are two applications running on the `pku` server, which runs on one JVM, and contains several resources, including a data source named `HSQL1`. We select the data source, and the *property view* on the right side shows its attribute values. All the elements, associations and attributes depict the *current* system state. That means if we select this model element again (that causes GMF to refresh the attributes), some attribute values may change, and if we select the canvas (that causes GMF to refresh the root element), some elements may disappear and new elements may appear. We can also directly use this diagram to change the system. For example, if we increase the `JDBCMaxConnPool` from 100 to 200, the underlying pool will be enlarged consequently. If we create a new model element in the type of `J2EE Application`, and set its `fileName` attribute as the address of an EAR file, the synchronization engine deploys this EAR file into the system, and some new model elements will appear in the diagram, standing for the modules and EJBs inside this newly-added application.

We use the following two experiments to demonstrate that the generated SE reflects a *valid* and *useful* runtime model for the JEE system.

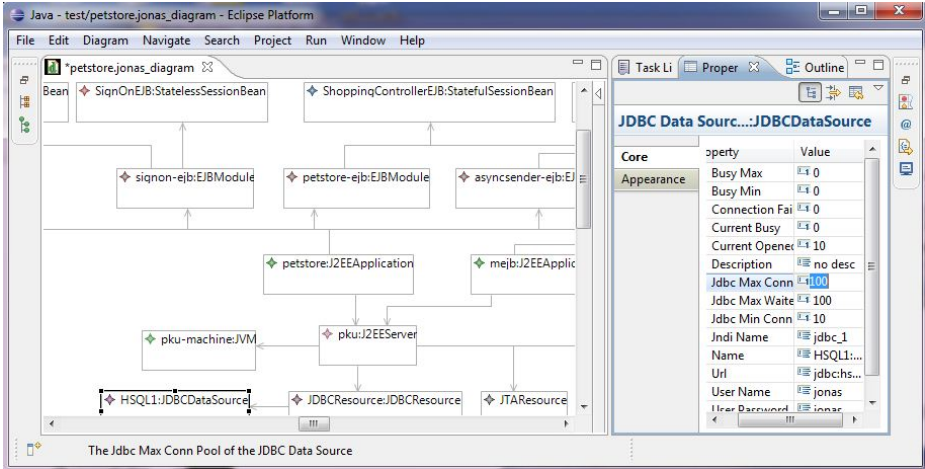


Fig. 7. A snapshot of the visualized model of JOnAS

The first experiment is about monitoring and altering attribute values of system elements. We wrote an automated script to continuously request the `SignOn` component. When executing this script, we noticed that the `currentOpened` connections of `HSQL1` was also growing, until it reaches 50, the value of the `jdbcMaxConn`. That means the data source's connection pool is full. Then we change the `jdbcMaxConn` to 100, and the `currentOpened` began to grow immediately. That means the database's maximal pool size has been successfully enlarged. After that we set `Max Pool` to 20000, but after synchronization, this value becomes 9999 (the upper limit of connection pool supported by this version of `HSQL`), and we receive a warning.

Our second experiment is about changing the structure of system elements. We choose a typical scenario to add `RSS` (Really Simple Syndication) capability into `JPS` at runtime to support subscription of pet information. We added a `StatelessSessionBean` into the diagram, which organizes the pet information as *products* (a product represents a pet breed [14]). We also add a `WebModule` for formatting the data as an `RSS` seed. We implemented the `EJB` and the web module, packaged them as `Jar` and `War` files, and set the paths of these file as the `fileName` attributes of the new added model elements. Then we connected them with the `pku J2EE` server, and connect these components together. After that, we got the `RSS` seed using a browser with item information. That means the components have been deployed into the server by our `SE`.

6.2 Other Case Studies

Table 3 summarizes all the case studies we have undertaken. For each case, we give the target system and its management API, the number of elements in the system meta-model, the items in the access model and the total lines of code in these items. After that, we list the sizes of the generated synchronization engines.

Table 3. Summary of experiments

target system	API	m-model (elements)	access model (items) (LOC)		generated (LOC)	contrast (LOC)	techs
JOnAS	JMX	305	28	310	18263	5294	GMF
Java classes	BCEL	29	13	124	10518	3108	UML2
Eclipse GUI	SWT	43	23	178	11290	-	EMF
Android	Android	29	9	67	8732	-	OCL

For the first two cases, we also list the size of the hand-written synchronization engines for comparison. Finally, we list the model-driven techniques we applied upon the generated SEs.

