Applying MDE Tools at Runtime: Experiments upon Runtime Models

Hui Song, Gang Huang *, Franck Chauvel, and Yanchun Sun

Key Lab of High Confidence Software Technologies (Ministry of Education)
School of Electronic Engineering & Computer Science, Peking University, China
{songhui06,huanggang,franck.chauvel,sunyc}@sei.pku.edu.cn

Abstract. Runtime models facilitate the management of running systems in many different ways. One of the advantages of runtime models is that they enable the use of existing MDE tools at runtime to implement common auxiliary activities in runtime management, such as querying, visualization, and transformation. In this tool demonstration paper, we focus on this specific aspect of runtime models. We discuss the requirements of runtime models to enable the use of model-driven tools, and present our tool to help provide such runtime models on the target systems. We apply this tool on a wide range of target systems, modeling the Android mobile system, the Eclipse GUI, the Java class structure, and the JOnAS inner structure. With the help of these runtime models, we perform the runtime management on these systems using classical MDE tools including OCL, QVT, and GMF.

1 Introduction

For a running system, developers often need to retrieve and update its data at runtime. The runtime data depict the system's configuration, structure, state, or environment. By analyzing and changing these runtime data, developers monitor and control the system at runtime to fix system defects, adapt to the changed environment, or meet newly emerged requirements. Take a mobile phone as a sample system, we may care about what wireless network (Wi-Fi) channels are currently available, as well as their signal intensity. We may also need to switch channels when necessary and possible.

* corresponding author

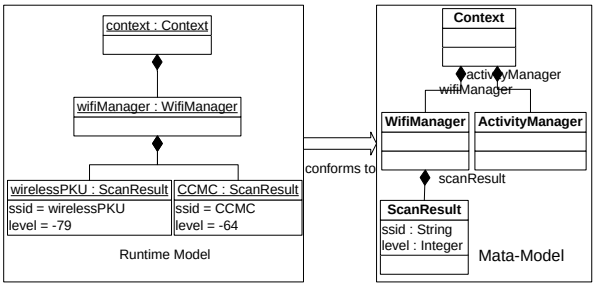


Fig. 1. A runtime model and its meta-model

However, manipulating the runtime data is not an easy task. Currently, most systems only provide low-level APIs for manipulating the runtime data [1], and developers have to write low-level code to invoke the APIs. For example, the code below illustrates how to invoke the Android (a mobile OS) API to print the signal IDs of available Wi-Fi channels.

```
1  WifiManager wm=(WifiManager) this
2      .getSystemService(Context.WIFI_SERVICE);
3  List<ScanResult> srs = wm.getScanResults();
4  for(ScanResult sr in srs)
5    Log.i("Wi-Fi_Signal_ID",sr.ssid);
```

It is tedious and error-prone to manage the system by directly using the management APIs. First, there lacks explicit definition about the data types. Second, there are different invocation manners for different systems or even different types of data inside the same system. Third, people have to re-implement many common auxiliary management activities on each of the APIs, such as querying, aggregation, visualization, etc.

Runtime model [2, 3, 1] provides a promising way to liberate people from the tedious APIs, and allow them to manipulate the runtime data in a higher abstraction level, utilizing the rich and mature MDE (model-driven engineering) techniques and tools, such as OCL for evaluation or querying, QVT for aggregation and analysis, visualization, etc. Figure 1 illustrates a sample runtime model and its meta-model for the Android system. Developers could use the following OCL rule to query the signal IDs of Wi-Fi channels.

```
self.wifiManager.scanResult->collect(e|e.ssid)
```

To enable the application of existing MDE tools, the runtime model should satisfy the following three requirements. Firstly, the model should be organized in a standardized form. Second, the runtime model must have an explicit meta-model which defines its semantics. Thirdly and most importantly, the model must have a *causal connection* with ever-changing system. That means if the system evolves, the model will change immediately, and if the model is modified, the system will change correspondingly.

These requirements call for a software agent to represent the runtime data as a standard model conforming to a specific meta-model, and to synchronize the model with the runtime data. We name such agents as "synchronizers". For a target system, *runtime model providers*, who are experts of the system and its API, develop such synchronizers, and *runtime model consumers*, usually the common developers, use the runtime model maintained by the synchronizer to manipulate the runtime data, using the MDE tools. Existing approaches on runtime model usually require runtime model providers to develop such synchronizers *by hand*[4, 1].

