Supporting Runtime Software Architecture: A Bidirectional-Transformation-Based Approach

Hui Song, Gang Huang, Franck Chauvel, Yingfei Xiong, Zhenjiang Hu, Yanchun Sun, Hong Mei

Supporting Runtime Software Architecture: A Bidirectional-Transformation-Based Approach

Hui Song[a], Gang Huang[a], Franck Chauvel[a], Yingfei Xiong[b], Zhenjiang Hu[c], Yanchun Sun[a], Hong Mei[a]

[a] *Key Laboratory of High Confidence Software Technologies (Ministry of Education), School of Electronic Engineering and Computer Science, Peking Uni., Beijing, China*
[b] *Department of Mathematical Informatics, University of Tokyo, Tokyo, Japan*
[c] *GRACE Center, National Institute of Informatics, Tokyo, Japan*

Abstract

Runtime software architectures (RSA) are architecture-level, dynamic representations of running software systems, which help monitor and adapt the systems at a high abstraction level. The key issue to support RSA is to maintain the causal connection between the architecture and the system, ensuring that the architecture represents the current system, and the modifications on the architecture cause proper system changes. The main challenge here is the abstraction gap between the architecture and the system. In this paper, we investigate the synchronization mechanism between architecture configurations and system states for maintaining the causal connections. We identify four required properties for such synchronization, and provide a generic solution satisfying these properties. Specifically, we utilize bidirectional transformation to bridge the abstraction gap between architecture and system, and design an algorithm based on it, which addresses issues such as conflicts between architecture and system changes, and exceptions of system manipulations. We provide a generative tool-set that helps developers implement this approach on a wide class of systems. We have successfully applied our approach on JOnAS JEE system to support it with C2-styled runtime software architecture, as well as some other cases between practical systems and typical architecture models.

Key words: software architecture, bidirectional transformation, runtime system management

1. Introduction

Nowadays, IT systems are required to be continuously available whereas the systems are running, their environments and user requirements are constantly changing. This calls for the system management at runtime to find and fix defects, adapt to the changed environments,

or meet new requirements (France and Rumpe, 2007; Kramer and Magee, 2007). Currently, many mainstream platforms have provided management APIs for retrieving and updating the system state at runtime (Sicard et al., 2008), but direct management upon these low-level APIs is not an easy task. First, the management API reflects the system in a solution space, requiring the knowledge of platform implementation. Second, the APIs are designed for general-purpose management, and thus are usually too tedious and complicated for a particular management activity.

To control the management complexity, many researchers propose to utilize software architecture for runtime management (Oreizy et al., 1998; Garlan et al., 2004; Sicard et al., 2008). They represent the running system as a dynamic architecture model, which has a *causal connection* with the system state. That means if the system state evolves, the architecture configuration will change accordingly. And similarly, if the architecture configuration is modified, the system will change accordingly, too. Thanks to this causal connection, management agents (human administrators or automated management services) can monitor and control the system by reading and writing this abstract architecture model, utilizing mature architecture-based techniques (such as architecture manipulation languages and architecture analysis (Blair et al., 2009)) to make high-level management decisions at runtime. We name such architecture models as *Runtime Software Architectures* (RSA, Huang et al. (2006)).

There are many approaches to RSA-based runtime management. These approaches reveal the usage and advantage of RSA, but their mechanisms for maintaining causal connection are tightly-coupled with the target system, requiring the systems to be implemented according to specific styles (Oreizy et al., 1998; Blair et al., 1998) or instrumented with specific management capabilities (Huang et al., 2006; Garlan et al., 2004). Due to the tight coupling, these approaches cannot be directly applied on the existing systems that are already implemented without consideration of RSA, because it is tedious and error-prone to instrument them with RSA-enabling code.

In this paper, we focus on providing RSA support to the existing systems. The key issue is to maintain the casual connection based on the general-purpose management API provided by the target system. We sum up this issue as a *synchronization* between architecture model and system state, and identify four required properties for such synchronization, namely *consistency*, *non-interference introspection*, *effective reconfiguration*, and *stability*. These properties are necessary for management agents to use the synchronized architecture model for monitoring and controlling the system.

However, there are some challenges to implement such synchronization.

1. There is an abstraction gap between the system state and the architecture model. The

system state is determined by the implementation of the target platform, located in the solution space, while the architecture model is determined by the management requirements, located in the problem space. To represent the system in the proper perspective, the architecture model and the system state usually have heterogeneous forms and asymmetric contents, and thus it is not easy to determine the effect of architecture changes on the system side, and vice versa.

2. Since the architecture and the system are changing simultaneously, the synchronization has to deal the conflicts between architecture modifications and system changes.
3. The system modifications through the API do not always lead to the expected effects. The synchronization needs to handle such modification exceptions properly, in order to prevent the management agents from getting the inaccurate information for the running system and thus making wrong decisions.

In this paper, we present a generic approach to synchronize architecture models and running systems, satisfying the above properties. To address the above challenges, we use bidirectional model transformations to propagate changes across the abstraction gap between architecture and system, employ a two-phase execution to filter out conflicts in the changes, and employ a three-way check to identify modification exceptions. We provide a generative tool-set to assist developers in implementing this approach on a wide class of systems. Developers need only provide high-level specifications about the system and the required architecture style, including two meta-models to specify what constitutes the architecture model and the running system, a model transformation to specify their relation, and a declarative specification about how to retrieve and update the system state, and our tool-set automatically generates the required synchronizer to support RSA on this system. The contributions of this paper can be summarized as follows.

- We formalize the generic synchronization between architecture models and running systems, and define a set of required properties for such synchronization.
- We provide an architecture-system synchronization algorithm based on bidirectional transformation, satisfying the above requirements.
- We provide a generative tool-set to assist developers in implementing the approach on a wide scope of systems, supporting RSAs on them.

We have applied this approach to several practical systems, including JOnAS/JEE and

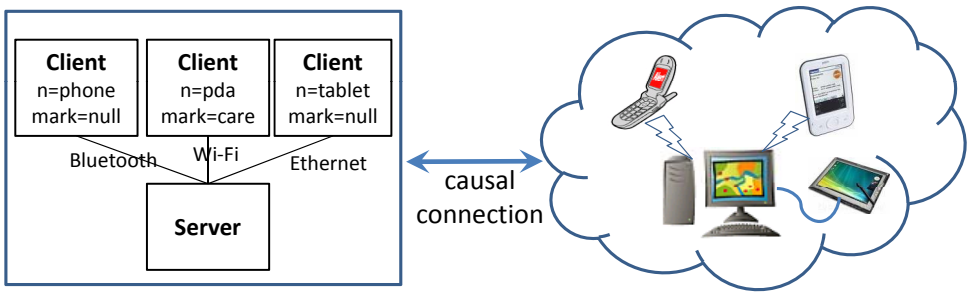

Figure 1: A client/server styled runtime architecture for a PLASTIC-based mobile system

PLASTIC[1]. These case studies demonstrate the feasibility, efficiency and wide applicability of our approach and tool-set.

This work is based on a series of our earlier approaches. We utilize the code generation approach for wrapping low-level management APIs (Song et al., 2009, 2010). The idea of using model transformation to achieve synchronization was discussed in Xiong et al. (2009b).

The rest of this paper is structured as follows. Section 2 illustrates the basic concepts of RSA. Section 3 discusses the synchronization for maintaining causal connection. Section 4 presents our synchronization approach based on bidirectional model transformation, and Section 5 introduces our generative tool-set to help implementing this approach. Section 6 describes our case studies. Section 7 summarizes related work and Section 8 concludes this approach.

2. Runtime software architecture

2.1. An illustrative example

The right part of Figure 1 shows a simple mobile computing system for file transmission, which we construct upon the PLASTIC Multi-radio Networking Platform (IST STREP Project, 2008). As shown in the figure, three devices are currently registered on a central desktop, which pushes files into these devices via different types of connections, including Wi-Fi, Bluetooth and wired cable. At runtime, the administrator may wish to *monitor* what devices are registered, and how they are connected. He also needs to *reconfigure* the system, e.g. when the Wi-Fi signal is too weak for the first device, he could switch the connection into Bluetooth.

[1]The tool-set and the artifacts used in the case studies can be found in our project website: `http://code.google.com/p/smatrt`

The left part of Figure 1 is an architecture model conforming to the Client/Server architecture style (Garlan et al., 2004). The `Clients` stand for the devices, the `Server` stands for the desktop computer, and the `Links` stand for the connections between them. Administrators could add or remove clients, change connection types. If they find a client that needs further maintenance, they add a be-careful mark on it..

To support the management activities mentioned above, the architecture model must have a *causal connection* (Blair et al., 1998) with the running system. The architecture model must change as the system changes. For example, if a device unregisters itself, the corresponding `client` element in the architecture will disappear. Similarly, the architecture modifications must cause correct system changes. For example, if the administrator switches a link type, the real network connection will be reset.

2.2. A formal description of runtime software architecture

The above example illustrates three elements of RSA, i.e., the architecture model, the system state, and the causal connection. This section discusses these elements in detail, with the help of a simple formal description.

