# SM@RT: Applying Architecture-based Runtime Management into Internetware Systems

Gang Huang, Hui Song, Mei Hong

# SM@RT: Applying Architecture-based Runtime Management into Internetware Systems

Gang Huang, Hui Song, Hong Mei

Key Laboratory of High Confidence Software Technologies, MOE
School of Electronics Engineering and Computer Science,
Peking University, Beijing, 100871, China
huanggang@sei.pku.edu.cn, songhui06@sei.pku.edu.cn, meih@pku.edu.cn

**Abstract** Architecture-based runtime management (ARM) is a promising approach for Internetware systems. The key enablement of ARM is runtime architecture infrastructure (RAI) that maintains the causal connection between runtime systems and architectural models. An RAI is uneasy to implement and, more importantly, specific to the given system and model. In this paper, we propose a model-driven approach for automated generation of RAI implementation. Developers only need to define three MOF models for their preferred architecture model and the target system (these models are reusable independently for different pairs of the model and system), and one QVT transformation for the causal connection. Our Eclipse-based toolset, called SM@RT, will automatically generate the RAI implementation code without any modification on the source code of the target system, and automatically and properly deploy the generated RAI into the distributed systems. This approach is experimented on several runtime systems and architectural models, including ABC architectural models on Eclipse GUI and Android, C2 architectural models on JOnAS, Rainbow C/S style on PLASTIC and UML models on POJO.

**Keywords:** Internetware, Architecture-based Runtime Management, Model Driven Development, Eclipse.

## 1. Introduction

Internetware is a paradigm of the cooperative, situational, emergent, evolvable, autonomous and trusted software systems running on the open, global, ubiquitous and smarter Internet (software of "Internet as a computer"). No matter what functionalities an Internetware system provides, it has to provide management capabilities for achieving promised or desired qualities since the quality has the same and even more importance than the functionality in Internetware. Considering the most important characteristics of Internetware, the management capabilities can be divided into: 1) Self-management means a system is capable of managing itself without intervention external of the system, i.e. does not control its external and does not allow its external to control itself); 2) Co-management means a system is capable of managing with intervention external of the system, i.e. can control or be controlled by its external. Almost all today's software systems support co-management, that is, only provide human or agent administrators with management interfaces and configuration files. Autonomic systems are typical ones that support self-management. Self-management of a system can be implemented by co-management of its sub systems, while co-management of a system can utilize self-management of other systems.

Due to the autonomous characteristic of Internetware and the cost of management, an Internetware system should support self-management capabilities as many as possible. Meanwhile, the cooperative characteristic requires an Internetware system to support co-management as flexible as possible since the cooperation may be neither predictable nor deterministic. Furthermore, the tangled or complementary relationship of self-management and co-management makes management capabilities much more complex and difficult to implement. Since Internetware implies one or more systematic approaches for developing, deploying, operating and evolving software systems running on the Internet, these approaches should support the construction and evolution of management capabilities with some basic requirements, including reduce the complexity and difficulty of management, separate management from functionality but synthesize them when necessary, etc.

There are several well-recognized approaches for implementing management capabilities, e.g. autonomic management loop and system control hierarchy. Recently, architecture-based runtime management (ARM) gained more and more attention as a promising approach for management of complex software systems [11,12,19,21]. We argue that ARM is the best-of-the-breed for Internetware because: 1) It reduces the complexity of management by representing the target system as high-level, understandable and operational architectural models; 2) It reduces the difficulty of management by utilizing model analysis and learning methods for automated management decision making; 3) It separates management from functionality via reflecting the target system as runtime architectural models; 4) It is compatible with the development of the target system because architectural models are the key artifacts in software development.

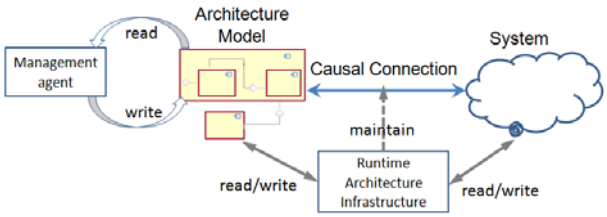


**Figure 1. Architecture-based runtime management**

As shown in Figure 1, ARM has three key elements: a system that is readable and/or writable supported by management mechanism of the runtime system, an architecture model that reflects the system from a certain perspective, and a runtime architecture infrastructure (RAI) that maintains the *causal connection* between the system states and architecture model. Here "causal connection" means the system changes cause corresponding architectural changes just-in-time, and vice versa.

In the past decade, there are many research approaches towards architecture-based runtime management, such as [20, 5, 12, 6, 14]. These approaches demonstrate the effectiveness and feasibility of the architecture models for runtime management. However, there are some open issues, especially for popularizing the ARM. Firstly, almost all RAIs are specific to their own architecture models. If management agents prefer other architecture models, they have to use the other RAIs. Secondly, very few RAIs support more than one target systems, mainly because different systems require different implementations of the management capability, which are usually time-consuming and error-prone. Last but not least, all RAIs are intrusive for the target systems. The system states are assumed to conform to the architecture model and the system has to implement corresponding management mechanisms which usually modify the original implementation of the target system. These issues bring some obstacles for ARM's popularization, such as the target system has to be modified (it is highly risky for large-scale systems), the system can be managed by one or very few architecture models (we cannot use the preferred models in many cases), etc.

In this paper, we report our SM@RT tool that allows developers to construct non-intrusive runtime architecture infrastructures in an automated, model-driven manner. Developers only need to define four declarative, model-level specifications, including an MOF (Meta Object Facility) [9] meta-model specifying the architecture style, another MOF meta-model specifying the structure of running system, an access model specifying how to invoke the management API, and a QVT (Query/Views/Transformations) [9] model transformation specifying the relation between the architecture model and running system. From these four specifications, the SM@RT tool automatically generates the corresponding RAI, without modifying the target system.

Our main contributions can be summarized as follows.

- We identify the main elements for architecture-based runtime management, i.e. architecture style, system structure, and causal connection between them, and provide a model-based way for developers to specify these elements.
- We provide a novel and generic approach to maintain causal connections between heterogeneous architecture models and running systems. Our approach is based on QVT bidirectional model transformation.
- We implement a generation tool to automatically generate the runtime architecture infrastructure from the model-based specifications, supporting the specified architecture-based runtime management.

The generated RAI has a good flexibility for deployment, from powerful servers to resource-limited devices. To balance the requirements for management activities, like short management time, low interference to target systems, we discussed several different deployment styles for the generated RAI, along with the kinds of systems they are proper to.

We successfully applied the SM@RT tool to support the typical architecture-based runtime management (the C2-based runtime evolution proposed by Oreizy et al. [20]) on a practical system (a JOnAS based JEE system [18]). We also undertook several other case studies to demonstrate the capability and usability of the SM@RT tool.

The rest of this paper is organized as follows. Section 2 introduces previous approaches. Section 3 overviews our approach with a running example. Section 4 and Section 5 presents how to specify an architecture-based management scenario, and how we generate the infrastructure to support the scenario. Section 6 discusses how to deploy the generated RAIs into target systems. Section 7 presents the case studies for applying our approach on typical and practical scenarios. Finally, Section 8 gives a brief discussion about the limitation of our work, and Section 9 concludes the whole paper.

## 2. Related Work

There are many approaches toward runtime architectures, focusing on different aspects. Some of them focus on the low-level mechanisms for retrieving and updating runtime states [3,4,5,7], and for ensuring the consistency of the running system after reconfiguration [17, 24]. Some other researchers focus on the high level representation and specification of the running system for intelligibility and usability [19,22], for automatically executing or evaluating the reconfiguration [20,26], or even for self adaptation [12]. There are also approaches toward constraining the runtime change from architecture level, and relevant approaches can be found in the surveys [6].