In this paper, we demonstrate a generative tool, *SM@RT*[1], which generates synchronizers for a wide class of systems. As shown in Figure 2, for a kind of target systems, like Android, we require the runtime model providers to define a *system meta-model* which specifies the types of the runtime data, and an *access model* which specifies how to manipulate the data through the API. From these two inputs, *SM@RT* automatically generates the synchronizer, which maintains a MOF standard runtime model for a running system instance, and ensures the causal connection between this model and the system's runtime data.

Our contributions can be summarized as follows.

- We propose that runtime models could facilitate the management of systems by enabling the use of existing model-driven techniques at runtime. We also identify the key requirements for such runtime models.
- We provide a generic synchronization solution between runtime models and system data, and based on this solution, we provide a generative tool to construct synchronizers for a wide class of systems.
- We successfully apply this tool on several systems, and undertake several experiments to utilize the provided runtime models for managing the sys-

[1] SM@RT: Supporting Models at Run-Time, the tool and the case studies are available on line: `http://code.google.com/p/smatrt`

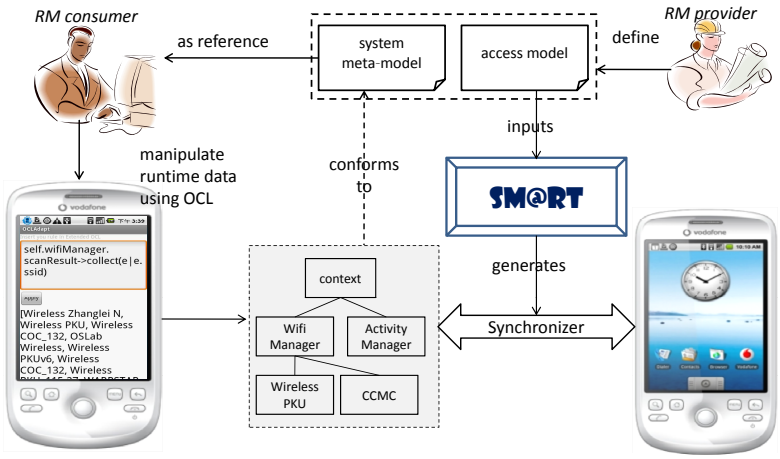


Fig. 2. Tool overview

tems. These case studies illustrate how the runtime models facilitate the management of systems by using model-driven tools, and how our SM@RT tool implement such runtime models.

The rest of the paper is structured as follows. Section 2 discusses the requirements of runtime models. Section 3 presents our SM@RT tool to implement such runtime models. Section 4 reports our case studies. Section 5 presents some related approaches and Section 6 concludes this paper.

2 Requirements of Runtime Models

In this section, we discuss what the runtime model should be like in order to facilitate the system management with the help of MDE techniques and tools. We summarize the following three requirements, considering the feature of both runtime management and the MDE tools.

Standardized. First, the format of the runtime models should conform to some widely accepted modeling framework (the meta-meta-model and the exchange format), such as MOF, fractal, XML, etc. Standardized models provide the runtime model consumers a consistent basis for understanding and manipulating the data. Moreover, since many MDE techniques and tools are defined and implemented on specific modeling standards, they can be directly reused only if the model conforms to the same standard. As the OMG's Meta-Object Facilities

(MOF) has became the most accepted standard, with rich tool support, in this paper we only consider the runtime models conforming to the MOF standard.

Explicitly defined. The types of runtime models should be explicitly defined by meta-models. Such meta-models provide an intuitive guidance and a strict constraint for runtime model consumers to understand and reconfigure the runtime models, and are also necessary reference for MDE tools to process the models. According to the MOF standard, a meta-model defines the types of model elements by the *classes*. For each class, the meta-model defines the data type of attributes that can be contained by the elements, and the potential relation between them and the elements of other classes.