2.2.1. Architecture models

Architecture models are constituted of a set of model elements (like clients and servers). These elements have attributes, and refer to other elements. The types of architecture elements in our example are shown in the left part of figure 2: A root element typed as `Structure` contains several `Servers` and `Clients`, which connect with each other through `Links`. An architecture configuration is a set of element instances conforming to these element types. Model manipulations (like adding a component or changing an attribute value) change the model from one configuration to another.

We use A to stand for the set of all possible architecture configurations (which is determined by the meta-model), and use $\Delta_A \subseteq A \times A$ to denote all possible changes from one architecture configuration to another. Following Alanen and Porres (2003), we present the model changes as a composition of primitive model modifications, including creating or deleting an element, and getting or setting a property. From this point of view, model differ ($- : A \times A \to \Delta_A$) finds a set of primitive modifications to represent the changes, and model merge ($+ : A \times \Delta_A \to A$) execute the modifications to get a new model. For architecture models, the effect of modifications is predictable, i.e., $\delta = a' - a \Rightarrow a + \delta = a'$.

2.2.2. Runtime system states

According to Sicard et al. (2008), a running system is constituted of system elements, like the devices and the desktop computer. The system elements may have local states (like

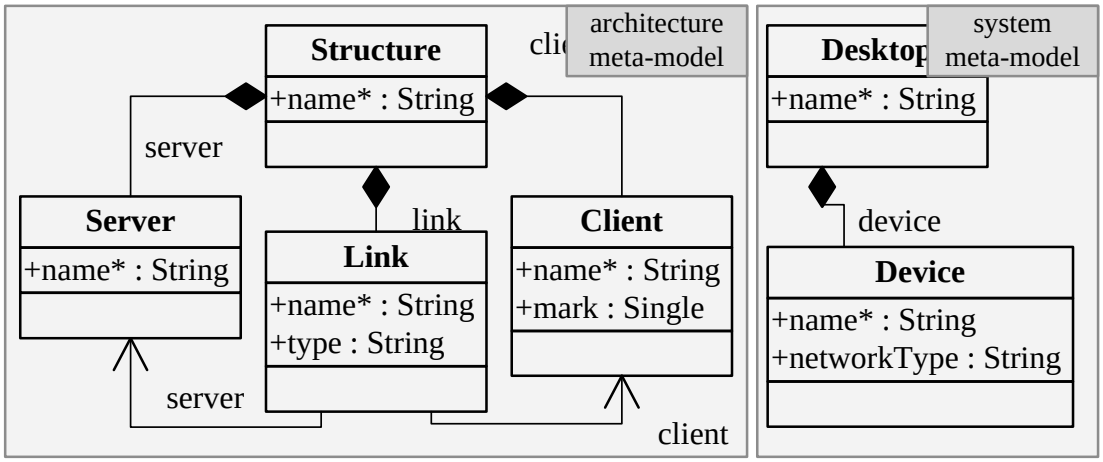

Figure 2: The definition of architecture and system elements for the running example

the connection type), or be associated with each other. The type of system states (like what elements exists, their local state values, and the references between them) can be also defined by a meta-model, as shown in Figure 2. Similar to architecture models, we use S and Δ_S to stand for the system meta-model and all possible changes. The changes may be caused by the system itself, or by manipulations from outside.

For the above example, reading and modifying the system state can be performed through the PLASTIC API. To see all the devices from the desktop computer, we could invoke the API on the desktop to get all registered `MNClients`, each of which stands for a device connected to this desktop. We can invoke `getActualNetworkQoS` on a `MNClients` to see its connection, and invoke `activateBestNetwork` on it to change the type[2].

Unlike architecture models, manipulations on running systems are *not always predictable*, i.e., $\delta = s' - s \not\Rightarrow s + \delta = s'$. The modifications may have *no effect* or *side effect*. For example, if we modify a device's network into Bluetooth but the device is suddenly outside the coverage of the desktop's Bluetooth signal, then this manipulation has *no effect*. If we delete (disconnect) the network between the desktop and the first device, then the device itself is also unreachable from the desktop, and thus the resulting system changes also include a *side effect* that is a deletion of the device. The reason for this is that the APIs usually do not reflect the complete system state. We abstract a running system as a tuple: (S, E, σ), with S, the system meta-model, E, the set of environment (e.g. the device is out of the Bluetooth range), and σ, the state transition function: $\sigma : S \times E \times \Delta_S \longrightarrow S$, standing for

[2] Originally, PLASTIC only open the interface for resetting networks according to a set of QoS requirements. To simplify our example, we altered it a bit to open the capability for switching networks directly by types.

```
transformation CS2PLA(arc:CS,sys:PLASTIC){                          relation
 key Structure{name};...key Device{name};
 top relation StructServer2Desktop{
  tmpName:String;
  enforce domain arc strt:Structure{name=tmpName,
    server=svr,client=clnt:Client{},link=lnk:Link{}};
  enforce domain arc svr:Server{name=tmpName};
  enforce domain sys dsktp:Desktop{name=tmpName, device=dvc:Device{}};
  where{ClientLink2Device(svr,clnt,lnk,dvc);}; }
 relation ClientLink2Device{
  tmpName:String; tmpType:String;
  enforce domain arc svr:Server{};
  enforce domain arc clnt:Client{name=tmpName};
  enforce domain arc lnk:Link{client=clnt,server=svr,type=tmpType};
  enforce domain sys dvc:Device{name=tmpName, type=tmpType}; } }
```

Figure 3: The relation between architecture and system

the system logic (e.g. if a network-link is removed, the device also disappears). For a current state $s \in S$, in the current environment $\epsilon \in E$, and after the execution of a manipulation $\delta \in \Delta_{S_M}$, the result $\sigma(s, \epsilon, \delta)$ is the subsequent state of this system.

2.2.3. Causal connections

Chan and Chuang (2003) and Sicard et al. (2008) refined the concept of "causal connection" into the following two requirements.

- **Correct Introspection.** No matter how system changes, the management agents could always get the *correct* system state through the architecture model.
- **Correct Reconfiguration.** Management agents could directly modify the architecture model, and the modifications will dynamically cause the *correct* system change.

Here the *correctness* depends on a given relation between the architecture configuration and the system state $R \subseteq A \times S$. When $(a, s) \in R$, we say that architecture a and system s are *consistent*, or a is a *reflection* of s. For our illustrative example, this consistency relation embodies the information like "if there is a device in the system, there must be a client in the architecture model with the same name, and vice versa". We illustrate this relation using a QVT transformation as shown in figure 3. It defines that the `Structure` and the `Server` together map to the `Desktop`; A `Client` and its connected `Link` together map to a `Device`.

3. Maintaining causal connections by architecture-system synchronization

We defined the Architecture-System Synchronization (ASS) according to a relation R, for a system σ, and under the environment ϵ, as a function:

$$Synch_{(R,\epsilon,\sigma)} : A \times A \times S \longrightarrow A \times S$$

The first two inputs are architecture configurations before and after management agent's modification. The third input is the current system state. The outputs are the synchronized architecture configuration and system state.

3.1. The four properties

In order to satisfy the two requirements above (i.e. introspection and reconfiguration), we use a series of *properties* to constrain the synchronization behavior. Specifically, for any $(a_o, a_c, s_c) \in A \times A \times S$ if

$$Synch_{(R,\epsilon,\sigma)}(a_o, a_c, s_c) = (a_s, s_s)$$

then we require the results (a_s and s_s) to satisfy the following propositions.

Property 1. *(Consistency)*

$$(a_s, s_s) \in R$$

First of all, we require the synchronized architecture configuration (a_s) and system state (s_s) to be consistent, so that Management Agents (MAs) could use the resulted architecture configuration to deduce the current system state and the modification effect.

Property 2. *(Non-interfering introspection)*

$$a_o = a_c \implies s_s = s_c$$

Consistency alone does not ensure correct introspection. For example, suppose that in a_o and a_c, the types of the first links are all `Wi-Fi`, and in s_c, the corresponding connection type has changed to `Bluetooth`. In this situation, the ASS could choose to change the real network type back to `Wi-Fi`. Although this result satisfies *consistency*, the MA will not get the *genuine* system state, but the one polluted by ASS. Therefore, we require that if the architecture is not modified, the ASS cannot change the system.

Property 3. *(Effective reconfiguration)*

$$a_c - a_o \subseteq a_s - a_o$$

Similarly, *consistency* is not enough for correct reconfiguration. For example, if the MA modifies the link from `Wi-Fi` to `Bluetooth` and in the current system the connection is still `Wi-Fi`, then to satisfy *consistency*, the ASS could ignore the architecture modification and leave the current system unchanged (that means the result is (a_o, s_c)). To ensure correct reconfiguration, we require that all the MA's modifications (i.e. $a_c - a_o$) *remain* in the final architecture change (i.e. $a_s - a_o$).

Property 4. *(Stability)*

$$(a_c, s_c) \in R \implies a_s = a_c \wedge s_s = s_c$$

Finally, we add an extra property to guarantee that, when the current architecture model and system state are already consistent, the ASS leave them unchanged. This property prevents irrelevant system changes from interfering the architecture model. It also allows the MAs to record some extra information on the architecture model. For example, the MAs could change the layout of the architecture model or mark some part of it to make it more intuitive, and since this change does not have any relation with the running system, this property ensures that the synchronization does not break the layout.