Distinguished from these typical approaches, we focus on the issue of constructing infrastructures to automatically maintain the causal connections, and allow developers to choose or construct their preferred architecture style and runtime system. This issue is not emphasized in most of the existing approaches, but it is important for the practical use of runtime architectures. By assisting developers to easily combine runtime systems with architecture models, we actually provide an attempt toward leveraging the sorts of research results mentioned above. We wish our common solution to this necessary issue can also help researchers on runtime architectures to continue concentrating on the proper forms or usages of architecture models, or the low-level mechanisms for manipulating runtime systems, without worrying about how to connect them together.

Among the existing approaches on runtime architectures, Rainbow [12] shares the most commonality with ours. The difference is that Rainbow provides a common structure and guidance for constructing different runtime architecture infrastructures, and help reusing knowledge between the constructions, but we provide a generating approach for constructing such infrastructures.

Our solution for maintaining the causal connection deeply roots in the research on bidirectional transformation and model synchronization. Typical approaches include J. N. Foster et al.'s bidirectional tree transformation [10], M. Antkiewicz and K. Czarnecki's model-code synchronization [2], and H. Giese and R. Wagnerand's incremental model synchronization [13]. Our approach attempts to apply bidirectional transformation technologies to a new area.

## 3. Approach Overview

In this section, we give an overview about our approach. We first present a small example, which will be used throughout this paper. Then we discuss how to construct a runtime architecture infrastructure in a traditional way, according to the running example. Finally, we briefly introduce our novel model-driven approach for constructing the infrastructure.

### *3.1 A running example*

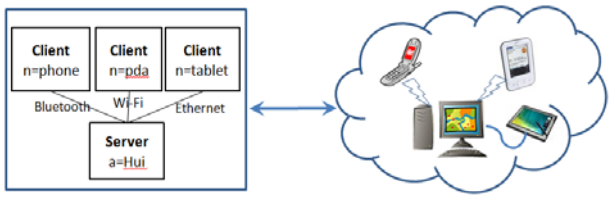


**Figure 2. A running example**

The example is shown in Figure 2. The right part shows a simple mobile computing system, where three mobile devices communicate with a desktop computer via PLASTIC Multi-radio Networking Platform [27]. The left part shows a Client/Server architecture model representing the current state of this system. The infrastructure between them enables the administrator to use this architecture model to intuitively *monitor* and *control* the system. Specifically, *monitor* means that if the system state changes, the infrastructure refreshes the architecture model to let the administrator know this change, and *control* means that if the administrator modifies the architecture model, the infrastructure reconfigures the system. For example, if the third device is powered off, the "tablet" component will disappear from the architecture model, and if an administrator changes the first link from "Bluetooth" to "Wi-Fi", the network between the server and the first device will be re-connected.

### *3.2 Constructing the runtime architecture infrastructure*

Considering the three key elements of the ARM, the following questions should be answered when constructing the RAI:

- What kind of system states can be managed and how to obtain and reconfigure them?
- What kind of information should be presented on the architecture model, and in what form?
- How to synchronize the architecture model and system states according to what rules?

#### *3.2.1 System states and management APIs*

The states of a software system are nearly infinite in general, while the ARM only cares about the states which can be accessed through the management APIs. Such managed states are constituted of management elements (e.g. devices, connections), local states or properties of these elements (e.g. the connection type), and the relation between these elements, as summarized by Sicard et al [23].

Today's software systems usually run on middleware, typically JEE and .NET, which provides plentiful and powerful APIs for managing the whole middleware-based system. Such middleware management APIs are derived from the application domains, and are different from system to system.

For our running example, the system is constituted of a local desktop computer, some remote devices and devices' active networks. These elements form a tree structure. The synchronizer manipulates them by invoking PLASTIC management API. For example, it can inspect the active devices by broadcasting a request and collecting the devices' responses.

#### *3.2.2 Architecture styles and architecture models*

An architecture style constrains a kind of architecture models. In some sense, it defines what kinds of elements may appear in the architecture models, the properties of these elements, and the configuration between these elements [12]. Though their original role is not for management, architecture styles can tell management agents how to understand and manipulate the architecture model. An architecture model can be manipulated according to the architecture style by APIs (for software agents) or GUIs (for human administrators) [20].

A system can be managed by different styles in terms of the preference of management agents or characteristics of the system. In our previous RSA work [15], a system can be managed by the Attribute-Based Architectural Style, Recovery Oriented Computing Style and Component Array Style for different management purposes.

The architecture model in our running example is in Client/Server style. It contains a `Server` and several `Clients` and `Links`. The `Server` records the name of current administrator. The `Clients` have unique names, and the `Links` have specific types. The architecture model is stored as XMI [9] files.

### *3.2.3 Causal connection and runtime architecture infrastructure*

When maintaining the causal connection, the RAI must follow a particular relation between the architecture model and the system states. This relation has different names (like consistency rules [20] or translation rules [12]) in different approaches, but its content is the same. For our running example, the relation can be briefly described as "the Server represents the desktop computer, the Clients represent the active devices, and the Link between a client and a server represents an available network provided by the corresponding device."

The major challenge for maintaining causal connection is the fact that the architecture model and the structure of running system are usually heterogeneous. For the running example, the architecture model is a flatten structure: the clients, servers, and links are all the children of the structure element, but the system is in a hierarchical structure: the desktop computer is the parent of all devices, because people can only get the devices through the desktop computer. Due to the heterogeneity, the relation between architecture and system are not simply one-to-one mapping, and maintaining the causal connections according to these complex relations is also hard.

Let us use a simple scenario to illustrate how a runtime architecture infrastructure supports this runtime management case. Suppose an administrator named "Yingfei" changed all the link types to "Wi-Fi", and changed the server's admin from "Hui" to "Yingfei". In the meantime, the running system itself also changed: the tablet computer stopped.

For this situation, the infrastructure has to do the following tasks. 1) It identifies the above architecture modifications and system changes. 2) It figures out that modifying the link type from "Bluetooth" to "Wi-Fi" means reconfiguring the network of the "phone" device. We assume this request succeeds and the network becomes "Wi-Fi". 3) It ignores the other modification of link type because of conflict. Specifically, the modification means a system change for reconfiguring the network of the "tablet" device. This is an illegal system change, because the device is stopped, and thus if the synchronizer sends a request to the tablet computer, it has to wait for the response until timeout. 4) It deletes the "tablet" client and the link in the architecture model, according to the system change. In the meantime, it preserves the modification of the server's admin, so that the subsequent administrator knows her predecessor.

In this way, the infrastructure reconfigures the system for the administrator, and returns a new architecture model to inform him the system changes and the reconfiguration result. But the infrastructure's work is still not over, because the invocation to management API may fail. For example, if the phone currently does not support "Wi-Fi", then the request in task 2 will not have any effect. This time, the infrastructure returns an architecture model where the first link is still "Bluetooth", and it also returns a warning about this failure.

Implementing an RAI carrying the above tasks is a complex task. Take some existing approach as an example, the rainbow prototype requires 102 kilolines of code [12]. The RAI generated by our approach is a reinvention of C2 based on JOnAS management APIs and has 28 kilolines of code (even without the reused QVT engine). Writing such amount of code manually is tedious and error prone.