Causally connected. Finally, we require the runtime models to have the *causal connection* with the running systems. The management agents monitor and reconfigure the system by reading and writing the model. The causal connection ensures that each time the management agent reads the model, it gets the information representing the current system state, and similarly, each time it writes the model, the information it writes causes the proper system change. Considering the Android example, if the device enters into the scope of a new Wi-Fi service, there will be a new `ScanResult` element appearing in the model immediately, so that the OCL query in Section 1 returns the ID of the new Wi-Fi service. Causal connection is an important feature of runtime models, which distinguishes them from the models used in design and development phases.

Notice that there are multiple levels for causal connection. The above requirement is just a basic one. Advanced usages of runtime models may require the model changes launched by the management agent would be stable as system evolves, or even require the model to hold some predefined constraints. But in this paper, we cares about the minimal requirement to enable MDE tools to be used for runtime management, and leave the advanced work as the task for "using the tool in a correct way".

3 The *SM@RT* Tool to Implement Runtime Models

We provide a generitive tool, the *SM@RT*, to help implement runtime models that satisfy the above requirements. Specifically, for a target system with a management API, the tool accepts a MOF meta-model defining the system data, and a description about the management API to access such data. Then it

automatically generates a *synchronizer* for the target system, which represents the system data as a MOF standard model conforming the system meta-model, and maintain the causal connection between the model and the system data.

3.1 Tool Input

To provide a runtime model for a specific system, we need the information about "what kind of data can be manipulated" in this system, and "how to manipulate them through the system's API". The former is defined by the MOF meta-model as discussed in Section 2. For the latter, we defined an API description language to specify how to access (invoke) the API to manipulate each type of the data.

The API access is described as code snippets annotated with their effects on the data. Look over the sample code in Section 1 for invoking the Android API. The first line tells us that from the root system element `this`, whose type is `Context`, how we can get its child named `wifiManager`. The above statement comprises three kinds of information for manipulating the system data, i.e., the manipulation target (an aggregation named `Context.wifiManager`), the manipulation type (`get`), and the action (Lines 1-2 in this code snippet). From this point of view, we define the access model for an API as follows.

$$AccItem : MetaElement \times Manipulation \longrightarrow Code$$

Here $MetaElement$ is the set of all the elements in the system meta-model (classes, attributes, etc.), $Manipulation$ is the set of 9 types of manipulations, including `getting` and `setting` attribute values, `creating` and `deleting` model elements, etc., and $Code$ is a piece of Java code [5].

3.2 Tool Output

The output of *SM@RT* is a "synchronizer" that maintains the causal connection between the runtime model and the running system.

The mechanism inside such synchronizers can be briefly described as "lazy and local refreshment". Specifically, the synchronizer maintains an in-memory MOF standard model, in the form of a set of Java objects implementing the `EObject` interface defined in Eclipse EMF. During runtime, the synchronizer keeps on listening to the external reading and writing operations on this runtime model. For reading operation, the synchronizer calculates what system data

are required, collects the data via the management API, and refreshes or complements the model according to the collected data. Similarly, for a writing operation, the synchronizer identifies the modifications on the model, calculates the corresponding changes on the system, and invokes the API to implement the changes. For different kinds of operations (getting, setting, adding etc.) and their target meta-elements (classes, attributes, single or multiple valued associations), the calculation methods are different. We name these methods as the synchronization strategies. We summarized and designed a set of synchronization strategies covering all the potential combinations of operations and meta-elements, as presented in our previous work [5].

3.3 Generating the Synchronizer

SM@RT automatically generates the synchronizers from the API description. The tool has two parts, a *common library* and a *code generation engine*. The common library implements the generic solutions inside the synchronizers, such as maintaining the mapping between model elements and system parts, and the hard-coded synchronization strategies for different kinds of elements and different operations. The code generation engine generates the parts of the synchronizers which are specific to the target system, such as all the standard model operations (depending on the system meta-model) and the effective API invocations to manipulate each kind of system data (depending on the access model). We generate the model operations by directly reusing Eclipse EMF generator, and generate system operations according to the items defined in the access model, using the defined API-invoking code snippets as the body of the system operation. The generated operations follows a strict naming convention, so that the synchronization strategy know the semantical relation between model and system operations, automatically.