3.2. The challenges

There are several challenges to implement an architecture-system synchronization that satisfies the above properties.

First, the architecture model and the system structure are *heterogeneous* and *asymmetric*. *Heterogeneity* means that the relation between architecture and system is not a simple one-to-one mapping between architecture and system elements. In our example, the link elements in the architecture model do not have corresponding system elements. They just represent of the connection type of the devices. *Asymmetry* means that the architecture and the system may all contain some information which is not relevant to the other side. For example, the "be careful" mark on the client elements does not have any counterpart in the real system. Due to the heterogeneity and asymmetry, it is challenging to propagate changes correctly from the system to the architecture and vice versa. And moreover, according to the *Stability* property, we also have to identify the irrelevant information and keep it unchanged during the synchronization.

Second, the architecture and system changes may happen *simultaneously*, and thus these changes may conflict. For example, if the MA changes the type of the first link, and in the meantime, the first device is closed. Then if the ASS still propagates the architecture modification, it will invoke the management API to reset the network of this inexistent device,

and cause unexpected results. If such conflicts are not properly handled, the synchronization may even cause harmful invocations to the management API.

Third, the system modifications are not predictable for the ASS, because it cannot get complete system information from the management API. If the exceptions are not properly handled, the synchronization results may be inconsistent. For example, if the MA changes the first link to `Bluetooth`, the proper system change is to switch the connection type of the first device. If this switching operation fails, to ensure consistency, the ASS should catch the exception and roll back the modification of link type.

4. Architecture-system synchronization based on bi-transformation

This section presents our approach to implementing architecture-system synchronization. Aiming at the three challenges discussed in the last section, our main ideas can be summarized as follows.

- We utilize bidirectional transformation and model comparison to translate changes between architecture and system.
- We employ a *two-phase execution* to filter out conflicting changes, preventing them from harming the system.
- We add a *validating read* after changing the system to get the actual effects of the system modifications, in order to construct a consistent architecture model even in the presence of modification exceptions.

In this section, we first introduce the enabling techniques of our approach. Then we explain the algorithm on our illustrative example. Finally, we evaluate the algorithm according to the four properties (Section 3.1).

4.1. Enabling techniques

Bidirectional transformation. Bidirectional transformation uses *one* relation between two sets of models (i.e. two meta-models) to derive two directions of transformations between them. Formally speaking, according to Stevens (2007), for two meta-models M and N, and a relation $R \subseteq M \times N$, the bi-transformation is constituted of two functions:

$$\overrightarrow{R} : M \times N \longrightarrow N$$
$$\overleftarrow{R} : M \times N \longrightarrow M$$

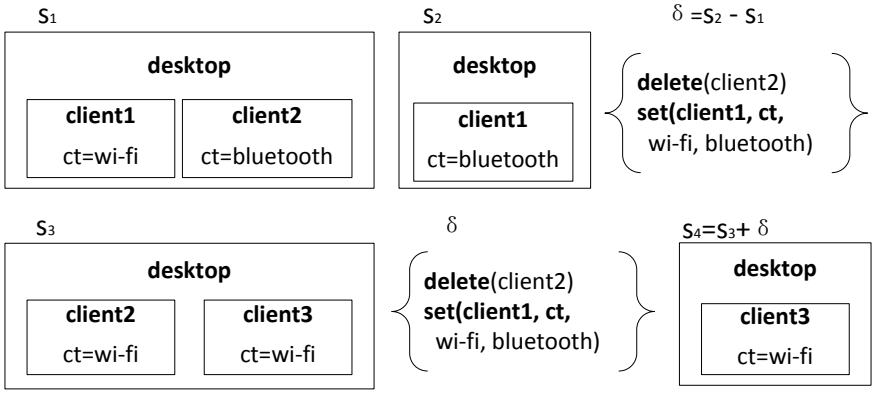

Figure 4: Sample for model difference

"$\overrightarrow{R}$ looks at a pair of models (m, n) and works out how to modify n so as to enforce the relation R: it returns the modified version. Similarly, $\overleftarrow{R}$ propagates changes in the opposite direction." (Stevens, 2007). Note that the transformation requires two parameters, because the relation between the two models is not bijective, i.e., for $m \in M$, there may exist more than one $n \in N$ satisfying $(m, n) \in R$. This is because each of the models may contain the information that is not reflected in the other one. Detail discussions and examples could be found in Czarnecki et al. (2009).

Model difference and merge. Model difference denoted by "-" compares two models to get the difference between them, i.e. $- : M \times M \longrightarrow \Delta_M$. It represents the calculated difference as a set of primitive model operations (Alanen and Porres, 2003). Model merge denoted by "+" executes the difference into a model to get a new model, i.e. $+ : M \times \Delta_M \longrightarrow M$. For example, the top half of Figure 4 shows two models and the difference between them, and in the bottom half, we merge this difference with another model: the `delete` operation eliminates `client2`, but the `set` operation has no effect, because the target element `client1` does not exists in this model.

System-model adaptation. The model transformation and difference techniques are based on standard models, usually the models conforming to the OMG's MOF standard (OMG, 2006). We assume the architecture models already conform to MOF standard, because more and more people choose MOF-based languages, like UML, to describe architecture models, and there are tools to convert architecture models in other languages into MOF-compliant ones (Cuadrado and Molina, 2009). But for system state, since most systems only provide ad hoc management APIs, we need a *system-model adapter* to support reading and writing the system state in a model-based way. Specifically, when reading, this adapter returns a MOF-compliant model that reflects the current system state. By contrast, when writing, it

Figure 5: Synchronization algorithm

Input: a_o and a_c, the original and current architecture model
Output: a_s the synchronized architecture, and δ_l the failed modifications
Side-effect: changing system state from s_o into s_c

```
1  Modification recognition:
2    s_c ⟵read adapter;  *get the system's "current state"
3    s'_o ← R→(a_o, s_c);  *get system model reflecting the original arch.
4    s'_c ← R→(a_c, s_c);  *get system model reflecting the modified arch.
5    δ_d ← s'_c − s'_o;  *the "desired system change" reflecting MA's modification
6  Two-phase execution:
7    s_d ← s_c + δ_d;  *attempt to execute the change" to the static model
8    δ_v ← s_d − s_c;  *the "valid change" that passes the attempt
9    adapter ⟵write δ_v  *executing the valid change to the system
10 Result feedback:
11   s_s ⟵read adapter  *retrieve the synchronized state  (s_s = σ(s_c, ϵ, δ_v) )
12   a_s ← R←(a_c, s_s);  *get the "final architecture"
13 Effectiveness Check:
14   δ_m ← a_c − a_o;  *get the MA's "modification"
15   δ_a ← a_s − a_o;  *get the "actual arch change" after synchronization
16   δ_l ← δ_m − δ_a;  *get the "lost change", and warn the MA
```

generates the proper invocation of the management API according to the model operation, changing the system state.

4.2. The synchronization algorithm

Figure 4.2 shows our algorithm in pseudo-code. It takes two architecture configurations before and after modification (a_o and a_c, respectively) as inputs. It propagates the architecture modifications into the current system, and reflects the new system state as the output architecture configuration (a_s). This algorithm has four steps. We first calculate what the architecture modifications mean on the system side. Then we use a two-phase execution to filter out conflicting changes, and execute the valid ones on the system. After the execution, we fetch the result system state, and feed it into the architecture. Finally, we check the result to see if all the MA's architecture modifications are successfully executed.