### *3.3 Automating the infrastructure construction*

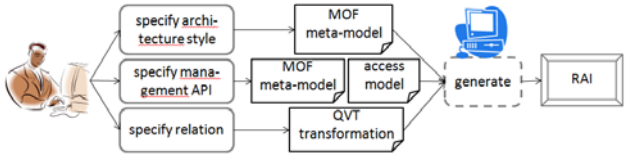


**Figure 3. Approach overview**

The core idea of our approach is that the above questions in RAI construction can be answered by user-defined model-level specifications, and the RAIs can be automatically

generated from these specifications. Figure 3 gives an overview of our approach from a developer's perspective. We require three manual tasks from developers, i.e. specifying the architecture style, the system structure and the relation between them. The specifications are in the form of standard MOF meta-models, QVT transformation, and an access model. We discuss about how to do this specification work in Section 4. From these specifications, the SM@RT tool automatically generates the infrastructure to support architecture-based runtime management according to developer's specification. We present how we achieve the automatic generation in Section 5.

## 4. SPECIFICATIONS

### *4.1 Specifying architecture styles and system structures*

At first, developers should specify the architecture style and system state types as two MOF meta-models. We employ MOF because it is a well-accepted standard and also powerful enough both for specification and code generation. MOF meta-models are constituted of `Classes`, and `Classes` contain `Properties` and `Operations`. `Properties` of basic data types (like integer and string) are usually called as `Attributes`, while `Properties` of `Class` types are called as `Associations` or `Aggregations` (if one element contains the others) [9]. We give the following guidance for defining these two meta-models.

For architecture styles, the `Component Type` and `Connector Type` can be specified by MOF `Classes`; the `Properties` like link types are specified by MOF `Attributes`; the possible `Configurations` like "which component joints with which link" are specified by MOF `Associations`. Here `Component/Connector Type`, `Property` and `Configuration` are main concepts in architecture styles, as summarized in [21].

We specify the architecture style for our running example as Figure 4. A `Structure` could contain several `Clients` and `Servers`, and they are connected by `Links`. The `Server` provides a number of resources, and each `Client` consumes part of these resources.

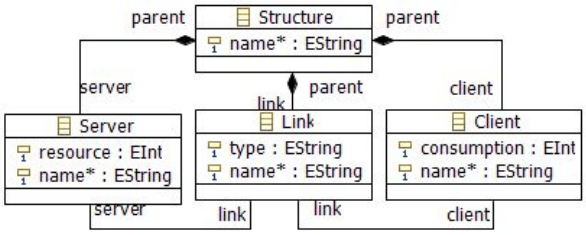


**Figure 4. Client/Server architecture style**

For system states, the types of management elements can be defined as Classes, their local states can be defines as Attributes, the connections between them can be defined as Associations, and the composition relation between them can be defined as Aggregations. These concepts of system states are summarized in [23].

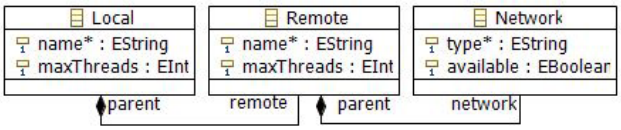


**Figure 5. System meta-model for PLASITC system**

Figure 5 shows the sample MOF meta-model we defined for PLASTIC. As a simple example, this specification does not involve functional links and connections.

These meta-models are not hard to define, for people who are familiar with MOF standards. And moreover they are reusable. For example, the Client/Server meta-model in Figure 4 can be reused directly for generating the RAI for managing .NET systems by the same style.

### 4.2 *Specifying management APIs*

System states are read and written by invoking management APIs. Different management APIs allow external programs (like RAIs) to invoke them through quite different ways. For example, to retrieve an attribute value from JEE systems, we have to obtain a remote management entry and call its `getAttribute` method, while we have to cast the element into an `AttributeController` interface and call one of its `get` methods from Fractal systems. In our approach, developers can define how to invoke the desired management APIs in an access model, which conform an MOF meta-model as shown in Figure 5. This meta-model defines an organization of how to associate a system state with management APIs and helps to reuse the repetitive lines of code.

Figure 6 shows part of the access model for our running example. We defined a `ClassMap` (Element 1) and a `PropertyMap` (4) to decorate the Local class and its remote aggregation (see Figure 5). Under the `PropertyMap`, we add a `ListSub` (5) to specify how to list all the active remote devices. The corresponding Java code is directly specified by the `Code:Fragment` (6). The code is shown in List 1, where we launch a listening thread (Lines 2-3), and then broadcast a request (Lines 4-7). During the sleep time (Line 8), the listening thread collects the active devices' responses and stores the information of each device under a hash map. The logic for broadcasting and listening under PLASTIC API is specified inside the two Java classes `Broadcast` and `Listen`, which are defined under the `UtilField` (3). The two variables `mncListner` and `mncBroadcast` are also declared in UtilField (3) and the logic for instantiating and configuring them is defined in `Lookup` (2). In the same way, we also specified how to get and set the local computer's maximal thread number (8,9), and how to active a new network for a device (12).

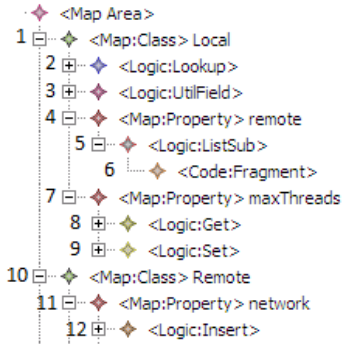


**Figure 6. Access model for PLASTIC systems**

```
1  HashMap res = new HashMap();
2  Listen lt = new Listen(mncListner, res);
3  new Thread(lt).start();
4  Broadcast bt = new Broadcast(mncBroadcast,
5        name+"Group",
6        mncListner.getMyPlasticAddress());
7  new Thread(bt).start();
8  Thread.sleep(1000);
9  return new ArrayList(res.values());
```

**List 1. Sample code inside access model**

### 4.3 *Specifying the relation between architecture model and system state*

QVT is the most widely accepted language for specifying the relation between two heterogeneous models, and thus we adopt it for developers to specify the relation between architecture models and system states. Such relation can be specified as a set of QVT relations, in the following format:

```
relation RelationName{
    domain arc a1:AC1{attribute1=variable1,…, }…
    domain arc an:ACn{…}
    domain sys s1:SC1{attribute2=variable1}…
    domain sys sm:SCm{…}
    when{OCL condition}
}
```

Here, $AC_1$... $AC_n$ are n classes defined in the architecture meta-model, and $SC_1$...$SC_m$ are m classes defined in the system state meta-model. The meaning of such a QVT relation is as follows. For any combination of `n` elements in the architecture model {$ae_1$...$ae_n$}, if they conform to the type $AC_1$...$AC_n$, respectively, and they satisfy the OCL condition in the `when` clause, then there must be a group of `m` elements in the system states {$se_1$...$se_m$}, and the attribute values of all these elements (both the architecture and system) must satisfy all the "`attribute=variable`" equations (that means, for example, $ae_1$.`attribute`$_1$ and $se_1$.`attribute`$_2$ must have the same value, because they both equal to the same `variable`$_1$). This relation is bidirectional. That means it also has the meaning of "if there are some elements in the system states…, there must be some elements in the architecture model…" To simplify the presentation in this paper, we skim some complex capabilities of QVT, like constant match and relation recursive. But actually, we do not limit the use of QVT. That means every legal QVT transformation can be accepted as a specification of relation. Therefore, developers can specify the relation by learning the QVT standard, or any QVT tutorials.

```
1  transformation CS2PLA(arc:CS,sys:Plastic){
2    key Structure{name};...key Network{type};
3    top relation StrServer2Local{ ... }
4    top relation Client2Remote{ ...  }
5    top relation Link2Network{
6     ltype:String;
7     enforce domain arc link:Link{
8      parent=strctr:Structure{},
9      client=clnt:Client{},
10     server=srvr:Server{},
11     type=ltype };
12    enforce domain sys network:Network{
13     parent=rmt:Remote{parent=lcl:Local{}},
14     available=true,
15     type=ltype};
16    when{StrServer2Local(strctr,srvr,local) and
17         Client2Remote(clnt,rmt); }  }  }
```

