GRACE TECHNICAL REPORTS

A Model-Driven Framework for Constructing Runtime Architecture Infrastructures

H. Song Y. Xiong Z. Hu G. Huang H. Mei

GRACE-TR-2008–05 Dec. 2008

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

A Model-Driven Framework for Constructing Runtime Architecture Infrastructures

Hui Song*† Yingfei Xiong‡
Zhenjiang Hu†‡ Gang Huang* Hong Mei*

* Key Laboratory of High Confidence Software Technologies
Ministry of Education, Peking University, Beijing, China
{songhui06,huanggang,meih}@sei.pku.edu.cn

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

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

December 2008

Abstract

Maintaining software systems at runtime becomes more and more important, and many researchers are considering the use of architectural models for runtime management. For the research and application of architecture-based runtime management, one unavoidable issue is how to maintain the causal connection between the architectural model and the system state, which are heterogeneous in structures. To address this issue, researchers have proposed and implement many runtime architecture infrastructures to maintain causal connection between the specific architecture and system concerned in their approaches, but as there are so many kinds of architectural models and systems, a framework is necessary to assist subsequent researchers or developers in constructing such infrastructures between them. In this paper, we report a model-driven framework, along with supporting tools we implemented, for constructing runtime architecture infrastructures. With the help of code generation and bidirectional model transformation, we support developers to construct infrastructures by just writing two MOF meta-models and a QVT transformation between them. We evaluate the usage and effectiveness of our framework through a case study for achieving the runtime management of JOnAS systems based on C2-styled architectural models.

1 Introduction

To adapt to the ever changing requirement and environment during its long and non-stoping life cycle, a software system demands to be manageable at runtime. On the one hand, researchers are designing effective mechanism to enable such

runtime management [5], and many of the mainstream platforms provide some form of management capabilities to enable external management agents (either human administrator or software-based agents) to monitor and control it at runtime [14, 21]. On the other hand, on the research of different forms of management activities, such as human-directed runtime evolution [18] or automatic self-adaptation [11], many researchers are considering abstracting the system states into some kind of models [10], especially architectural models. Architectural models shield out the complex and technical details, and provide an abstract and usually domain-specific view of the running system states. Management agents can monitor and control the systems by reading and writing their architectural model in proper forms.

There are many research approaches towards architecture-based runtime management, focusing on different issues, from low-level mechanism [5, 24, 13] to high-level management assistance [6, 11, 18], but one issues is unavoidable for all these approaches, i.e. how to make sure the architectural model reflects the current system states and in the mean time the system states will be changed correctly according the architectural changes [10, 3]. Some researcher name this issue as "maintaining the causal connection between architectural model and system states" [5, 13, 3], and name the particular parts in their approaches for maintain causal connections as "runtime architecture infrastructures"[18]. We follow these two name in the rest of this paper to simplify the discussion. Although it is not the main concern for all the approaches mentioned above, researchers of all these approaches did propose different but effective methods for maintaining causal connections, and implement corresponding infrastructures, usually specific to the kinds of architecture models and running systems they chose.

Only individual infrastructures are not enough. There are many different kinds of running systems with different management capabilities, and there are also many architecture styles, fitting for different domains and management capabilities. If a developer wants to provide a proper kind of architectural models for managing a specific kind of running system, she usually has to first put much effort on implementing an infrastructure to connect the architecture and the system, even if this is not her main concern. And moreover, these particularly constructed infrastructures are often inevitable to have some kinds of little defects, and are often not easy for maintenance. It is also a hard task for developers to demonstrate the correctness of their infrastructures to the users or subsequent developers. Having noticed this problem, some researchers begin to consider a framework to help constructing such infrastructures.

Existing frameworks, as far as we know, still have some limitations. The authors of Rainbow [11] summarized a common structure of architecture-based self-adaptable systems, and identified the reusable parts, but they did not provide automatic supports for constructing all these parts; In their succeeding work [20], they provide a high-level language for specifying the relation between system and architecture, along with an engine to execute this language, but their language and engine are unidirectional, which cannot propagate the architecture change into running system; Genie [4] supports a model-based approach for developing self-adaptive systems, but currently this tool is specific to one kind of running systems, i.e. the systems running upon the Gridkit reflective middleware platform.