4 Demonstration

We demonstrate four case studies for *SM@RT*, using it to provide runtime models for four different target systems, including Android mobile systems, Eclipse SWT windows, Java classes and JOnAS JEE enterprise systems. We describe the Android case in detail, showing how to construct the system-model synchro-

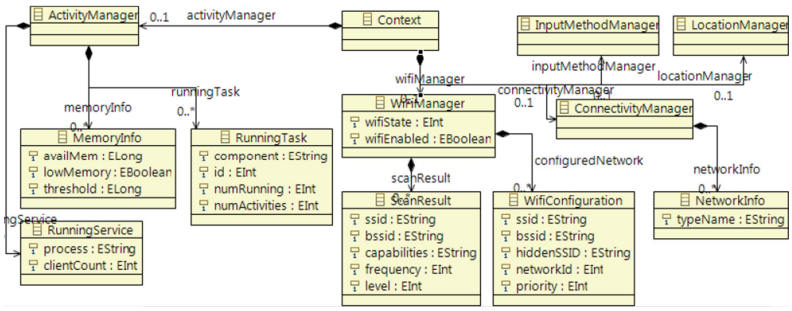


Fig. 3. Tool overview

nizer, how to use the MDE tool (the OCL query engine in this case) upon the runtime model, and how the synchronizer works to maintain the runtime model.

4.1 The Android Case

Android is a mobile operating system developed by Open Handset Alliance[2]. It allows developers to write managed code in Java to manipulate (read or write) a device, by invoking the API of a set of Google-developed Java libraries.

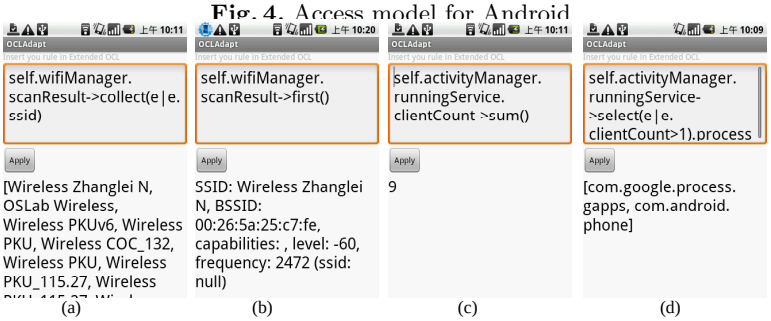
Figure 3 shows the system meta-model we define for Android runtime data. In this demonstration, we care about the memory, connections, running tasks and Wi-Fi. We define each type of system data as a class, and define the relations between them as properties. For example, this meta-model tells us that from a root element in type of `Context`, we can first get its `wifiManager`, and then get the manager's `scanResult` to enumerate all the Wi-Fi signals. For each scanned signal, we can get its attributes like `ssid`, `frequency`, etc. This meta-model is not only an input to our synchronizer, but also a guidance for using the runtime model (like writing the OCL query) and a reference of the MDE tool (like the OCL engine).

Figure 4 shows an excerpt of the access model, which defines how to get a `Context`'s `wifiManager`. The annotations (keywords starting with "@") indicates the constitution of the item, i.e. a `AccItem` containing a `MetaElement`, `Manipulation` and a `Code` fragment, and whose values are defined on the right hand side of the equal signs. Inside the code fragment, we define a piece of Java code to say that to get a `wifiManager` (`$sys::result`) from a `Context` instance

[2] `http://www.android.com`

```
@AccItem @MetaElement=Context::wifiManager
@Manipulation=Get @Code=@Begin
  $sys::result=($sys::type)$sys::this
    .getSystemService($sys::type.WIFI_SERVICE);
@End @EndAccItem
```

Fig. 4. Access model for Android



(a) (b) (c) (d)

Fig. 5. Android screen shots

(`$sys::this`), we should invoke a method named `getSystemService` with a parameter `Context.WIFI_SERVICE`. The entire access model contains 95 items like this, with 431 lines of code (including the structural lines like `"@AccItem"`).

We use these two inputs to generate the synchronizer. The generation result is in the form of Java source code. We compile it as an Android package, and deploy it onto an Android supported mobile phone, the "HTC Magic (G2)".