**List 2. Excerpt of QVT relation for CS-PLASTIC sample**

List 2 is an excerpt of the QVT transformation for the running example. This QVT transformation contains three top relations, specifying the relation as described in Section 3. For example, the third relation, `Link2Network` (Line 5), specifies that "the link between a client and a server represents an available network provided by the corresponding device". We can read this rule as follows. For each link in the architecture model (Line 7), if its parent is `strctr` (Line 8), its type is `ltype`, and it connects `clnt` and `srvr`, then there must be a `network` in the system model (Line 12). This `network` is available (Line 14) and its type equals `ltype` (Line 15). Moreover, the network's grandparent `lcl` (Line 13) satisfies the `StrServer2Local` relation with `strctr` and `srvr` (Line16), and its parent `rmt` (Line 13) satisfies the `Client2Remote` relation with `clnt` (Line 17). These `when` clauses locate the network under the correct client.

## 5. Generating Infrastructures

From the above specifications, the SM@RT tool automatically generates the RAI to support the architecture based runtime management.

Our main idea is as follows. We first implement a generic synchronization engine. This engine maintains the causal connection between architecture models and system states according to the relation specified by QVT. To make this generic engine work for various architecture styles and systems, we develop two code-generation tools to automatically generate architecture adapters and system adapters. These adapters wrap the XMI files storing architecture models and the various system management APIs by a standard MOF-reflection interface, so that the generic engine can manipulate all the architecture models and system states in a unified way as manipulating standard MOF models. In summary, SM@RT tool contains three parts: two generation tools for architecture and system adapters and a generic synchronization engine.

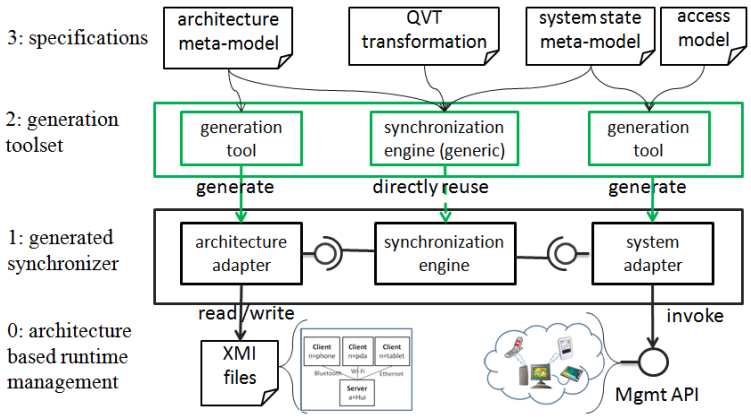


**Figure 7. Generation overview**

### *5.1 Generating adapters*

Architecture adapters are generated from architecture meta-models. For each MOF Class, EMF generates a Java class that implements the MOF reflection interface. All generated Java classes expose their MOF reflection interfaces as the API of the architecture model. Their instances form the in-memory image of the model. A prefabricated Java library synchronizes the memory image and XMI file by invoking MOF reflection interface and Eclipse XMI parser.

System adapters are generated from system state meta-models and access models. Firstly, we reuse EMF generator to generate a Java class per MOF Class in the system state meta-model. These Java classes also implement the MOF reflection interface. Secondly, we generate another set of Java classes that encapsulate the code defined in the access model and these classes know how to access the management APIs. Finally, some prefabricated Java classes will connect the two sets of Java classes so that when the synchronization engine invokes the MOF reflection interface, its invocation will be delegated to the corresponding management API that manipulates the system state actually.

### *5.2 Maintaining causal connections*

We implement a generic synchronization engine that maintains the causal connection according to the meta-models and QVT transformation.

As we discussed before, there are two major challenges to maintain causal connections. First, architecture model and the running system are heterogeneous. That means we cannot directly map the architectural modifications into the system changes, but have to deduce what an architecture modification means on the running system from the QVT relations, and vice versa. Second, the system is dynamically changing, and thus there might been conflicts between architecture modifications and system changes. We address these challenge through a synchronization algorithm based on QVT bidirectional transformation and model comparison.

The basic idea of our algorithm is: get the system-side meaning of architectural modifications by transforming the two architecture models into two system models and calculate the difference between them, then try to manipulate the system according to the difference and retrieve the manipulation result, and finally transform the resulted system state back and reflect it in architectural level.

The algorithm is shown in Figure 8. This algorithm processes two XMI files and a running system. Before the execution of this algorithm, the two XMI files stores the architectural models before and after the management agent's modifying, and after the execution, the system state is changed according to the management agent's modification, and the two XMI files reflect the current system state.

#### *5.2.1 Techniques overview*

Architecture adapter and system adapter are generated from the meta-models, as discussed in the previous section. The arrow from an adapter to a model means loading a model through the adapter. We achieve this by invoking the standard "get" method recur-

sively to retrieve the complete information from XMI files and running systems, and then copy the constructed model.

Forward and backward transformations are bidirectional transformations on the basis of the relation defined by QVT. Forward transformation looks at a pair of models (`m,n`) and works out how to modify `n` so as to enforce the relation: it returns the modified version. Similarly, backward transformation propagates changes in the opposite direction.

Model difference [1] calculates the difference between two models under the same meta-model. The difference result is a set of model modifications like creation of a new element or update to a feature. Model merge is the operation for modifying a model according to a difference, and return the modified model. The arrow from a differ activity to the system adapter means merging the modifications in the difference result on the system state.

*5.2.2 The algorithm*

Based on the above techniques, the algorithm is constituted of the following steps.

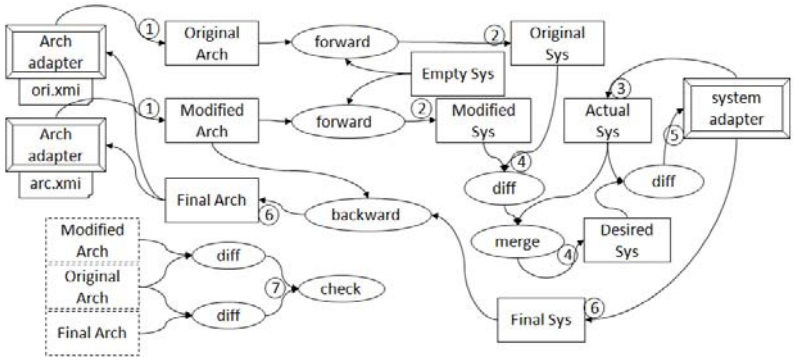


**Figure 8. Synchronization algorithm**

**Step 1. Loading architectural models.** Read the XMI files through the architecture adapter, and construct the in-memory models for further processing.

**Step 2. Forward transformation.** Transform these `Original Arch` and `Modified Arch` into `Original Sys` and `Moidfied Sys`, respectively. We use an empty system model as the second input of forward transformation, because at this time, we do not require extra system information.

**Step 3. Retrieve System State.** Invoke a model copy method from system adapter to an intermediate model named `Current Sys`. During the copying process, this method will recursively invoke the standard get methods, so that the entire system state will be retrieved and stored in the `Current Sys` model.

**Step 4. Merge the management agent's modification.** We first differ `Modified Sys` and `Original Sys`. The difference we obtained is actually the management agent's intention on changing system state. We merge this difference into `Current Sys` to get the `Desired Sys`, which is the system state expected by the management agent.