In this paper, we report our initial attempt towards a framework to support

constructing runtime architecture infrastructures in a model-driven approach. By utilizing bidirectional transformation [9, 22] and model difference [1] in a well-designed process, we support the maintaining of causal connections in both directions. And moreover, by utilizing the standard model-driven techniques of MOF, QVT and code generation, our framework can be used between different kinds of architectures and systems, in a unified, model-driven way.

The advantages of our framework can be summarized as follows.

- Our framework can improve the productivity. Developers can construct an infrastructure by specifying two MOF meta-models, writing a QVT transformation and supplementing a small quantity of code. The mechanism for maintaining the causal connection is transparent to developers. We will demonstrate the improvement on productivity in our cased study.
- The behavior of the obtained infrastructures is clear, predictable, and stable. For a constructed infrastructure, the architecture style, the system management capability and consistency relation between them are all specified explicitly, under formal and standardized languages, and we also prove that the constructed infrastructures preserve some key properties. In addition, the clear behavior also makes our framework good for maintenance and incremental development.
- We separate the development into three relatively independent concerns. When a developer is in charge of specifying the system states, he or she does not need to care about the architecture and the relation. This makes the specification of one kind of systems, along with the filled hook methods, reusable for different architecture styles. It is also the case for specifying architectural model and the relation.

The rest of this paper is organized as follows. Section 2 gives an overview of runtime architecture infrastructures, and how our framework supports constructing such infrastructures. Section 3 and Section 4 introduce how we support specifying and accessing architectural models and system states in a unified, model-driven way. Section 5 discusses how to specify the relation between architecture and system using QVT relational, and how we achieve maintaining causal connections according to such QVT specification. Section 6 presents a case study for supporting managing JOnAS [16] systems based on C2 styled architecture [18].

2 Runtime Architecture Infrastructures

Before introducing our framework, we would like to give an overview of its targets, i.e. runtime architecture infrastructures. We first use a simple example to illustrate the approaches of architecture-based runtime management. Based on this example, we will discuss the common structure and requirements of runtime architecture infrastructures to support such approaches. At last, we discuss the technical issues for implementing an effective infrastructure, and introduce what support our framework will provide for these issues.