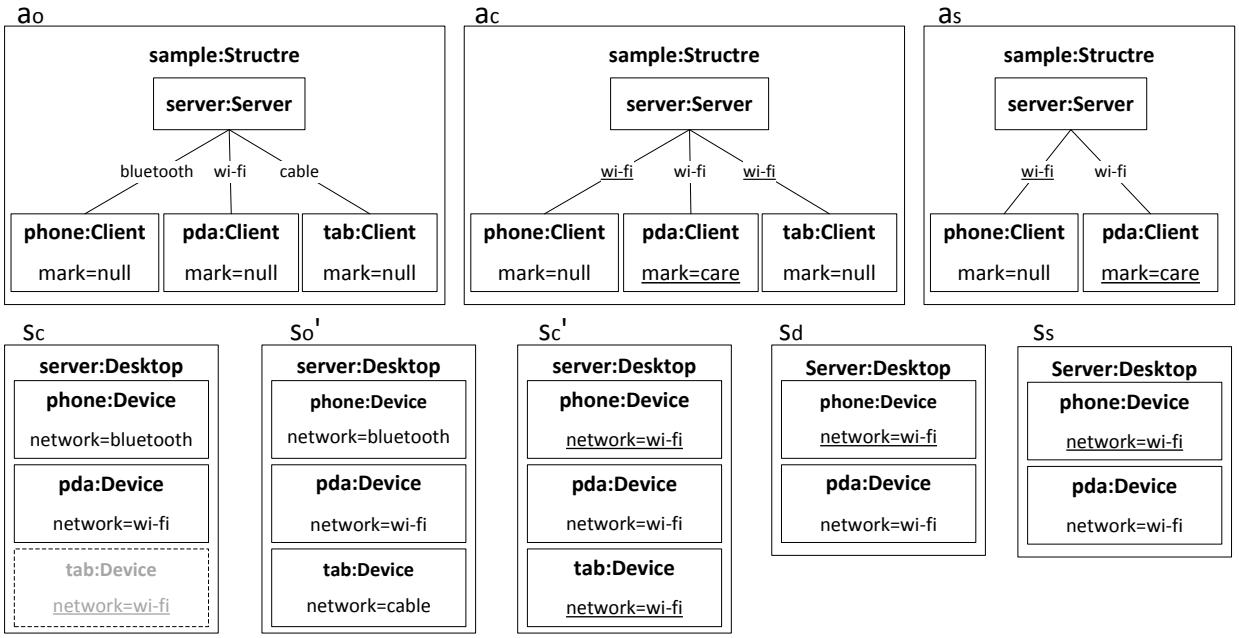

Figure 6: Sample architecture model and system state

To explain this algorithm intuitively, we use a synchronization scenario on our mobile computing system, as shown in Figure 6. The top half of this figure illustrates the evolution of the architecture configuration during the execution of this algorithm, and the bottom half illustrates the evolution of system state. The original architecture model a_o contains one server and three clients. The MA resets two links into Wi-Fi, and add a be-careful mark on a client. The modification result is a_c. In the meantime, the system has changed into s_c, with one device disappearing (shown by dashed lines and grayed text). Note that the architecture modification and the system change conflict, i.e., the MA changed the type of the third link, but the corresponding device (`tab`) does not exist in the system any more. We expect the resulting architecture model (a_s) to reflect the system change (a client deleted) and keep the non-conflicting architecture modifications (the type of the first link and the mark of the second one). The resulting system state (s_s) contain the effect of this architectural modifications (i.e. having the first network changed into Wi-Fi).

Step 1: Modification Recognition. Our first step is to recognize the meaning of the MA's architecture modifications on the system side. We first read the system-model adapter to get the current system state and preserve it as a system model s_c (as shown in Figure 6, marked as s_c). Then, we use this system model as a reference to transform the original and modified architecture models (i.e. a_o and a_c, respectively) into two system models (s'_o and s'_c). These two system models are not the representations of the real original and current

system states, but the images of the two architecture models. We compare these two images, and get the difference as follows:

```
[ set(phone, network, bluetooth, wi-fi),
  set(tab, network, cable, wi-fi) ]
```

This difference is the meaning of the MA's architecture modifications (resetting two link types) in the system side, and we name it as the *desired* system changes (δ_d) by the MA.

Step 2: Two-phase execution. Our second step is to execute the desired system modifications into the running system. Due to the conflicts between architecture and system changes, the desired modifications may contain some invalid modifications, like `setting` the network type of the `tab` device. We cannot directly execute these modifications into the system, and thus we employ a "two-phase" execution: the first phase filters out the invalid modifications, and the second phase executes the valid ones. Specifically, in the first phase, we execute the desired modifications on the system model s_c. Since this execution is not performed on the real system, the invalid modification does not harm the system. Moreover, according to the behavior of the model merge (Alanen and Porres, 2003), this modification does not change the model. After this "fake" execution, we get a system model which records the desired and valid system state by the MA (see s_d in Figure 6). We compare this model with s_c again, and get the valid modifications, i.e.

```
[ set(phone, network, bluetooth, wi-fi) ]
```

Finally, in the second step, we invoke the system adapter to execute this valid modification (δ_v) into the real system. Notice that currently we check the validity in a simple way: Checking if the modifications are syntactically meaningful to the current system state. We plan to introduce OCL into system meta-model to specify semantical constraints, in the future.

Step 3: Result feedback. Our third step is to propagate the original system change (the disappearance of the `cable` device) and the actual effect of the system modification (setting `phone`'s network to Wi-Fi) into the architecture model. We use the adapter again to read the system state. This state is the real modification result determined by the system state before synchronization (s_c), the system logic (σ) and the current environment (ϵ). If this modification is executed successfully, the retrieved system model is like s_s in Figure 6. Then we use the backward transformation to transform this final system state into the architecture side, as the resulted architecture model. In order to preserve the irrelevant architecture modifications (like marking the "pda" client), we use the modified architecture model (s_c) as the basis to perform this transformation.

Step 4: Effectiveness check. We cannot always ensure the *effective reconfiguration*

property (we discuss this later in Section 4.4). So we employ an extra step to check what architecture modifications have not been successfully propagated. We first compare a_o and a_c, and the difference is constituted of the original architecture modifications from the MA (δ_m).

```
[ set(link1, type, bluetooth, wi-fi),
  set(link3, type, cable, wi-fi),
  set(pda, mark, null, care) ]
```

Then we compare a_o with a_s, and the difference reflects the actual architecture evolution (δ_a).

```
[ set(link1,type, bluetooth, wi-fi),
  set(link3, type, cable, wi-fi),
  delete(tab) ]
```

Finally, we calculate the *relative complement* of δ_a in δ_s, to see which expected modifications do not remain in the actual effect:

```
[ set(link3, type, cable, wi-fi)]
```

We warn the MA about this "lost modification", so that the MA could choose to re-try this modification or to find a substitute solution.

4.3. Assumptions

Our algorithm depends on the following assumptions.

First, we assume that a pair of forward and backward transformations $\overrightarrow{R}$ and $\overleftarrow{R}$ satisfy two basic properties of bi-transformation (Stevens, 2007). The first property is *Correctness*. That means the relation R holds on the transformation results: $(m, \overrightarrow{R}(m,n)) \in R \ \wedge \ \overleftarrow{R}((m,n),n) \in R$. The second property is *Hippocraticness*. That means the transformations do nothing for the already consistent models: $(m,n) \in R \Longrightarrow \overrightarrow{R}(m,n) = n \wedge \overleftarrow{R}(m,n) = m$

Second, we assume the model difference and merge (like Line 14 or Line 7, but not for the reading and writing on system states through adapters) to be deterministic (Alanen and Porres, 2003): $\forall m, m' \in M, \delta \in \Delta_M . \delta = m' - m \Longrightarrow m' = m + \delta$ In addition, we also require the modifications to be idempotent, i.e. $\forall m \in M, \delta \subseteq \Delta_M . m + \delta + \delta = m + \delta$. For MOF-based models, to satisfy idempotency, we require the multiple properties to be unique and unordered (Xiong et al., 2009a).

Finally, we assume that the environment does not change during a synchronization process[3]. This assumption is not difficult to satisfy in practical situations. On the one hand,

[3]a duration after the MA modifies the architecture and launches the synchronization, and before they get the resulted architecture configuration. We do not require the environment to be stable during the time when MA modifies the architectures

most system changes concerned by MAs do not happen frequently, like components added or parameter changed. On the other hand, as a fully automated process, the synchronization spends much less time, comparing with the time for MAs to make their management decision. For the systems where the casual violation of this assumption is not acceptable, developers could utilize an environment lock before each synchronization process. This assumption does not prevent multi-objective management: Different management agents could utilize different RSA of the same system, and perform management activities simultaneously, providing that the synchronization processes do not overlap.

4.4. Discussion about the algorithm and the properties

We evaluate this algorithm according to the four properties we discussed before. In summary, this algorithm satisfies three properties *in any situations*, and satisfies "effective reconfiguration" *when the MA's modification intention is reachable at the current system.*

The algorithm satisfies Consistency*, i.e.* $(a_s, s_s) \in R$. After the final backward transformation (Line 12), a_s and s_s has the following relation: $a_s = \overleftarrow{R}(a_c, s_s)$. According to the "Correctness" property of bi-transformation, $(a_s, s_s) \in R$.

It satisfies Stability*:* $(a_c, s_c) \in R \Rightarrow a_s = a_c \wedge s_s = s_c$. Since a_c and s_c are consistent, i.e. $(a_c, s_c) \in R$, the "Hippocraticness" property of bi-transformation ensures that the forward transformation (Line 4) results $s'_c = s_c$. δ_d changes some state into s_c (Line 5), and thus executing δ_d on some state will also return s_c, so finally, $\delta_v = \phi$. Since we assume that the environment is stable, executing an empty modification will not cause the system to change, and thus we get $s_s = s_c$. For the other part of *Stability*, since $s_s = s_c \wedge (a_c, s_c) \in R \Rightarrow (a_c, s_s) \in R$, the "Hippocraticness" property also ensures the backward transformation does not change the architecture, and thus we get $a_c = a_s$.

It satisfies Non-interfering Introspection*:* $a_o = a_c \Rightarrow s_s = s_c$. $a_o = a_c$ means that the two transformations in Lines 2 and 3 has the same inputs. The deterministic transformation produces the same outputs, i.e. $s_o = s_m$, and thus in Line 5, $\delta_d = \Delta(s_o, s_m) = \phi$. Similar to the above discussion, this empty change will cause no effect on the current system, and so $s_s = s_c$.

It satisfies Effective Reconfiguration*, if the MA's desired system modification is reachable for the current system.* We first explain the premise. The desired system modification (δ_d in Line 5) is the intention of MA's architecture modification. We say the desired system modification is *reachable for the current system*, if we can successfully effect this modification in the current system. Formally, a reachable modification $\delta \in \Delta_S$ for the current system (s, ϵ, σ) must satisfies $\sigma(s, \epsilon, \delta) = s + \delta$.