**Step 5. Try to reconfigure the system**. Compare `Desired Sys` and `Current Sys`, and the resulted difference contains the modifications we need to change the system according into `Desired Sys`. Then we merge this difference into the actual system through the system adapter. This merge process is the same as we used in the last step, i.e. translate the modifications into invocations to the standard MOF reflection interface.

**Step 6. Get final architectural model.** We execute backward transformation between Modified Arch and Final Sys, to get the architectural model reflecting the system state after synchronization.

**Step 7. Check the synchronization result.** The synchronization is over at Step 6, but we have to inform management agents that some of their modifications do not have expected result. We do this by get an intended difference from `Original Arch` to

`Modified Arch`, and an actual difference from `Original Arch` to `Final Arch`, and then check if the intended modifications all remain in the actual modification.

*5.2.3 The properties of this algorithm*

In this section, we discuss some properties of this algorithm, explaining how it satisfies the requirements of maintaining causal connection.

**Property 1. Synchronization leading consistency.** According to the "Correctness" property discussed in literature [24], after the backward transformation, `Final Arch` and `Final Sys` satisfy the relation specified in QVT. Therefore, synchronization always result consistent architectural model and system state. This property ensures that management agents get the correct system state through the architecture model.

**Property 2. Non-interference reading.** That means if management agents do not modify the architecture model, the RAI will not affect the running system. Go back to the algorithm, if the `Original Arch` and the `Modified Arch` equal to each other, we will get equal `Original Sys` and `Modified Sys`, so the difference between them is empty, and thus `Desired Sys` equals to `Current Sys`. Finally, the empty difference between `Desired Sys` and `Current Sys` will not cause any modification to the system adapter.

**Property 3. Effective writing.** That means if the management agent modifies the architecture model, the RAI will change the system accordingly. For normal conditions, this algorithm satisfies this property. The modifications on architecture side will be translated to a proper modification stored in the difference between `Desired Sys` and `Current Sys`, and this modification will cause reconfiguration in the system through the system adapter, and the correct reconfiguration result will be reflected. For exception conditions, e.g. the modification on systems does not lead expected result, or the modified part in architectural model cannot be transformed into the system model. For such situations, we will inform the management agents.

**Property 4. No insignificant change.** If the architectural modifications and changes do not break the relation, the RAI keeps the architecture model and running system. For such situation, `Desired Sys` will equal to `Current Sys`, and there will be no reconfiguration operations on the system adapter. So the system will not be changed. Furthermore, according to the "Hippocraticness" property [24], when executing the backward transformation, the Modified Arch will not be changed.

*5.2.4 Implementation*

We implemented this algorithm using a QVT transformation engine named "medini QVT" [28], and developed a model compare and merge engine by ourselves [25]. The generation tool is in the form of a set of Eclipse plug-ins, which can be executed in Eclipse platform 3.4 or later, with EMF support.

## 6. Deploying the Generated Infrastructures

In this section, we discuss how to deploy the generated runtime architecture infrastructures into the Internetware systems, and how they can be used by management agents. As discussed before, Internetware systems are constituted of different kinds of devices, which communicate with each other through complex networks. In order to make SM@RT applicable for a wide scope of Internetware systems, we provide a flexible way for deploying the generated infrastructures, considering both device resources and network conditions. In the rest of this section, we first introduce the architecture of the generated infrastructures (not the "architecture models" maintained by the infrastructures, but the architecture of the infrastructures themselves), and then we discuss about some typical deployment solutions (i.e. the potential architecture configurations), suggesting for what kinds of systems they fit. These suggestions help developers ultimately utilize the SM@RT approach on their systems.

*6.1 The architecture of infrastructures*

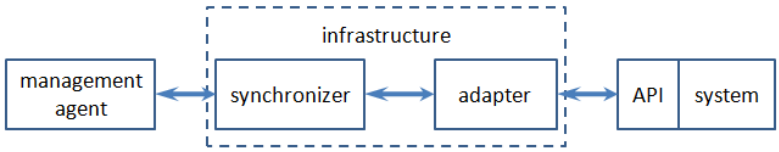


**Figure 9. The architecture of infrastructures**

Figure 9 shows the common architecture of the generated infrastructures. Generally, and architecture is constituted of two components: an adapter (here it refers in particular to the system adapter) that reflects the system state into a homogeneous system model, and a synchronizer that synchronize this system model with the abstract architecture model. During runtime, the management agent communicates with the synchronizer to get and write back architecture models. The synchronizer then communicates with the adapter to synchronize the architecture model with the system model. Finally, the adapter invokes the API to get the real system state and execute the system changes. Next, we first discuss the implementation and runtime features of these two kinds of components, and then investigate the different types of connections between them.

*6.1.1 The components*

The system **adapters** are generated as pure Java code, conforming to the syntax of Java 1.5 and later. Besides some basic J2SE APIs for I/O, data structures, etc, this generated code only depends on some essential EMF library. To achieve better flexibility, we cropped a bit on the EMF library to make it only depends on some basic J2SE APIs, and deliver its source code together with the generated adapter source code. Therefore, the whole source can be compiled and wrapped as a single and self-contained package, which could be deployed on a device with basic J2SE support. An important benefit of this solution is that the adapter could not only be compiled and executed by the standard Sun J2SE SDK, but also could be used under some J2SE-compliant platforms. One typical platform is the "Dalvik" platform employed by the Google Android OS for mobile phones. Dalvik adopts the syntax of Java language, and supports a set of basic J2SE APIs, but its compiler generates different byte-code from Java standard, and such byte-code can be only executed on the Dalvik virtual machine. We have carried an experiment, and it demonstrate that the generated adapter source code can be compiled by Dalvik compiler, and compiled adapters works well on the real Android mobile phone, as will be shown in the next section. Either on the standard J2SE platform or on the Dalvik platform, the compiled adapters are light-weight. That means their package size is small, they occupy relatively little system resources during runtime

The generic **synchronizer** is delivered directly as compiled Java byte-code. It contains the classes implementing the synchronization algorithm, as well as some reused tools like the medini QVT engine, and the model comparison library. The specific synchronizers are just the generic one carrying some customizing files, like the QVT transformation and the MOF meta-models. Such synchronizers are quite heavy-weighted, and due to some implementation reasons by the medini QVT engine, these synchronizers can be only executed on J2SE 6.0 or later.

**Table 1**. Comparison of sample synchronizers and adapters

| | host device | form | Package size | Memory occupation | CPU time |
|---|---|---|---|---|---|
| Synchronizer | desktop | jars | 6.9MB | ~60MB | 2.3s |
| Adapter(J2SE) | desktop | jar | 1.2MB | ~4MB | 0.4s |
| Adapter(Dalvik) | phone | pkg | 0.7MB | 1.1MB | 6.5s |

Table 1 summarizes and compares some features of the synchronizer and the adapters implemented in the two platforms, using some samples from our case study. The data taken from these samples may not reflect the exact features of the two kinds of components, but they do give a intuitive comparison between them.

The samples of the synchronizer and the J2SE adapter are taken from the C2/JOnAS case, for which we will present in the next section. For this case, the synchro-

nizer and the adapter are running on the same desktop computer with J2SE 6.0. The synchronizer is constituted of several jar files, 6.9MB in total, while the adapter is formed of only one jar file of 1.2MB. The memory occupation of them is approximately estimated by observing the change of the memory occupied by JVM process before and after executing the two components. The CPU time are calculated by the start and stop timestamp of different phases during the execution of the algorithm. Comparing the data, we can see that synchronizer is much more heavy-weighted than the synchronizer. It is because the bi-transformation process is quite resource consuming. The sample of the Dalvik adapter is taken from the Android case, which runs upon a real mobile phone, and the data are calculated using some existing Android facilities. The package size and the memory occupation is quite the same as the J2SE version, but it takes much longer time for execution, because of the poor calculation capacity of the mobile phone. We can image that it is not practical for the synchronizer to run on such a device.