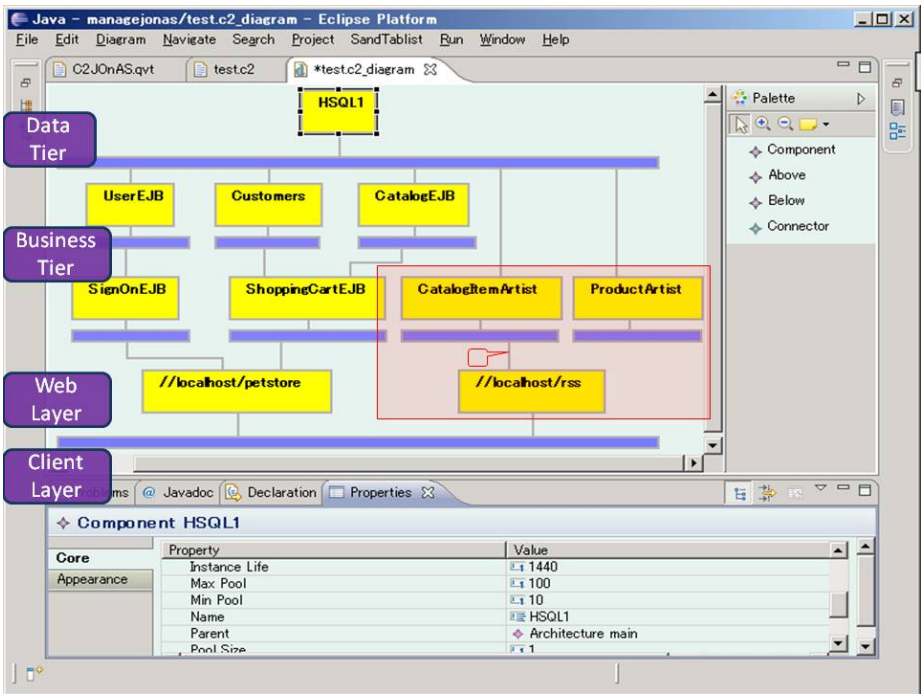

Figure 1: A snapshot of using C2 architecture to manage JPS on JOnAS

2.1 An example of architecture-based runtime management

We consider a typical example of architecture-based runtime management, i.e. using a C2-styled architecture model to manage Java Pet Store (JPS) [15] running on JOnAS application server [16]. JPS is a J2EE blueprint application implementing an online retail website, and JOnAS is a widely used open source J2EE application server. JOnAS provides powerful management capability under JMX standard [14] for monitoring and configuring systems states, and deploying or undeploying components (EJB, Database, Services, etc), at runtime. By default, maintainers can manage the applications on JOnAS (like JPS) by writing Java code to directly access the management API, or manipulate a build-in web-based user interface. These two ways are powerful enough, but not the best ways for every situation. If a maintainer is familiar with online shop (in problem space), but not familiar with J2EE standards (in solution spaces), these two ways are difficult for him to grasp. Architecture-based management is a better choice in such situations. Researchers are working for years on an architecture style named "C2" [18], which is proper for design and runtime management of GUI oriented systems like JPS.

Figure 1 is a snapshot of a simple graphical modeling tool, with an architectural model of JPS in C2 style. The left part of the architectural model (outside the red box) abstracts the original login and shopping relevant parts of JPS. Ideally, this architectural model should be obtained from design time, but as the JPS developers did not provide such a concrete model, we first construct one according to JPS documents. Thanks to C2 style, this model clearly embody the

four layers of web-based applications as marked in the snapshot, and it is also a hierarchical model (the "Customers" component encapsulate some components dealing customer information). These features make it easy for maintainers to understand and manipulate.

To use this architectural model for runtime management, we first develop a runtime architecture infrastructure. Now maintainers can launch a "synchronize" command to ask this infrastructure to synchronize the architectural model and the system state. After the first synchronization, some components will reflect the current state or configuration information of their corresponding implementation elements (EJB, Database...). For example, through the bottom part of Figure 1, maintainers can see that the current busy connection to this data source is 1, and the size of connection pool is configured between 10 and 100. Maintainers can change some value on this window, e.g. reduce the max pool size to 50, and after launching "synchronization", the max pool size of the database will be changed at runtime. The maintainers can also use this architectural model for runtime evolution. For example, suppose they would like to supplement an RSS function to the JPS system, he can add new components, like the ones in the red frame in Figure 1, just in the same way as supplement a design model. Then they can provide this new architectural model to some J2EE experts for developing corresponding EJBs or Web Modules. Finally, they can add some necessary information to these components, like names and file paths pointing to development result, and launch "synchronization" again. Now the implementation EJBs and Web Modules are automatically deployed into the running system, and users can use the url "http://localhost/rss" to retrieve an RSS seed with all pet item information. If the maintainer think the current RSS contents not satisfying, they can redirect the link from "//localhost/rss" component to the connector under "ProductArtist" component, and after synchronization, requesting the "http://localhost/rss" url will obtain an RSS seed with different contents.

During the whole management process, maintainers do not have to care about the J2EE specific techniques and concerns, as well as many irrelevant details like the many middleware services and components for other applications. This example is just a simple illustration, and detailed discussion about the advantage of using C2 styled architecture for runtime evolution can be found in literature [18]. And moreover, using C2 styled architecture model for manual evolution is just one kind of architecture-based runtime management. We can also provide architectural models in other styles as different views for maintainers with different concerns, and by employing the technologies presented in literature [6], we can also help achieve automatic management. The precondition is that we have effective runtime architecture infrastructures to connect these architectural models with the running systems.