Due to space limitation, we give only an informal proof for this theorem. We divide MA's

architecture modification ($\delta_m = a_c - a_o$ in Line 14) into two parts, say $\delta_m = \delta_{ms} \cup \delta_{mi}$. δ_{ms} is significant to the system (like changing the link type), and the desired system modification δ_d is the image of δ_{ms} in the system side. According to the premise, after the adapter reading and writing, the δ_d will be merged into the current system, and is contained in the resulted system state s_s. In the backward transformation, a_c contains δ_{ms} and a_s contains its image. This implies that the transformation does not need to break δ_{ms} to make a_s consistent with s_s, and thus a_s still contains δ_{ms}. The other part, the δ_{mi}, is insignificant to the system (like marking a device). These modifications will never break the consistency between architecture and system. According to the "Hippocraticness" property, the backward transformation will keep this modification in a_s. As a result, the whole δ_m remains in the resulted architecture a_s, and thus $a_c - a_o \subseteq a_s - a_o$.

In practical situations, we cannot ensure that the MA's modification is always reachable for the current system. First, MA needs a relatively long time to make modification decisions. During this time, the system may change and making the MA's modification outdated. Second, for some specific system logic and environment, the modifications will fail or cause side-effects, and MAs cannot predict that. It is usually a big burden if we constrain the MA to only perform reachable modifications. As a result, in this paper, we choose a simpler solution: we allow MA to perform any modification, and after synchronization, we inform them about the violations to this property (the "effectiveness check" step). Such violations help MAs understand the current system, and find reasonable modifications through attempts.

5. Generating synchronizers for legacy systems

We developed a tool-set named SM@RT to help developers in implementing our synchronization approach on different systems to provide runtime software architectures for them. As a generative tool-set (shown in Figure 7), from the developer's specifications about the system and the architecture (Layer 3), the tool-set (Layer 2) automatically generates the synchronizer (Layer 1) to maintain causal connection between the architecture and the system at runtime (Layer 0). We design this generative tool-set in two steps.

First, we provide a generic implementation of the synchronization algorithm discussed in Section 4. This engine is independent of architecture styles and running systems. To make this generic engine works for a specific legacy system and a specific architecture style, we need to customize it with the following artifacts (recall the algorithm in Figure 4.2):

1. the architecture and system meta-models that guide model comparison,
2. the relation between them to guide the transformations,
3. the system adapter for manipulating the system state, and

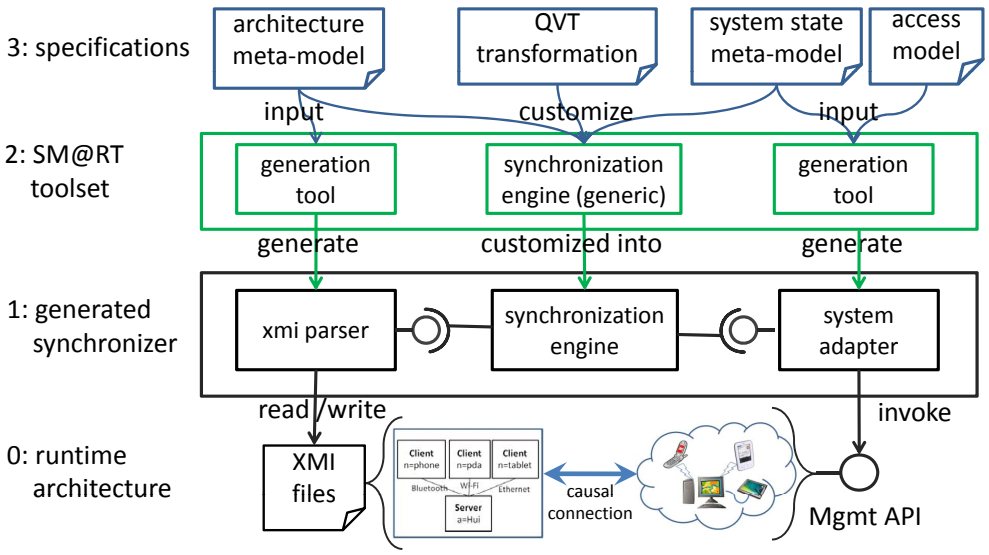

Figure 7: Overview of SM@RT toolset

4. the XMI parser to read and write the architecture model

Our second step is to assist developers in providing these customization artifacts. For the first two artifacts, we choose the MOF model and the QVT transformation language for developers to specify the meta-models and the relation, respectively. But since writing the adapters and the parsers from scratch is tedious and error-prone, we provide further assistance for the last two artifacts. We automatically generate XMI parsers from the architecture meta-model, and automatically generate the adapter from the system meta-model and a declarative specification of the management API, which we name as "access model".

In summary, the SM@RT tool-set has two main parts: a generic synchronization engine, and two generation tools. It also contains some auxiliary tools, like the graphical editor for specifying MOF meta-models and the textual editor for the access models. In the rest of this section, we briefly present our implementation of the two major parts.

5.1. Implementing the generic synchronization engine

We implement the synchronization algorithm in Figure 4.2 using a set of existing model processing tools based on Eclipse Modeling Framework (EMF: Budinsky et al. (2003)), which can be regarded as an implementation for Essential MOF (EMOF: OMG (2008), a core subset of MOF standard).

We choose an open source QVT transformation engine, the mediniQVT (ikv++, 2009), to implement the bidirectional transformations. mediniQVT is implemented on the EMF

framework, and uses the EMF generated Java classes to manipulate models. It is a complete QVT implementation, supporting the expressive power defined by QVT language, and satisfying the properties we require as assumptions.

We apply a model comparison engine that we have developed before (Xiong et al., 2009a) to implement the model different and merge. This comparison engine is also implemented on the EMF framework, conforming to the definition by Alanen and Porres (2003). As an experimental tool, our comparison engine has some constraints on the meta-models, i.e. any classes must have an ID attribute and the multi-valued references are not ordered.

To work for a specific system, this generic implementation can be customized by two meta-models defining the architecture and system, and a QVT transformation specifying their relation. The meta-models and the QVT transformation for our running example are shown in Figure 2 and Figure 3. The attributes marked with stars are the IDs of the classes.

5.2. Generating specific XMI parsers and system adapters

The generator for XMI parsers takes the architecture meta-model as an input, and produces the parser automatically. We implement this generator by directly reusing the EMF code generation facility.

The generator for system adapters takes as inputs the system meta-model and an "access model", and produces the adapter. This generator is an achievement of our previous work (Song et al., 2009), and in this paper, we just briefly introduce its input and output.

To generate the adapter for a specific system, we require developers to provide an "access model" to specify how to invoke the system's management API. An access model is a set of items, each of which defines a piece of code that implements a primitive manipulation operation (get, set ,create, etc.) on a specific kind of system data (Device, Network, ect). Figure 5.2 shows one of the items in the access model for PLASTIC, defining how to "*get* a *device's connection type*". The meta element (Line 2) indicates the type of target system element. The manipulation (Line 3) indicates the primitive operation, and the Java code (Lines 4-8) shows how to invoke the API. The logic for this API invocation is as follows: We get the instance of `MNClient` for this device, retrieve its QoS information, and return the network type from the information[4].

From the access model, our generator automatically produces the system adapter. The adapter maintains an EMF compliant model at runtime, and external programs (like our

[4]As mentioned before, we revise PLASTIC a bit to add "network type" as a new QoS value, and let the PLASTIC framework to choose network directly by type. Note that this revision is not a necessary part for applying our approach. We did it just for making *this example* straightforward

Figure 8: Excerpt of access model for PLASTIC

```
1  @Map
2  @MetaElement=Device::Network
3  @Manipulation=Get
4  @CodeFragment=@Begin
5    MNClient mnc=(MNClient)$core;
6    QoSInfo qos=mnc.getActualNetworkQoS();
7    $result=qos.getNetworkType();
8  @End @EndMap
```

synchronization engine) use the standard operations to manipulate this runtime model, like copying the runtime model to a common static model (the `read` operation in the algorithm in Figure 4.2), or executing the modifications to the runtime model (the `write` operation). In the background, the adapter synchronizes the model state with the system state at real time, so that the external programs could always get the current system state, and their modifications on the runtime model will immediately be executed to the real system. More details about this low-level real-time synchronization could be found in our previous paper (Song et al., 2009).

6. Case studies

We have applied our approach on several practical systems, providing RSA support for them. These cases illustrate its feasibility and validity, as well as the development efficiency for implementing it. In the following of this section, we first describe a C2-JOnAS case in detail. Then we present other cases briefly, and summarize all the cases.

6.1. C2-JOnAS

Our first case study is to provide C2-styled runtime architecture for JOnAS. Here JOnAS (OW2 Consortium, 2008) is an open source JEE application server, while C2 (Oreizy et al., 1998) is an architecture style aiming at the runtime evolution of UI-centric systems. Since many JEE applications are UI-centric, it is a natural idea to use C2-styled architecture models for managing JOnAS-based systems.