### *6.1.2 The Connections*

The **connections between adapters and systems** are determined by the access model. If developers define the access model using local invocation to the management API, then the adapter must be deployed on the same device with the target system. Alternatively, if developers define the access model using Remote Procedure Calls (RPC), the adapter can be deployed on a remote device. Once the access model is defined, the connection type is fixed, and cannot be changed during runtime.

There are two types of **connections between synchronizers and adapters:** The synchronizer could directly invoke the adapter and copy the state as a static system model. It could also notify the adapter from a remote device, and the adapter will construct the system model and transmit it into the synchronizer using serialization. We implement the mechanisms for remote notification and model serialization, and developers could switch between the two types before deploying the RAI.

Similarly, there are also two types of **connections between the management agents** and synchronizers. They could directly edit the architecture model, or ask the synchronizer to send the architecture models through networks.

### *6.1.3 The potential deployment solutions*

Combining the management agents, synchronizers and adapters using different kinds of connections, we can derive several different configurations of RAIs. In the remainder of this section, we will show some typical configurations, as shown in Figure 10, discussing the advantages and disadvantages of these deployment solutions, as well as the proper conditions for them. This helps developers to ultimately utilize our SM@RT approach.

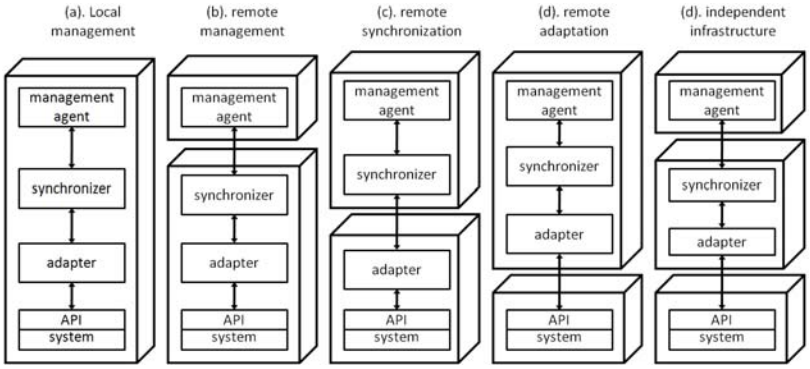


**Figure 10**. Typical deployment solutions for RAI

**Local management**. The most common and basic solution is to deploy the synchronizer and the adapter in the same node where the target systems is running, and employ a management agent which also runs on the very same node (For human administrators, this "management agent" could be the auxiliary or visualization tools, like the architecture edior). Communicating only through local connections, this configuration has the theoretically maximal performance, providing that the only node is powerful enough. This configuration is proper for the cases that people want to manage a software system run-

ning on a normal machine, and the management activities are not too complex or too frequent. It is also useful for equipping a node inside a big Internetware system with some local and independent management capability.

**Remote management**. Usually, the management agent is much more resource consuming than the synchronizer and the adapter. For example, the graphical architecture editor usually require a lot of memory to catch the graph of different architecture elements and configurations, and the AI-based automated management agents usually spend a long time to execute the AI algorithms. Considering this, it is an intuitive idea to isolate the management agent from the target system. This configuration cause lower burden to the target system, considering with "local management", and since the management agent only communicate with the synchronizer twice during one management loop (see the algorithm in Section 5.2, the communication cost is also low. This configuration is proper to the Internetware systems with central management. In Internetware systems, when there are a lot of nodes, people may need some nodes to manage a set of other nodes, e.g. setting their parameters to achieve higher performance, replacing some dead nodes to make the whole system stable, etc. If people want such management nodes to treat all the target nodes in a unified and abstract way, they could deploy an RAI into each of the target nodes, so that the management node could manipulate them through their runtime software architectures.

**Remote Synchronization**. The synchronizer is still too heavy-weighted. Developers could choose to extract the synchronizer, and deploy it in the same node with the management agent. Considering the complexity of most management agents, this synchronizer will not cause big burdens to the management node. The problem of this configuration is that the remote communication between the synchronizer and the adapter will slow down the overall performance. Because the adapter has to serialize the model, and the synchronizer has to deserialize it, and vice versa. And this system model is more complex than the architecture model, comparing with the "Remote management". This configuration is proper to the systems constituted by some devices that are resource-limited but still support J2SE or equivalent platforms like Dalvik. Currently, we only support one such platforms, i.e. Dalvik, and we will try to adjust our generated code to support more platforms like the J2ME. This configuration is also proper to the systems whose nodes are very busy for their functional tasks, because the light-weighted adapters will not cause big burden to the nodes.

**Remote Adaptation.** If developers define the access model using RPC invocations to the nodes, the management logic can be completely separate from the target nodes. The advantage of this configuration is that the management activities do not disturb the functional activities the target nodes. But its disadvantage is that the overall performance of the management activities is even lower, because in each management loop, there are a lot of remote invocations from the adapter to the managed nodes. This configuration is proper to the systems whose computational resources are central on some few nodes, like the mobile computing networks as shown in our running example, and the sensor networks. For such systems, the nodes do not support complex computation tasks, or are not convenient to deploy extra components for management.

**Independent Infrastructure**. Besides the above basic configurations, there are also some hybrid ones. For example, a common configuration is to deploy the whole runtime architecture infrastructure into an independent node, so that it will not disturb either the management or the system logic. This configuration is proper to the systems which reuse some existing management techniques.

## 7. Case Study

We applied SM@RT tool for several case studies. We present one of these case studies in detail to demonstrate the power of SM@RT tool for supporting a typical architecture-based runtime management approach (the C2-based runtime evolution [20]) on a practical system (the JOnAS JEE system). After that, we give brief introductions about other case studies to evaluate the usability and development efficiency of SM@RT tool.

### *7.1 C2-JOnAS*

Let us take a look at how to manage a JEE (Java Platform Enterprise Edition) system by the C2 architecture style.

JOnAS [18] is an open source JEE application server. C2 [20] is an architecture style aiming at the design and runtime evolution of UI-centric systems. Since many JEE applications employ web-based user interfaces as their centers, it is a natural idea to use C2-styled architecture models to manage JOnAS-based systems (JEE applications running on JOnAS servers). We choose this case because it is both typical (C2 is a famous style for runtime management) and practical (JOnAS is widely used JEE server).

#### *7.1.1 Specification*

Figure 11(a) is the architecture meta-model conforming the description of C2 in [20]. Figure 111(b) is the system meta-model of JEE systems. The managed elements used in JPS sample include the `EJBs`, `JDBCDataSources` and `WebModules`. Their types are specified as MOF classes. The local states of these elements are specified as attributes inside the corresponding classes.

Figure 11(c) shows the access model for Jave Management eXtension (JMX) [16] API supported by JEE systems. In the MapArea, we specified the way to manipulate each kind of managed elements through JMX API. For example, for the class `JDBCDataSource`, we specified how to create and destroy a data source. We defined two Code elements to wrap two pieces of Java code for invoking JMX to deploy and undeploy a data source, and put these Codes under a Create element and a Destroy element, respectively. We also use several FeatureMaps to define the way to manipulate some attributes, referring to the reused Codes in CodeArea for retrieving string-valued or integer-valued attributes.