2.2 Requirements for runtime architecture infrastructures

From the above example, we can go on to discuss the requirements for runtime architecture infrastructures. We first derive the the main capabilities a runtime architecture infrastructure should have, then we present some facts which have will affect the implementation of infrastructures, and finally we will discuss the meaning the causal connection by listing a set of properties.

Figure 2 shows a common structure of architecture-based runtime manage-

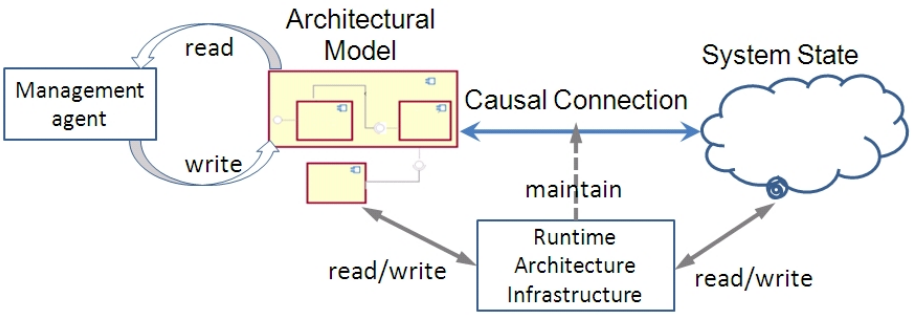

Figure 2: A common structure of architecture-based runtime management

ment approaches. Management agents perform the system only by reading and writing the architectural model, through some kind of interfaces or modeling tools. To ensure these management activities effective on the real system, there needs to be a causal connection between the architecture and the system, and the infrastructure is in charge of maintaining this causal connection. In particular, the infrastructure will read the architectural model and the system state, calculate the modifications needed on both of them to maintain the causal connection, and finally modify the architectural model and system states according to its calculation result. This is a common structure, and literatures [] also discussed about similar structures. From such a structure, we can summarize the three main capabilities of runtime architecture infrastructures, i.e. manipulating the architectural model, manipulating the system state, and calculating how to maintain the causal connection.

Besides the structure and capabilities, we would also like to clarify some facts of architecture-based runtime management, and these facts have significant influence on the implementation of runtime architecture infrastructures.

- *Fact 1.* The structures of architecture model and the system state are not isomorphic. In the above example, the architectural model are hierarchical, but in the running system, all EJBs locate in the same layer. Another example is that elements with the same type of "Component" may reflect different kinds of elements in the system. The essential of this heterogeneity is that the architectural model and system states respectively locates in problem space and solution space

- *Fact 2.* Both architectural model and system state may contain information which are irrelevant to the other side. On the one hand, as architectural model is usually an *abstract* view to the system state, there is certainly some information in system states which is not reflected. On the other hand, the architecture model may contain some information for easy to understand, like the comments on components or the layout information between components, and such information will not relevant to system state.

- *Fact 3.* Management activities may not always lead to the expected effect on the system. For example, if a maintain set the MaxPoolSize of the data source with a very big number, the actual max pool size of the database

after synchronization will not the be the same number as the maintainer have given. This uncertainty origins from the fact that the architectural model abstracts out many complex relations exists in the running system.

In the end of this sub section, we emphasize on the most important part of the requirements, i.e. the properties of maintaining causal connections, or in other words, what is the proper behavior of runtime architecture infrastructures.

Commonly, the task of maintaining causal connection is simply defined as to ensure that the architecture model is an ongoing representation of the running system, that means the architecture should change as system changes, and vice versa. To discuss this task more strictly, we introduce a consistency relation between architectural model and system state, and if the current architectural model and the system state satisfy this relation (or in other word, if they are "consistent"), we say that this architectural model reflects the current system state.