The second case is a reproduction of the Jar2UML tool³, which reflects the class structure in a Jar file as a UML model. We utilized the UML meta-model (defined by Eclipse UML2⁴) as our system meta-model, and defined an access model, wrapping the invocation to BCEL library⁵ for analyzing Java binary code. We used Eclipse UML2 tools to visualize the reflected UML model as a class diagram.

The third case is an illustrative example. We wrapped the management functions of SWT (the GUI system of Eclipse), and used the tree-based model editor to achieve dynamic configuration of an Eclipse window, like changing a button's caption or a label's background color. A step-by-step presentation of this case study can be found in a tutorial on our project web-site.

Our fourth case is to reflect the packages in an Android⁶ system as a model. We wrote a simple extended OCL rule (we extend OCL to support value assignment, as presented in our previous work [15]) stating that when there is no Wi-Fi service, the Wi-Fi based packages should be stopped. We successfully deployed the generated adapter into an HTC G2 mobile phone, along with the extended OCL interpreter. The rule works well: When the phone is out of the Wi-Fi range, the packages are stopped.

6.3 Discussion

Feasibility: The case studies above illustrate the feasibility of our approach: it generates SEs for a wide range of systems, and the generated SEs enable existing model-driven techniques for runtime management.

Generation Benefits: Our generation approach improves the development efficiency of SEs. Among the complex functionalities of SEs (see Section 2), we only require developers to care about the monitoring and controlling of the system. Specifically, we reduce 94.1% hand-written code for the JOnAS case (310 vs. 5294 LOC), and 98% for the Java case (62 vs. 3108 LOC).

³ <http://ssel.vub.ac.be/ssel/research/mdd/jar2uml>, a use case of MoDisco [11]

⁴ <http://www.eclipse.org/uml2>

⁵ Byte Code Engineering Library, <http://jakarta.apache.org/bcel/>

⁶ <http://www.android.com>

Synchronization Performance: The performance of the generated SE is acceptable for human-based management. For the JOnAS case, we deploy the JOnAS server and the synchronization engine on a personal computer with 3.0GHz CPU and 2.0GB memory. We spend 3.17 seconds in average to show the diagram shown in Figure 7, with 98 manageable elements in total, and we spend less than one second to refresh an element or change an attribute. The performance is similar to the default web-based management tool, the JOnAS Admin. For the Android case, we spend 1.7 seconds to perform the OCL adaptation rule.

7 Related Work

Many researchers are interested on model-based runtime management. The representative approaches include “runtime software architecture” [6, 16], “models at runtime” [17], etc. Currently, these approaches focus on the problems and ideas of model-based management, and implement their ideas on specific systems and models. By contrast, we focus on the reflection of models for different systems, and provide automated support.

Some researchers also focus on reflecting different systems into standard models. Sicard et al. [8] employ “wrappers” to reflect systems states into Fractal models. Researchers of MoDisco Project [11] focus on developing “discoverers” to discover MOF-compliant models from systems. The “wrappers” and “discoverers” are similar to our SEs, but our work supports developers in constructing SEs from a higher level, not by directly writing code in ordinary programming language. Another major difference between our work and MoDisco is that our SEs support writing the model changes back to the system.

Currently, many approaches employ many different mechanisms to maintain the causal connections between models and running systems. First, some early approaches require the target systems to be developed under some specific techniques which has built-in RSA support. For example, Oreizy et al. [6] require their target systems to developed under the Java-C2 class framework, and to use Fractal architecture at runtime [18], the system classes must implement some interfaces defined by Fractal. This requirement limits their applicability in practice. Second, some approaches allow the target systems to be developed under industrial standards, but enhance their runtime platforms (middlewares) with RSA mechanisms. These approaches are also known as “reflective middleware”, and covers many mainstream component models, like OpenORB [19] for CORBA, and PKUAS [16] for JEE. The problem of these approaches is that their middlewares are still not widely adopted in practice. Third, some researchers try to inject probes and effectors into existing systems to collect runtime data, organize them as model, and perform model modifications [7]. But since most existing systems are not designed for code-level evolution, injecting code into them, if possible, is usually tedious and unsafe. Our approach is close to the third type in that we also seek to provide a generic mechanism for existing systems, but we choose a more safe way, utilizing the low-level management APIs provided by the existing systems.