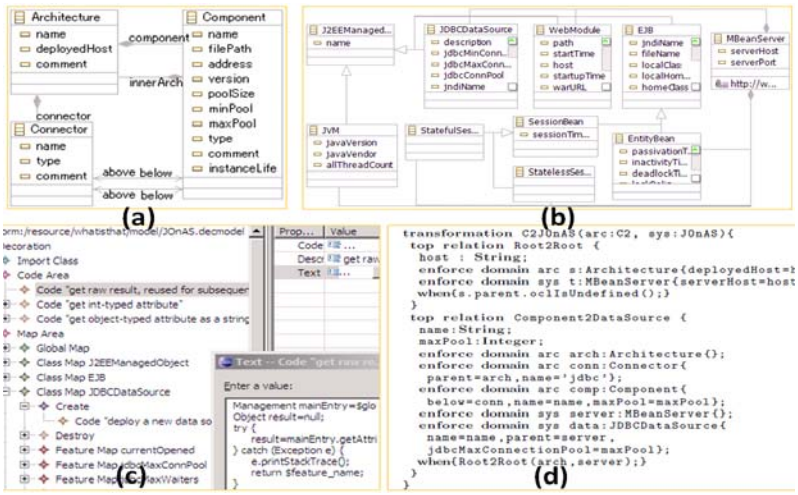


**Figure 11. Specifications of the C2-JOnAS case**

Figure 11(d) shows part of the QVT transformation we wrote for specifying the relation between C2-style architecture models and JEE system states. It means that a Component in the architecture model corresponds to a JDBCDataSource in the runtime system, if and only if they had the same name, and the Component links to a Connector named 'jdbc'.

#### *7.1.2 Using the generated infrastructure*

We use a sample JOnAS system, a Java Pet Store application running on JOnAS server, to illustrate how the infrastructure supports the typical C2-based runtime management on JOnAS system.

Being a JEE blueprint application, JPS (Java Pet Store) sells pets over the Internet. As shown in Figure 12 (b), people can browse and order the pets from a website running on a JEE application server (we use JOnAS here). First of all, the runtime architecture of JPS should be visualized so that we can understand the system to be managed. As shown in Figure 12(a), we employ the layered C2 style which represents the four tiers of a web application clearly. This architecture model illustrates the current structure of JPS, as

well as the contexts of components and connectors, and we can change the structure and contexts by directly editing this architecture model.

Consider a simple management task, through the bottom panel of Figure 12(a), we can find that the current connection pool size of HSQL1 (the component abstracts the backend database) is set between 10 and 100. If the in-use connection number is always more than 50, we can increase the Min Pool Size from 10 to 50 for avoiding the frequent release of database connections. If the number keeps 100 time to time, we can increase the Max Pool Size from 100 to 200 for reducing the waiting time of database access requests. Every time after we change the value on the architecture model, we can invoke a "synchronize" command to ask the RAI to maintain the causal connection. After this synchronization, the configuration of the backend database will be changed into the value we set. If we change the pool size to 10,000, which exceeds the capacity of HSQL, the attribute value will be only 9999, and we will also receive a warning.

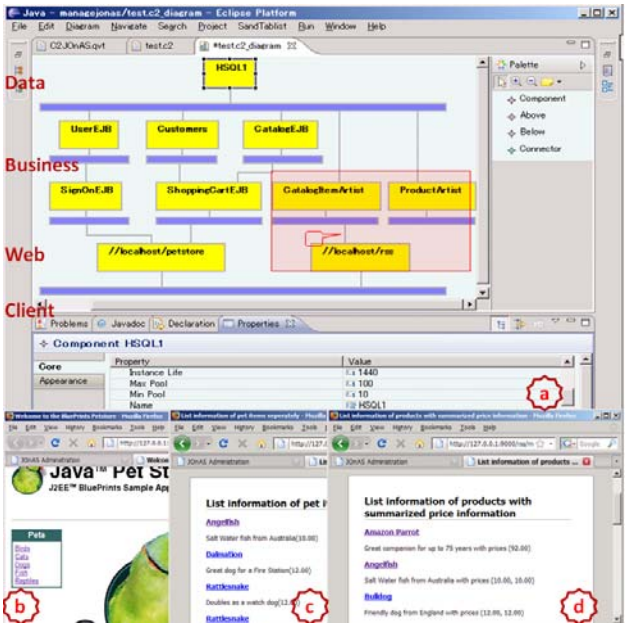


**Figure 12. JPS web pages and its architecture model in C2 style**

For a more complex task, suppose we want to add an RSS (Really Simple Syndication) function to the JPS to support subscription of pet information. Considering the typical evolution scenario of C2 style discussed in [20], we can add `ProductArtist` and `CatalogItemArtist` components for organizing the raw data as products (a product represents a pet type) and items (the same products sold by different sellers are regarded as different items), respectively, and the "//localhost/rss" component for presenting the data as a RSS seed, as shown in the red box of Figure 10(a). After having the new components implemented as EJBs and web modules, we can invoke the synchronize command, and the RAI will deploy the new components into JOnAS. If "//localhost/rss" connects to `CatalogItemArtist` in the architecture model, customers can subscribe a RSS seed with all items via the "http://localhost/rss" URL, as shown in Figure 12(c). If "//localhost/rss" connects to `ProductArtist`, customers can subscribe all products by the same URL, as shown in Figure 12(d). Such switching can be done at runtime by changing the connection between these components in the architecture model.

### *7.1.3 Deployment of C2-JEE case*

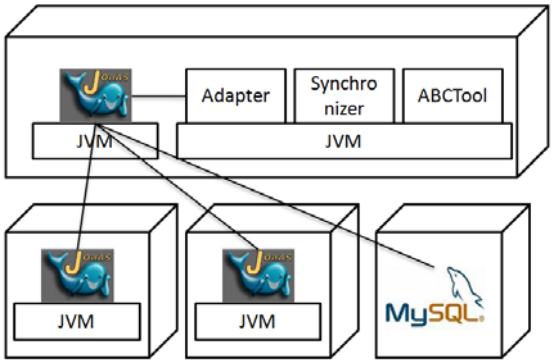


**Figure 13**. The deployment of C2-JEE case

We adopt the "local management" configuration to deploy our RAI for this case: The adapters, synchronizer, and the management agent (which is a graphical architecture editor provided by ABCTool) are all deployed on the central node a this JEE system, but on a different JVM from the JOnAS system, so that the JOnAS could run independently The adapter invoke the JOnAS's JMX API via RPC, but since this RPC only cross two JVMs, not two nodes, the performance is not extremely affected. The central JOnAS also collect and reset manageable data of other servers, but the communication between them are not in charge of our RAI. Therefore, from this perspective, we only provide a management facility for the central node. We choose a local management configuration because the node carrying the JOnAS server must be powerful enough, and also because JOnAS is not a computation-intense system, and thus the node has enough computation resource for doing management.

### *7.1.4 Discussion of C2-JEE case*

The SM@RT tool improves the development efficiency of RAIs significantly. For supporting the JPS management scenarios, we wrote 3 models with 192 model elements in total, 237 lines of code, including the Java code wrapped inside the access model and the QVT transformation text, as shown in Figure 9. Our toolset generates about 27000 lines of Java code, including 8761 and 18263 lines for architecture adaptors and system adapters respectively. We finish this case in two days. As a contrast, an equivalent infrastructure we developed before takes us nearly one month, and consists of 5000 lines of Java code.

The execution performance of the generated RAI is not good, but still acceptable. We perform this sample on a laptop with an Intel Pentium 1.6GHz process and 1.5G memory. All synchronizations could be finished between 2 to 5 seconds. If taking the human manager's operation time into account in the RSS addition task, it takes 37 seconds to draw the three components and connectors, assign attributes, and wait 3 seconds for synchronization. By contrast, we do the same task using the build-in web-based JOnAS management tool in totally 189 seconds, because we have to open the deployment page, locate the EJBs and the web modules, and deploy them one by one.