- *Property 1. Synchronization leading consistency.* First of all, when management agents launch the synchronization command, they definitely expect that the resulted architectural model and system state are consistent, otherwise, they cannot get the correct system state or the system state does change as they want.

- *Property 2. Non-interference reading.* In order to check the fresh state of system by reading architectural models, management agents have to launch synchronization before reading. In such situations, the management agents make no change on the architectural model,and the infrastructure should not do anything to interfere the running system and make its state change.

- *Property 3. Effective writing* If management agents make some changes on the architectural model and then launch synchronization, he wish that these changes remain in the resulted architectural model. That means the system has been changed properly according to these architectural changes. Note that according to Fact 3, sometimes this property may conflict to Property 1, and in such situations, the infrastructure should make the agents aware of that.

- *Property 4. No insignificant change* Sometimes, before synchronization the architecture and the system are already consistent. That may be because there are no changes on architecture and system, or the changes are all within the scope of irrelevant information (see Fact 2). In such situations, the infrastructure should not change any of them.

2.3 Constructing a runtime architecture infrastructure

In short, constructing a runtime architecture infrastructure is the work of implementing the 3 tasks.

The first two tasks, i.e. manipulating architectural model and running system are relatively easier. For architectural model, developers can choose some existing tools to read and write the model files usually in XMI format, and for system state, developers can write code for accessing the management interface

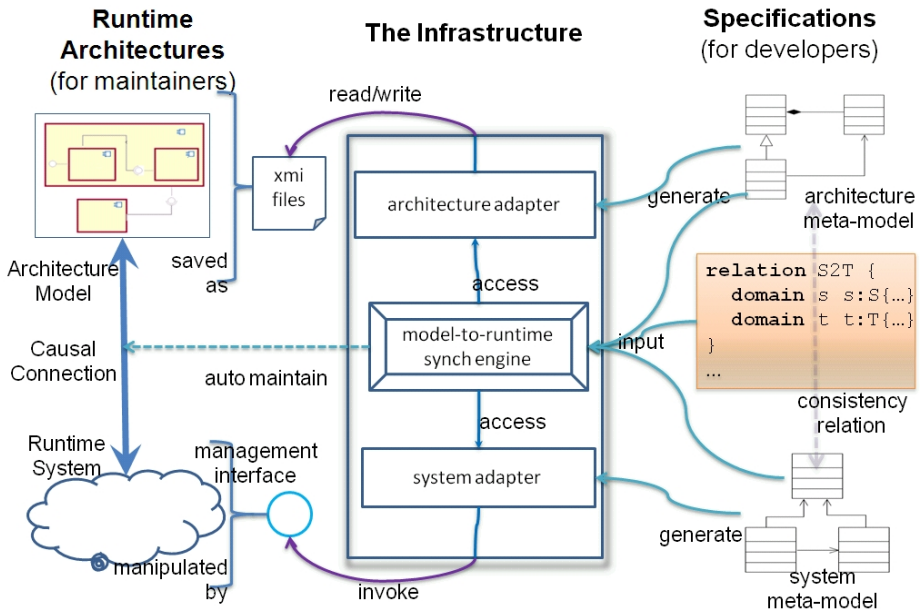

Figure 3: Approach overview

provided by the system, following some specific accessing method. The problem here is that the code for implementing these two tasks should be independent from the code for maintaining causal connection, otherwise, considering all these things together may make the logic very complex, and the constructed infrastructure will be hard to maintain and reuse. Therefore, developers have to decide either to consider all things together, sacrificing the maintainability and reusability, or take some time to define clear and reusable interfaces between the three tasks. The latter choice itself is a non-trivial technical issue.

The third task is more difficult. First, according to Fact 1, calculating how to maintain causal connections is more complex than "comparing two piece of data, since the heterogeneity between architecture and system must be considered. Moreover, developers also have to consider the extra information (according to Fact 2), and try to deal with situations when the management activity failed (according to Fact 3). Finally and most importantly, to provide a effective causal connection infrastructure for management agents, developers also have to satisfy the four properties, and prove this satisfaction to the human maintainer who will use this infrastructure, or the subsequent developers who will develop the management agents based on this infrastructure.