We prepare the four inputs as shown in Figure 9, to let the *SM@RT* tool-set generate the synchronizer. We defined the *architecture meta-model* (Figure 9(a)) following the description of C2 style (Oreizy et al., 1998), where `Architectures` contain several `Components` and `Connectors`, which link to each other through `above` and `below` associations. We defined the *system meta-model* (Figure 9(b)) according to the JOnAS document. We care about the

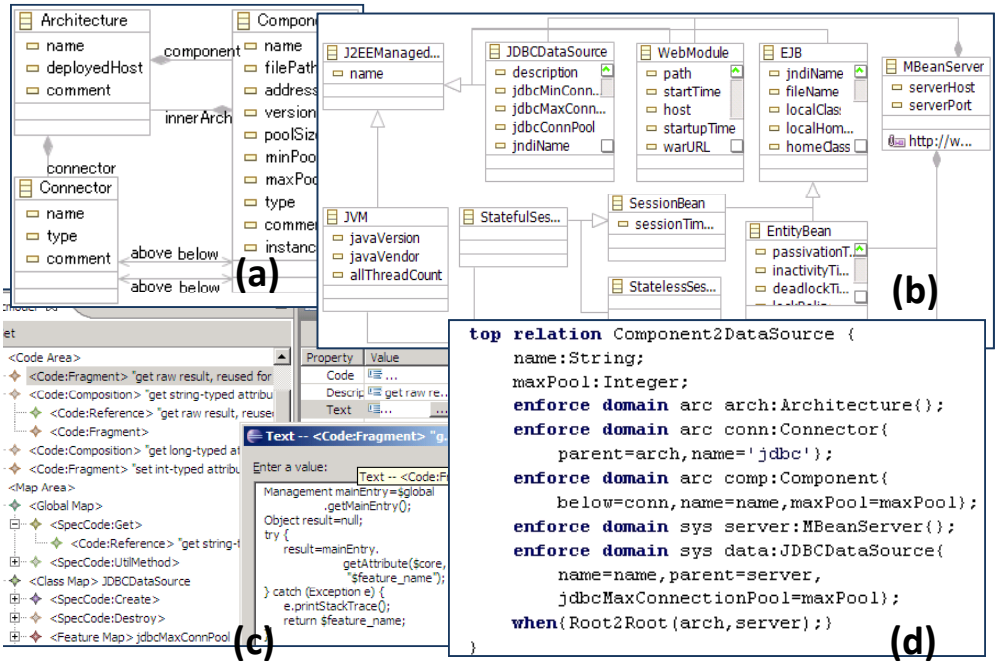

Figure 9: Specifications for generating C2-JOnAS synchronizer

`EJB`s, `JDBCDataSource`s and `WebModule`s running on a system, and a set of their attributes. We defined the *access model* (Figure 9(c)) by studying the sample code for using JOnAS management API (Hanson, 2004). We wrapped the Java code for deploying and un-deploying EJBs, data sources and web modules, and for getting and setting their attributes. The pop-up diagram shows a sample item for invoking `getAttribute` method of JMX to get all kinds of attributes. Finally, we defined a *QVT transformation* to connect the architecture and system meta-models. We use five QVT relations to reflect all types of management elements to components. Figure 9(d) shows one of these relations. It specifies that a `Component` maps to a `JDBCDataSource`, if and only if they had the same name, and the `Component` links to a `Connector` named "jdbc".

The generated synchronizer maintains a C2-styled runtime architecture for a JOnAS system. For this case, the target system is a JOnAS server deployed with a Java Pet Store (JPS) application (Sun, 2002). We launch the synchronization engine with an empty model as the initial architecture. After the first synchronization, we obtain an architecture model showing the current structure of the running JPS. The left snapshot of Figure 10 shows this model opened in a graphical C2 architecture editor, after a manual adjustment of the layout. At this time, the area inside the red dashed frame is empty. This system contains one component (`HSQL1`) to provide the data, several other components to organize and aggregate the data (such as `CatalogEJB` and `ShoppingCartEJB`), and finally one component to present

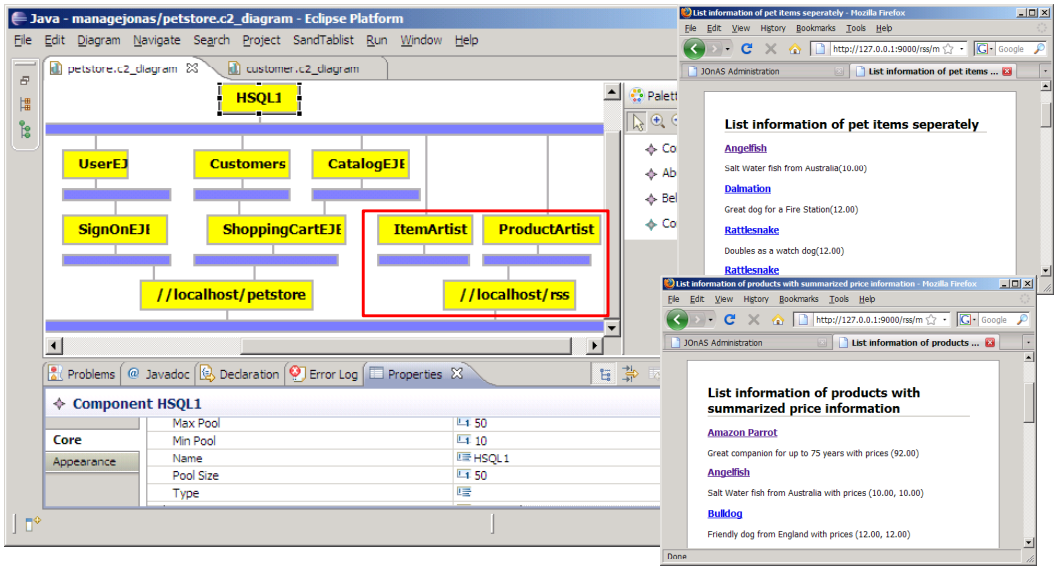

Figure 10: Snapshots of C2-based JEE management

the data (`petstore`). We use the following two management scenarios to further show how to use the RSA and how the synchronizer works.

We first use a simple experiment to show how to use this RSA to tune system parameters at runtime: We write a script to continuously request the `SignOn` component, and we soon notice that the `Pool Size` of `HSQL1` becomes 50, which means the data source's connection pool is full. So we change the `Max Pool` to 100 and launch the synchronizer. After a while, we launch the synchronizer again, and the `Pool Size` exceeds 50. That means the database's maximal pool size has been successfully enlarged. Then we set `Max Pool` to 20000, but after synchronization, this value becomes 9999 (the upper limit of connection pool supported by this version of HSQL), and we receive a notification warning us that the change did not succeed, suggesting us for further actions like rolling back the modification.

Our second scenario simulates the runtime evolution case used by Oreizy et al. (1998). We want to add RSS (Really Simple Syndication) capability into JPS at runtime to support subscription of pet information. Following the typical C2-based evolution scenario, we add `ProductArtist` and `ItemArtist` components for organizing the raw data as *products* (a product represents a pet breed Sun (2002)) and *items* (same breed of pets from different sellers are regarded as different items), respectively, and add the `rss` component for formatting the data as an RSS seed. These new components are shown inside the red box in Figure 10. We implement these components as two EJBs and one web module, and then launch the synchronizer, which automatically deploys them onto the JOnAS server. Now we can subscribe an RSS seed with all *items* via "http://localhost/rss" (top-right of Figure 10). After

that, we find the item information is too tedious, and want to see if the product information will be better. We just change the link above `rss` from `ItemArtist` to `ProductArtist`, and launch the synchronization again. The system behavior is changed immediately, and we get an RSS seed with different contents, from the same address (bottom-right of Figure 10).

6.2. Client/Server-JOnAS

The second case study is a combination of the first one and the running example. We provide a Client/Server-styled RSA for JOnAS, where the server represents the data source and the clients represent the EJBs interacting with this data source. The server and the clients all have a `resource` attribute, which represent the data source's max connection pool and the EJB's instances amount, respectively. This RSA is useful for database administrators to see how the data source interacts with other components.

To construct the synchronizer, we directly reuse the meta-model we have defined for Client/Server style (Figure 3), and the meta-model and access model for JOnAS (Figure 9). We write a new QVT transformation to specify that `Server` maps to `J2EEDataSource`, and `Client` maps to the `EJB` that depends on this data source, and their attributes map correspondingly. This QVT transformation has 56 lines in total.

We perform a self-adaptation scenario on this RSA, imitating the one presented by Garlan et al. (2004). We specify the self-adaptation rule using an extended version of OCL (Song et al. (2007)).

```
context Server do
let sum:Real=self.link->
          collect(e|e.client.consumption)->sum()
in sum > self.resource => self.resource <- sum
```

At the architecture level, this rule means that if a server's resource is less than the sum of all its clients' consumption, then enlarge this server's resource. We input this rule to the extended OCL engine, and execute "synchronize, OCL-execute, synchronize" every five minutes. The effect on JOnAS system is automatically enlarging the data source's connection pool, when the sum of EJB's instance size (an instance implies a potential database connection) exceeds the data source's maximal connection pool size.