## *7.2 Other case studies*

We have also performed several other case studies and Table 1 summaries the specification size of each case (including the models and code inside them, that is all the manual work we have to do for 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 2. Summary of case studies**

| # | system | arch style | Model (elements) | code (LOC) | gen (LOC) | con-trast (LOC) | deploy-ment |
|---|---|---|---|---|---|---|---|
| 1 | JOnAS | C2 | 192 | 237 | 27024 | 5294 | local mgmt |
| 2 | Java classes | class diagram | 91 | 124 | 10518 | 3108 | local mgmt |
| 3 | PLASTIC | Client/Server | 67 | 570 | 9126 | - | rmt adapt |
| 4 | Eclipse GUI | ABC/ADL | 86 | 78 | 11290 | - | local mgmt |
| 5 | Android | ABC/ADL | 33 | 47 | 14757 | - | rmt sync |

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, Xingrun Chen. He wrapped the PLASTIC management functions and reflected 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 designed 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 the generated adapter (part of the RAI) on the HTC G1 mobile phone, and with the help of a synchronization engine running on a PC, we visualize the package structure of the smart phone using our previous architecture modeling tool name ABCTool.

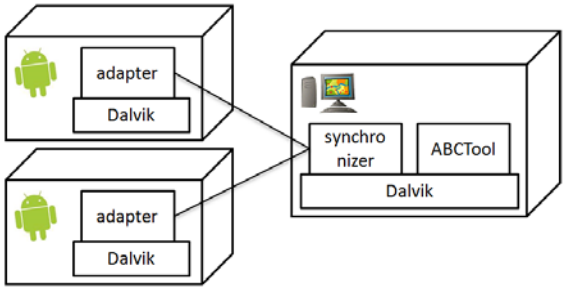


**Figure 14**. Deployment of Android case

Figure 14 shows the deployment of this android case, which is an example of "remote synchronization" configuration. The generated adapter runs on the Dalvik platform on an Android phone, and the synchronizer communicate with this adapter through serialization and de-serialization of runtime models.

## 8. Discussion and Future Work

The main value of our approach is the *explicit, model-level specification* of the requirements for an architecture-based runtime management scenario, and the *automatic generation* of runtime architecture infrastructures. Though the generated code may be not compact and efficient as those written by experts, we believe the experts cannot reduce the code drastically, while the efficiency of generated codes could be improved significantly if we optimize the synchronization engine. Note that the intrusive approach may generate much more compact and efficient codes. But it brings high risks due to the source code modification of architecting tools and managed systems. The manageability has a significant impact on the value of our approach, that is, how many architecture changes can be exactly executed in the runtime system. Since our approach is non-intrusive, its manageability is mainly determined by the management API of the target system. The sample management tool in Figure 2 can reflect 6 kinds of management elements and nearly 50 kinds of attributes. It supports adding and removing EJBs, web modules and data sources through architecture models and changing 12 kinds of their

attributes. If we add JEE Management API and Deployment API into the access model, more architecture-level management can be done.

Currently, the capability of our approach for specifying and maintaining the causal connection is limited by the QVT standard and implementation. Both the QVT standard and the medini QVT engine are powerful enough for our existing case studies. When doing more case studies in the future, we will pay attention on the situations when some causal connection cannot be specified or maintained. We believe such situations are valuable for the research on architecture-based runtime management as well as for the improvement of QVT.

The main cost of our approach is the model-level specifications. We plan to facilitate developers by such as deriving which part of architecture model is writable, introducing semantic-based check for the conflicts and mis-matches, extracting the system state meta-model and access model from the sample usages or testing cases of a management API. We will also try to improve the performance of the generated infrastructure, like reducing the size of state to retrieve by cache, and use incremental transformation to reduce the transformation time.

## 9. Conclusion

In this paper, we propose a model-driven approach for constructing runtime architecture infrastructures in a cost-effective manner. Developers only need to provide 4 model-level specifications, and our toolset can automatically generate the infrastructure for managing a given system via a given architecture style. We have applied this approach on several practical systems.

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, 60933003; the National S&T Major Project under Grant No. 2009ZX01043-002-002; and the EU FP7 under Grant No. 231167.

## 10. References

[1] Marcus Alanen and Ivan Porres. Difference and union of models. 6th Unified Modeling Language (UML) Conference, pages 2-17. 2003.

[2] Michal Antkiewicz and Krzysztof Czarnecki. Framework-specific modeling languages with round-trip engineering. In Model Driven Engineering Languages and Systems, 9th International Conference (MoDELS), pages 692-706, 2006.

[3] N. Bencomo, G. Blair, and R. France. Summary of the workshop Models@run.time at MoDELS 2006. In Lecture Notes in Computer Science, Satellite Events at the MoDELS 2006 Conference, pages 226-230, 2006.

[4] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In International Conference on Software Engineering, pages 811-814, 2008.

[5] G.S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. 1998

[6] Jeremy S. Bradbury, James R. Cordy, J urgen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. 1st ACM SIGSOFT Workshop on Self-Managed Systems, pp. 28-33, 2004.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, J.B. Stefani, I. Rhone-Alpes, An Open Component Model and Its Support in Java, CBSE, pp. 7-22, 2004

[8] F. Budinsky, S.A. Brodsky, and E. Merks. Eclipse Modeling Framework. Pearson Education, project address: http://www.eclipse.org/modeling/emf.

[9] Catalog of OMG Modeling and Metadata Specifications, http://www.omg.org/technology/documents/ modeling_spec_catalog.htm.

[10] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst., 29(3):17, 2007.

[11] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In Future of Software Engineering (FOSE) in ICSE '07, pages 37-54, 2007
[12] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer, 37(10):46-54, 2004.
[13] Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In Model Driven Engineering Languages and Systems, 9th International Conference (MoDELS), pages 543-557, 2006.
[14] Gang Huang, Hong Mei, and Fuqing Yang. Runtime recovery and manipulation of software architecture of component-based systems. Autom. Softw. Eng., 13(2):257-281, 2006.
[15] G. Huang, X. Liu, H. Mei. An Online Approach to Feature Interaction Problems in Middleware based Systems. Science in China, series F Vol 51, No. 3, pp. 225-239, 2008.
[16] Java Management Extensions, http://www.jcp.org/en/jsr/detail?id=77.
[17] Java PetStore, http://java.sun.com/developer/releases/petstore/.
[18] JOnAS Project. Java Open Application Server, http://jonas.objectweb.org.
[19] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng., 16(11):1293-1306, 1990.
[20] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In Proceedings of the 20th international conference on Software engineering (ICSE), pages 177-186, 1998
[21] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In Companion of the 30th international conference on Software engineering, pages 899-910, 2008
[22] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. IEEE Trans. Softw. Eng., 32(7):454-466, 2006.
[23] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. Using components for architecture-based management: the self-repair case. In Proceedings of the 30th international conference on Software engineering, pages 101-110, 2008
[24] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In Model Driven Engineering Languages and Systems, 10th International Conference, pages 1-15, 2007.
[25] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, Hui Song, and Hong Mei. Supporting Automatic Model Inconsistency Fixing, In Foundations of Software Engineering (FSE), 2009, to appear
[26] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In International Conference on Software Engineering (ICSE), pages 371-380, 2006.
[27] The PLASTIC Platform, http://www.ist-plastic.org/
[28] The mediniQVT transformation engine, http://projects.ikv.de/qvt