The generated synchronizer allow the device users to use OCL for querying the device data. Figure 5 shows the snapshots of four scenarios for executing OCL rules on Android. For the first scenario, we want to list the IDs of all the Wi-Fi signals available for the device. Initially, we know that the root element (we refer to it as `self` in the OCL rules) is in type of `Context`. Then we check the system meta-model and find that `Context` has an association named `wifiManager`. The target class `WifiManager` has an multiple-valued association named `scanResult`. And finally, the target class `ScanResult` has an attribute named `ssid`. According to terminology of Wi-Fi technique, we know that we can list the signal IDs by querying out the values of these `ssid`s. We input the OCL rule as shown in Figure 5(a), click the button, and the result is printed under the button. Similarly, Figure 5(b) shows how we print the detailed information of the first Wi-Fi channel. Figure 5(c) shows how we calculate the total number of clients

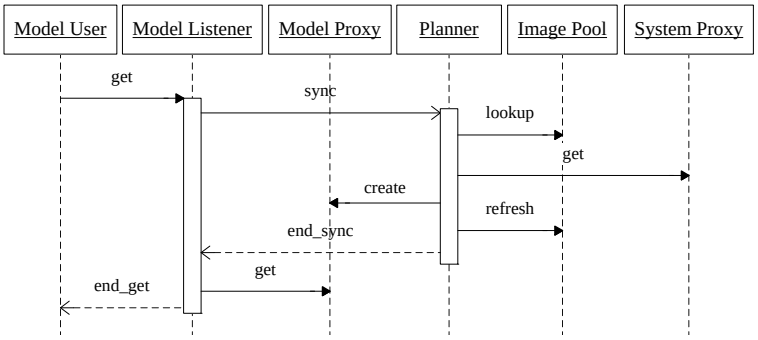


Fig. 6. A sample behavior of the synchronizers

registered on all the running services. Figure 5(d) shows a relatively complex query: We want to see what services have more than one clients listening to them. The OCL rule means "getting the running services, selecting the ones from them whose client count is greater than 1, and finally retrieving the process name of the selected running services".

The above scenarios are implemented by directly using the Eclipse OCL engine. We compile the OCL engine on Android platform, and deploy it on the same device. After the user clicking the `Apply` button, the GUI retrieves the inputted OCL rule, instantiates a root element in type of `Context` from the synchronizer, and invoke the `evaluate` method of the OCL engine using this root element and the OCL rule. During the execution of the OCL engine, it will manipulate the model from this root element, by means of standard model invocation defined by EMF. And in the same time, the synchronizer breaks the invocation, and synchronizes the model with system on-demand.

Figure 6 illustrates how the synchronizer works, when we evaluate the OCL query in Section 1 on the runtime model as shown in Figure 1. Each life-line in this sequence diagram represents a component that constituting the synchronizer. At first, the initial model only contains one root model element, in type of `Context`. Following the query, the interpreter first retrieves the root's child named `wifiManager`, by invoking `get` on the model. The *model listener* interrupts this invocation, and asks the planner to perform synchronization. According to the synchronization strategy for "getting single-valued aggregation" [5], the *planner* first looks up the *image pool* and finds that this root element corresponds to a context object provided by the Android API. Then the *planner*

performs `get` on this context object. The logic of this system `get` operation is just the one defined in the description item shown in Figure 1. This `get` operation returns a system object which points to the Wi-Fi manager, and the *planner* `create`s a model element in type of `WifiManager` as an image for this object, `refresh`es the image pool, and notifies the *model listener* about the `end` of this synchronization. The *model listener* then invokes `get` on the model again, and returns the newly created model element as a result to the interpreter. Following the remaining parts of the OCL query, the interpreter performs `get` operations successively to obtain the `WifiManager`'s `scanResult`, and to obtain these results' `ssid`s. The behavior of the synchronizers is similar as shown before.

4.2 The Eclipse-SWT Case

In this case, our target systems are SWT-based Eclipse UI parts, which could be "views", "editors" or "dialogs" running on an Eclipse platform. Such UI parts, also known as `Shell`s according to the SWT terminology, are constituted of a set of `Control`s, like the `Label`s for presenting information, the `Text` fields for inputting texts, the `Button`s for triggering commands, etc. Each `Control` has its own configurations which can be retrieved and updated at runtime, such as the presented text, the background color, etc. The `Control`s and their configurations form the runtime data of such `Shell`s. The main idea of this case study is to provide runtime models for Eclipse windows, so that developers could reconfigure the windows intuitively *at runtime*. That means developers do not need to completely decide the appearance of the windows, and reconfigure the window at runtime. Moreover, this configuration is simply editing models, through visualized model editors. This is a prototype for "design at runtime", and may be useful in customizable GUIs or WYSIWYG GUI development.