6.3. Other case studies

Besides the running example on PLASTIC, and the above two cases on JOnAS, we also performed several other cases on different platforms, implementing different kinds of RSAs. We perform these cases based on the system adapters we have generated in our earlier work (Song et al., 2009).

Table 1: **Summary of the cases**. For each target system, we list the platform name, the size of system meta-model (the total number of model elements, including classes, attributes, associations), and the size of access model (lines of code). For the architecture, we list the name of style or ADL, and the size of architecture meta-model (for the last two cases, we reuse the existing ADL and tools, without defining meta-models). Then we list the size of QVT (lines of code), and the approximate upper bound of time spent (in seconds) for a single synchronization. Finally, we briefly describe the type of management activities we have tried on these RSAs.

	Target System			Architecture		QVT	Time	Usage
#	platform	mm	acc	style	mm		max	
1	PLASTIC	6	547	C/S	15	15	0.5	dynamic configuration
2	JOnAS	61	237	C2	29	157	2	runtime evolution
3	JOnAS	61	237	C/S	17	56	1	self-adaptation
4	BCEL	29	124	UML	-	139	6	reverse engineering
5	SWT	43	178	ABC	-	104	1	dynamic configuration

Jar-UML. We wrap the BCEL (Byte Code Engineering Library, http://jakarta.apache.org/bcel/) API to reflect the class structure inside a Jar file, and write a QVT transformation to map this Java-specific class structure with the UML class diagram. This case is a reproduction of the Jar2UML tool(http://ssel.vub.ac.be/ssel/research/mdd/jar2uml). It is a weakened case of RSA, since it only supports the introspection of class structures, without reconfiguration.

Eclipse-GUI. The target system for this case is any Eclipse window (views, editors, dialogs, etc.). We generate a system adapter to reflect the SWT widgets (buttons, text box, containers, etc.) constituting the window. The architecture model is a generic GUI model constituted of components connected with composition relations. Using this RSA, developers can change the attributes of the widgets dynamically, such as the text, the background color, etc. They can also add or remove widgets into the window at runtime.

6.4. Summary and discussion

Table 1 lists all the cases we have introduced in this paper. In this section, we summarize these cases to evaluate and discuss our RSA approach in three aspects. We first discuss the application scope of this approach, including what target systems it applies to and what kind of RSA usages it supports. Then we evaluate how this approach supports these RSA usages on the target systems, emphasizing on how the four properties (Section 3.1) are embodied in these cases. Finally, we show its practicability, i.e. it is easy to implement this approach on different systems, and the implementations have acceptable performance.

6.4.1. Application scope

This approach applies to a wide range of target systems. According to Table 1, we have applied it on four different kinds of systems, covering enterprise (JOnAS), desktop (SWT), and mobile computing systems (PLASTIC). Actually, whether it applies to a particular system depends on whether we can construct the adapter for this system, and Song et al. (2009) has revealed that we can generate adapters for a wide range of systems.

This approach supports typical RSA usages. The above cases reproduces several typical RSA usages presented in classical literatures (like the runtime evolution in Oreizy et al. (1998), and the self-adaptation in Garlan et al. (2004)) and the ordinary usages (like reversing the class structure of jar files or dynamically configuring the GUI). With this conclusion, we can expect that this approach has the potential to support a wide range of RSA usage.

The usage of RSA depends on the capability of the management API. We realize runtime evolution and self-adaptation on JOnAS, because its JMX API supports deploying components and configuring their parameters at runtime. On the contrary, we do not support changing the UML model reflected from Jar files, because BCEL does not support changing the class structures. That means the RSA usage is mainly limited by the capability of the management APIs. This limitation is reasonable, for the current goal of this approach is to help management agents to utilize the existing management capabilities of the target systems in an RSA-based way, but not to instrument the systems with new management capabilities. Actually, the usage is also influenced by the adapter and the QVT relation. However, we have showed that we can generate the strong enough adapters to wrap all the provided management capabilities (Song et al., 2009), and we also believe that as the standard transformation language, QVT is capable of relating the system with any architecture model (this could also be demonstrated weakly by the cases, because we define all the required relations as QVT rules, in very small sizes).

6.4.2. Evaluation about the four properties

This section evaluates if we provide the right RSAs for runtime management. We have define the *right RSA* by four properties, and proved that our approach satisfies these properties. The cases further demonstrate that these properties are necessary and important for RSA-based runtime management.

Obviously, in all the cases, the synchronization satisfies *Consistency*, otherwise the reasoning upon the RSA is meaningless. Notice that even the modification has exceptions, like we assign a too big value to the pool size in case #2, the resulted architecture is also consistent with the resulted system state. This ensures the management agents always plan their activities based on the genuine current system state.

Non-Interfering Introspection ensures that all the system changes are reflected immediately. Therefore, in the self-adaptation case (#3) the adaptation engine observes the situation of too many resource consumptions *on time*, to make proper changes

The synchronizer satisfies *Effective Reconfiguration* for the *reasonable* architecture modifications, like the change of maximal pool size to 100 and the addition of components. But the change of maximal pool size to 20000 is not effectively reconfigured to the system. This is not the fault of the synchronizer, but because we, as the administrator, performed an invalid modification. The synchronizer raises a warning to help us notice and analyze this failure.

Finally, the synchronizer satisfies *Stability*, and thus the tuning on the architecture layout (case #2) is preserved after the synchronizations. This property also allows the management activities to be performed step by step. Take the evolution scenario in case #2 as an example, we can first add a component without connecting it with any connectors. At this step, the relation is not broken, and thus we can still execute the synchronization to see the system changes, before we go on to finish configuring this new component.

6.4.3. Implementation efficiency and execution performance

This section discusses how much effort is required to implement RSA on a particular system, and how it works in practical environment.

The major benefit of our approach is that it enables developers to *efficiently* implement runtime architecture on existing systems. The case studies highlight this benefit. The RSA we provide for JOnAS is not a trivial one: It reflects 6 kinds of management elements (supporting adding and removing most of them) and 54 kinds of attributes (with 12 of them writable), but to implement it, we only defined four model-level specifications (as shown in Figure 9), with 90 model elements and 394 lines of code in total. The whole work takes us only 2 full days, from study, specification to debugging. The generated code contains 27024 lines of code. The other cases also use inputs in small sizes to generate complex code. By contrast, a case study in the Rainbow project costs 102K lines of manual code (Garlan et al., 2004), and the case in C2 project costs 38K lines of code (Oreizy et al., 1998). Although the cases and the contrasts are not directly matched, the code size at least reveals that our generative tool-set could save developers' effort. The efficiency depends on the complexity of RSA usages, as well as the type of the target platform. For commercial solutions, like JOnAS and SWT, we write small code for a big architecture meta model (bigger architecture supports more powerful management capabilities), in contrast with the academics prototype platform like PLASTIC.

The cases also reveal the potential of reuse supported by our approach. For case #3, we

reuse the system specifications on JOnAS (case #2), and add only two attributes on the C/S architecture meta-model (case #1). Reuse will further increase the efficiency to implement our approach.

The performance of this synchronizer is *acceptable* for human participant runtime management. We define *acceptable performance* with the reference to the existing and widely used runtime management tools. For JOnAS, the official runtime management interface is a web-based management console, the `jonasAdmin`. Using this default console, after each modification, the administrator has to wait about 1 second (depending on the environment) before the page is refreshed. For Java class structure, using eclipse JDT, it also takes about 1 second to expend the full structure of a medium-sized package (1000 classes). Since these tools are already widely used in practical runtime management, and also considering the delay of human decision, we regard the synchronization time around 1 second as *acceptable* for manual runtime management. From this point of view, case #1, #2 and #5 are acceptable. For case #1 and case #5, the average synchronization time for all kinds of architecture changes is 0.31 and 0.85 second, respectively. For case #2, the synchronization time varies for different kinds of architecture changes. For example, changing the pool size and adding the RSS components (case #1) take 0.75 and 1.37 seconds in average. Case #4 takes longer time: Constructing the UML model from scratch for a large Jar file (`bcel.jar` itself) with 1155 classes takes 5.62 seconds on average. The good news is that for such reverse engineering task, we do not often need to construct the system architecture. However, for automated management, like case #3, the performance is not ideal: The self-adaption loop takes 0.81 second in average. That means the self-adaptation loop has to be performed in a much longer interval, in order to avoid too heavy system burden. So currently, our approach is only proper to the automated management scenarios which handle the not-so-frequent system changes.

There are complex factors that affect the performance of synchronizers.

First, the execution time of synchronization process is constituted of the time spent on QVT transformation and the API invocations. The latter plays the leading role in the current cases. For example, the difference of the two scenarios in case #1 (0.75s vs 1.37s) is caused by the fact that deploying an EJB costs 0.6s while setting an attributes is almost transient. Similarly, case #4 takes so much time mainly because it had to invoke the BCEL for so many times to collect the information about the one thousand classes. Alternatively, the time spent by the model transformation is almost constant for the existing cases (between 0.5 to 1 second).

Second, the performance is affected by both the complexity (the size of meta-models and the QVTs) and the scale (the size the final architecture model for a specific scenario) of the

RSA. The current cases show that the scale is more important (case # 4 takes much more time than others). We have tested mediniQVT with complex QVT rules, and found that the execution time becomes unacceptable when the QVT rules reach 1000 lines. But so far, the RSA cases do not require so complex relations.