Our language for modeling management APIs shares the similar idea as the approaches towards feature-based code composition [20]. They allow developers to model code slices as features, and to compose or customize features at a high-level. From the feature composition, their generator composes the code slices behind the features into an initial version of the program.

Our synchronization mechanism is related to the approaches on model synchronization [21, 22]. The difference is that these approaches use the same model processing interface to manipulate the two participants of synchronization, but we try to integrate ad hoc management APIs into the synchronization process.

8 Conclusion

To efficiently leverage the use of model-based techniques at runtime, it is necessary to have a model-based view of the running system. In this paper, we report our initial attempt towards the automated generation of synchronization engines that reflect running systems into model-based views. We require developer to specify “what to manage on the system” as a MOF meta-model, and specific “how to use the related API to do so” as an access model. From these specifications, we automatically generate the synchronization engine that reflects the system as a direct MOF compliant model. We have successfully applied our approach on several practical systems, and enabled several typical model-based techniques at runtime. As future work, we plan to give more support for developers to specify the running systems and their APIs. We also plan to perform further analysis such as model checking to ensure a deeper correctness and completeness of the generated causal link.

Acknowledgment

This work is sponsored by the National Key Basic Research and Development Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60821003, 60873060; the National S&T Major Project under Grant No. 2009ZX01043-002-002; and the EU FP7 under Grant No. 231167.

References

1. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: Future of Software Engineering (FOSE) in International Conference on Software Engineering, pp. 259–268 (2007)
2. Shannon, B.: Java Platform, Enterprise Edition 5, Specifications (April 2006)
3. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.: The Fractal Component Model and its Support in Java. *Software Practice and Experience* 36(11-12), 1257–1284 (2006)
4. DiMarzio, J.: *Android: A Programmers Guide*. McGraw-Hill Osborne Media (2008)
5. Hanson, J.: *Pro JMX: Java Management Extensions* (2004)

6. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE, pp. 177–186 (1998)
7. Garlan, D., Cheng, S., Huang, A., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37(10), 46–54 (2004)
8. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: ICSE, pp. 101–110 (2008)
9. Chauvel, F., Barais, O., Borne, I., Jézéquel, J.M.: Composition of qualitative adaptation policies. In: Automated Software Engineering Conference (ASE 2008), pp. 455–458 (2008) (Short paper)
10. Batista, T.V., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Morrison, R., Oquendo, F. (eds.) EWSA 2005. LNCS, vol. 3527, pp. 1–17. Springer, Heidelberg (2005)
11. MoDisco Project, <http://www.eclipse.org/gmt/modisco/>
12. OW2 Consortium: JOnAS Project. Java Open Application Server (2008), <http://jonas.objectweb.org>
13. Budinsky, F., Brodsky, S., Merks, E.: Eclipse Modeling Framework. Pearson Education, project address (2003), <http://www.eclipse.org/modeling/emf>
14. Sun: Java PetStore (2002), <http://java.sun.com/developer/releases/petstore/>
15. Song, H., Sun, Y., Zhou, L., Huang, G.: Towards instant automatic model refinement based on OCL. In: APSEC, pp. 167–174 (2007)
16. Huang, G., Mei, H., Yang, F.: Runtime recovery and manipulation of software architecture of component-based systems. *Autom. Softw. Eng.* 13(2), 257–281 (2006)
17. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE) in ICSE 2007, pp. 37–54 (2007)
18. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J., Rhone-Alpes, I.: An Open Component Model and Its Support in Java. In: CBSE, pp. 7–22 (2004)
19. Blair, G., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., Parlavantzas, N.: Reflection, self-awareness and self-healing in OpenORB. In: Proceedings of the first workshop on Self-healing systems, pp. 9–14 (2002)
20. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages with round-trip engineering. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 692–706. Springer, Heidelberg (2006)
21. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 543–557. Springer, Heidelberg (2006)
22. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: ASE, pp. 164–173 (2007)