The foreground image of Figure 7 presents a simplified version of the system meta-model. We defined three common types of controls, and defined some typical attributes for them. The background snapshot illustrates how to use the runtime model. The snapshot is an Eclipse platform, with the target system (the bottom part, an Eclipse "view") and the runtime model (the top part, a model opened in a tree-based visual model editor), together. The model elements reflect the controls in the window, and their attributes reflect the controls' configurations. We change the system by typing "Hi" on the text field, the model element's `text` attribute changes instantly. We can edit the model to manipulate

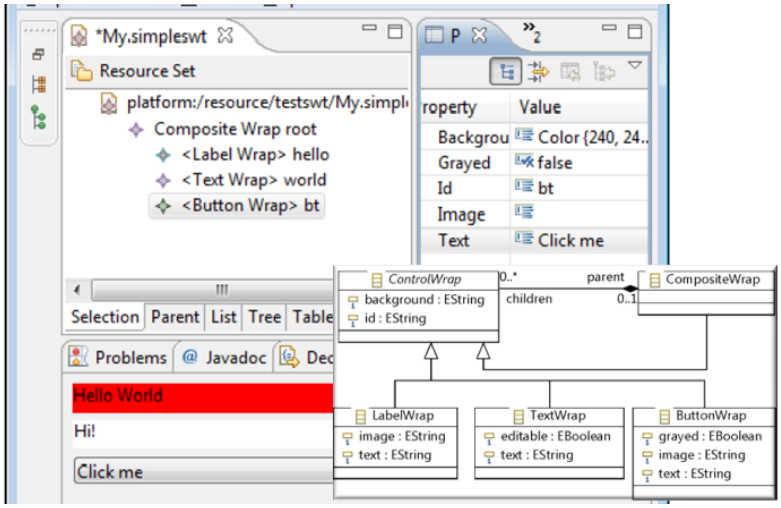


Fig. 7. Visual management of an SWT window

the system: We change the `background` of the first `Label` into "red", and then the color of the system label changes automatically. Finally, we add a new model element in the type of `Button`, and a new button appears in the window .

4.3 The Java Class Structure Case

This case is a reproduction of the Jar2UML tool[3], which reflects the class structure in a Jar file as a UML model. We utilized the UML meta-model (defined by Eclipse UML2[4]) as our system meta-model, and define the API provided by the BCEL library[5] for analyzing Java binary code. We used Eclipse UML2 tools to visualize the reflected UML model as a class diagram.

4.4 The JOnAS Case

Our last case study is to equip a JOnAS JEE application server[6] with runtime model. This model reflects the inner structure (e.g. what applications and EJBs are deployed), the configuration (e.g. the size of data source's connection pool), and the state (e.g. the number of EJB instances) of a running JOnAS server.

[3] `http://ssel.vub.ac.be/ssel/research/mdd/jar2uml`, a use case of MoDisco
[4] `http://www.eclipse.org/uml2`
[5] Byte Code Engineering Library, `http://jakarta.apache.org/bcel/`
[6] `http://jonas.ow2.org`

Table 1. Summary of case studies

system	**API**	**meta-model** *elems*	**access model** *items/LOC*	**generated** *LOC*	**compared** *LOC*	**tools**
Android	Android	87	95/431	21732	-	OCL
Eclipse	SWT	43	36/220	11290	-	EMF
Java class	BCEL	29	13/109	10518	3108	UML2
JOnAS	JMX	305	47/270	37263	5294	GMF, QVT