7. Related Work

Using software architecture for runtime management is a hot topic in the recent decade, and much work has been devoted to the high-level representation and utilization of RSA, and the low-level mechanisms for maintaining causal connection between an architecture model and a running system.

For the representation and utilization of RSA, Kramer and Magee (1990) first propose to represent system structures as *nodes* and *links*, and allow people to manage the system by adding, removing or replacing these nodes and links. Oreizy et al. (1998) propose a layered architecture style named C2 to support runtime evolution of GUI-centric systems. Garlan et al. (2004) propose using RSA for policy-driven self-adaptation, and their policies originate from the design time architecture constraints. Oreizy et al. (2008) surveyed many relevant approaches, and summarized several typical architecture styles. Huebscher and McCann (2008) also surveyed several approaches, focusing on the ones that use RSA for self-adaptation. In this paper, we do not focus on a specific representation or usage of RSA, but the generic approach to implement RSAs in different styles, supporting different usages.

Current approaches employ different mechanisms to maintain the causal connections between architecture models and running systems. We roughly classify the mechanism into three kinds, according to their degree of coupling with the target systems. First, some early approaches require the target systems to be developed with built-in RSA support. For example, Oreizy et al. (1998) require their target systems to developed under the Java-C2 framework. To use Fractal architecture at runtime (Bruneton et al., 2004), the system classes must implement the interfaces defined by Fractal. This requirement limits their applicability in practice. Second, some approaches allow the target systems to be developed under industrial standards, but enhance their runtime platforms (middlewares) with RSA mechanisms. These approaches are also known as "reflective middleware", covering many mainstream component models, like DynamicTAO (Kon et al., 2000) and OpenORB (Blair et al., 2002) for CORBA, and PKUAS (Huang et al., 2006) for JEE. The problem here is that these platforms are not yet well accepted in practice, and thus few existing systems are constructed on them. Third, some researchers try to insert probes and effectors into existing systems to collect runtime data, organize them as architecture model, and effect architecture

modifications (Garlan et al., 2004; Schmerl et al., 2006). But since most existing systems are not designed for code-level evolution, inserting code into them, if possible, is usually tedious and unsafe. Our approach is close to the third type in that we also seek to provide a generic mechanism for existing systems, but we choose a safer way, utilizing the low-level management APIs provided by the existing systems.

Another advantage of our approach is its ease of application: To bridge the abstraction gap between architecture and system, developers only need to provide a declarative specification about their relation. Some approaches embed similar ideas. Chan and Chuang (2003) allow developers to map detailed system events into abstract architecture events using a simple event composition language, and Schmerl et al. (2006) develop a more sophisticated language to map events. Taking the specification, their event transformation engines causally connect the architecture and system during runtime. In this paper, we choose model transformation language for specifying state-based (not event-based) relations between architectures and systems, which is proper to the way of manipulating system states through active APIs invocations (not by passive event notifications).

Our general solution for architecture-system synchronization has its root in the research on bidirectional transformation (Czarnecki et al., 2009) and model synchronization (Vogel et al., 2009). We applied this technique to a novel field, i.e. synchronizing a common model (the architecture) with a dynamically changing model (the system), and thus we meet some new challenges like conflicting changes.

8. Conclusion

In this paper, we presented a synchronization approach to maintaining runtime software architectures for a wide range of existing systems. We applied bidirectional model transformation to bridge the abstraction gap between architecture models and system states, and adapted bi-transformation to handle conflicting changes and identify modification failures. This approach satisfies a set of well-defined properties, i.e. *Consistency*, *Non-interfering introspection*, *Effective reconfiguration*, and *Stability*, which ensure the validity of the RSAs for runtime management. We also provided a generative tool-set to assist developers in implementing this approach on different running systems. We applied our approach to provide RSAs for some practical systems.

Our approach requires the target systems to have low-level management capabilities, usually some kinds of management APIs. It could also utilize other forms of management capabilities, such as configuration files, system commands, etc. provided that people could define how to manipulate them as pieces of code. But in this paper, we use only man-

agement APIs as examples. Since runtime management becomes an important concern for modern systems, more and more systems provide such low-level management capabilities. Our approach is an effort to link such existing low-level management capabilities with the research on architecture-based runtime management. An issue here is how to help developers determine if the management API is sufficient for a particular architectural adaptation. We have an approach to analyze the capability of system API (Song et al., 2010), and we plan to provide a QVT analysis support to find out if the architecture operations used by the architecture adaptation is included in API capability.

Currently, our approach cares only about the structural part of RSA. That means after each synchronization, the resulted architecture configuration is a snapshot of the current system state. In future, we will investigate how to analyze a series of such snapshots to obtain the behavioral models for the system.

References

Alanen, M., Porres, I., 2003. Difference and union of models. In: UML. pp. 2–17.

Blair, G., Bencomo, N., France, R., 2009. Models@ run.time. Computer 42 (10), 22 –27.

Blair, G., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., Parlavantzas, N., 2002. Reflection, self-awareness and self-healing in OpenORB. In: The first workshop on Self-healing systems. pp. 9–14.

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

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

Budinsky, F., Brodsky, S., Merks, E., 2003. Eclipse Modeling Framework. Pearson Education, project address: http://www.eclipse.org/modeling/emf.

Chan, A. T. S., Chuang, S.-N., 2003. MobiPADS: a reflective middleware for context-aware mobile computing. IEEE Trans. Softw. Eng. 29 (12), 1072–1085.

Cuadrado, J., Molina, J., 2009. A model-based approach to families of embedded domain-specific languages. IEEE Transactions on Software Engineering 35 (6), 825–840.

Czarnecki, K., Foster, J., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J., 2009. Bidirectional transformations: A cross-discipline perspective. Theory and Practice of Model Transformations, 260–283.

France, R., Rumpe, B., 2007. Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE) in ICSE '07. pp. 37–54.

Garlan, D., Cheng, S., Huang, A., Schmerl, B. R., Steenkiste, P., 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37 (10), 46–54.

Hanson, J., 2004. Pro JMX: Java Management Extensions.

Huang, G., Mei, H., Yang, F., 2006. Runtime recovery and manipulation of software architecture of component-based systems. Autom. Softw. Eng. 13 (2), 257–281.

Huebscher, M. C., McCann, J. A., 2008. A survey of autonomic computing—degrees, models, and applications. ACM Comput. Surv. 40 (3), 1–28.

ikv++, 2009. medini QVT, http://projects.ikv.de/qvt.

IST STREP Project, 2008. PLATIC Multi-radio device management - developer guide. http://www.ist-plastic.org/.

Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhã, C., Campbell, R., 2000. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In: Middleware. pp. 121–143.

Kramer, J., Magee, J., 1990. The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. 16 (11), 1293–1306.

Kramer, J., Magee, J., 2007. Self-Managed Systems: an Architectural Challenge. In: Future of Software Engineering (FOSE) in ICSE. pp. 259–268.

OMG, October 2006. Meta object facility (mof) core specification. Available Specification ptc/04-10-15, OMG.

OMG, 2008. Catalog of OMG Modeling and Metadata Specifications. http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

Oreizy, P., Medvidovic, N., Taylor, R. N., 1998. Architecture-based runtime software evolution. In: ICSE. pp. 177–186.

Oreizy, P., Medvidovic, N., Taylor, R. N., 2008. Runtime software adaptation: framework, approaches, and styles. In: ICSE, Companion version. pp. 899–910.

OW2 Consortium, 2008. JOnAS Project. Java Open Application Server. http://jonas.objectweb.org.

Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., Yan, H., 2006. Discovering architectures from running systems. IEEE Trans. Softw. Eng. 32 (7), 454–466.

Sicard, S., Boyer, F., De Palma, N., 2008. Using components for architecture-based management: the self-repair case. In: the 30th International conference on Software engineering (ICSE). pp. 101–110.

Song, H., Huang, G., Xiong, Y., Chauvel, F., Sun, Y., Mei, H., 2010. Inferring Meta-Models for Runtime System Data from the Clients of Management APIs. In: Proceeding of MODELS 2010, LNCS 6395, to appear.

Song, H., Sun, Y., Zhou, L., Huang, G., 2007. Towards instant automatic model refinement based on OCL. In: APSEC. pp. 167–174.

Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H., 2009. Generating synchronization engines between running systems and their model-based views. In: Models in Software Engineering, the MoDELS Workshops (LNCS 6002). pp. 140–154.

Stevens, P., 2007. Bidirectional model transformations in QVT: Semantic issues and open questions. In: MoDELS. pp. 1–15.

Sun, 2002. Java PetStore. http://java.sun.com/developer/releases/petstore/.

Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B., 2009. Incremental Model Synchronization for Efficient Run-Time Monitoring. Models in Software Engineering, the MoDELS Workshops (LNCS 6002), 124–139.

Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H., 2009a. Supporting automatic model inconsistency fixing. In: ESEC/FSE. ACM, pp. 315–324.

Xiong, Y., Song, H., Hu, Z., Takeichi, M., 2009b. Supporting parallel updates with bidirectional model transformations. In: ICMT. pp. 213–228.