The system meta-model defines all the 21 types of MBeans supported by JOnAS, including EJBs, applications, middleware services, etc. The API description specifies how to manipulate these elements and their properties through the JMX API provided by JOnAS. We deploy the generated synchronizer on a JOnAS server with a Java Pet Store application deployed on it, and utilize Eclipse GMF[7] to visualize the runtime model maintained by the synchronizer. The graphical model editor based on GMF can be used as a graphical JOnAS management tool: We can see the inner structure of the current JOnAS server, deploy new applications or EJBs by adding model elements, and check the system elements' current states and modify their configurations. We also use the QVT transformation to synchronize this runtime model with a software architecture model in C2 style, reproducing the architecture-based runtime evolution proposed by Oreizy et al. [6]. The details of this case study could be found in our earlier work [7]

4.5 Summary and discussion

Table 1 summarizes the case studies. For each case, we list the target system and its management API, the number of elements in the system meta-model, and the number of items in the API description. After that, we list the size of the generated synchronizer. For the last two cases, we also list the sizes of the hand-written programs with the equivalent capabilities, which are developed by ourselves or other developers [4]. Finally, we list the model-driven techniques we applied upon the runtime model to implement runtime data manipulation.

These case studies illustrate the following aspects of *SM@RT*.

– *Feasibility.* The case studies covers a wide class of systems, from enterprise systems to mobile devices.

[7] `http://www.eclipse.org/modeling/gmf`

– *Efficiency for development.* It is not a hard task to define the system meta-model and API description, comparing with the multi-time-larger generated code (which approximately reflects the work required to support runtime model) and the actual manual effort to realize runtime models.
– *Effectiveness.* The generated synchronizers enable the existing MDE tools to be directly used for runtime management. In particular, we use OCL for runtime data querying, and use GMF/EMF, and UML2 for different purpose of visualization and manipulation of runtime data. All the MDE tools are directly used upon the synchronizers.

5 Related Work

There are many research approaches towards runtime models, according to a recent survey [3] and the annual workshops [2]. As an emerging topic, many of these approaches focus on how to utilize the runtime models, but not how to implement runtime models on existing systems, which is exactly the target of *SM@RT*. We share the similar idea with Sicard et al. [1] and the MoDisco project [4], i.e. wrapping the systems' APIs to reflect runtime data as standard models. Their *wrappers* or *discoverers* play the similar role as our *synchronizers*. The difference is that they require runtime model providers to manually develop the wrappers or discoverers, while our *SM@RT* tool automatically generates the synchronizers. Our API description language shares the similar idea as feature-based code composition [8]. The common synchronization mechanism roots in the earlier research on reflective middleware [9], and the model synchronization approach towards runtime management [10].

6 Conclusion

In this paper, we focus on a specific usage of runtime models, i.e., facilitating the runtime management by enabling the use of existing MDE techniques and tools at runtime. We discuss the requirements of such runtime models, and present our *SM@RT* tool to help on providing them. We evaluate our idea and the tool through a set of case studies on a wide range of target systems.

ACKNOWLEDGEMENT This work is supported by the National Basic Research Program of China under Grant No. 2011CB302604, the National Natural

Science Foundation of China under Grant No. 60933003, 60873060; the High-Tech Research & Development Program of China under Grant No. 2009AA01Z116, and the National S&T Major Project under Grant No. 2009ZX01043-002-002, the EU Seventh Framework Programme under Grant No. 231167.

References

1. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: ICSE. (2008) 101–110
2. Bencomo, N., Blair, G., France, R.: Summary of the workshop Models@run.time at MoDELS 2006. In: Satellite Events at the MoDELS 2006 Conference, LNCS,. (2006) 226–230
3. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering, in ICSE. (2007) 37–54
4. Bruneliere, H.: The MoDisco Project, `http://www.eclipse.org/gmt/modisco/`
5. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating synchronization engines between running systems and their model-based views. In: MoDELS Workshops 2009, LNCS 6002. (2009) 140–154
6. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE. (1998) 177–186
7. Huang, G., Song, H., Mei, H.: Sm@rt: Applying architecture-based runtime management into internetware systems. Int. J. Software and Informatics **3**(4) (2009) 439–464
8. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages with round-trip engineering. In: MoDELS. (2006) 692–706
9. Huang, G., Mei, H., Yang, F.: Runtime recovery and manipulation of software architecture of component-based systems. Auto. Soft. Eng. **13**(2) (2006) 257–281
10. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental model synchronization for efficient run-time monitoring. In: MoDELS Workshops. (2009) 124–139