From the above discussion, we can see that developing an effective runtime architecture infrastructure manually is complex and tedious. Nowadays, model-driven engineering has been wide-accepted as an approach to liberate developers from such tedious manual work. In this paper, we report a model-driven approach for constructing runtime architecture infrastructures, along with a framework to support such an approach.

Figure 3 is an overview of this model-driven approach. The left part shows an architectural model and a running system used for architecture-based runtime management, and the middle part (discarding its inner structure right now) is

the infrastructure for maintaining causal connection between them. Here we make two assumptions. First, the architectural models are saved as XMI files. XMI is an OMG standard format for restoring models, so this assumption is reasonable. Second, the running system has provided a management interface for external software to manipulate its runtime state, this is also a common case for modern systems. Having these two assumptions, we can specialize the three tasks to "reading and writing XMI files", "invoking management interface for retrieving and updating system state" and "calculating how to main the causal connection". We divide the generated infrastructure into three independent parts for these three tasks, as shown in the middle part of Figure 3. The "Model-to-runtime synchronization engine" automatically interacts with the other two parts through an accordant interface which comply with the MOF reflection standard.

Our framework supports developers to construct these three parts in a model-driven way.

First, developers can define the meta-model of the architecture models using a standard meta-modeling language named MOF. This meta-model specifies what kinds of elements will appear in the architectural model, the attributes of these elements, and the possible association between them, and it is actually the specification of architecture style. From this meta-model, our framework will automatically generate Java code for reading and writing XMI files storing the architectural model.

Second, developers can use MOF to define the meta-model for system states, specifying the kinds of manageable elements, the attributes of these elements and the association between these elements, and use a fixed format of annotations to specify the system-specific ways for accessing these system states. From this meta-model, our framework will generate the system adapter for accessing the system's management interface.

Finally, developers can use a model transformation language, QVT, to specify the consistency rule he required between the architectural model and system state, and input the two meta-models and the consistency rule into a pre-implemented common synchronization engine. This engine can automatically retrieve the architectural model and the system state through the two adapters, calculating how to maintain the causal connection according to the consistency rules, and finally write the calculated changes back to the architecture and the system.

Notice that the steps in this constructing process are on the model level, and thus this approach has the common advantages of model-driven engineering, like productive, stable and maintainable. And moreover, we can also prove that the infrastructures constructed under our framework will satisfy the 4 properties of causal connection. This will not only deliberate developers from caring about the correctness of their infrastructures, but also deliberate them from convincing the users of their infrastructures.

3 Manipulating Architectural Models

Using our framework, developers just need to write a MOF meta-model to define the architecture style, and the framework can generate the proper architecture adapter from this meta-model. Through this generated adapter, the synchro-

nization engine can automatically manipulate the architecture models stored as XMI files, according to the defined architecture style.

It is natural to define an architecture style as a MOF meta-model. An architecture style defines what kinds of elements may exists in an architectural model (usually, but not necessarily, some kinds of components and connectors), the properties of each kind of elements, and the probable relationship between model elements. Users can use MOF classes, attributes and associations to define the the element types, properties and relationships, respectively. An example can be found in Figure 6, where we use a MOF meta-model to define an architecture style imitating C2.

Currently, we do not restrict the ways for defining "constraints" in an architecture style, and users can choose some mature model-driven techniques like OCL. Note that to support architecture-based runtime management, researchers have tried to add dynamic features, like update operations, into architecture styles [19]. In our approach, developers do not need to specially define such dynamic features, and we support the standard model modification like changing attribute, adding elements, etc. by default.

In our framework, we reuse a MOF implementation named Eclipse Modeling Framework (EMF) [7] to automatically construct an architecture adapter from the meta-model *. From the meta-model, EMF can generate a set of Java classese, and these classes implement the standard MOF reflection interface. With the help of an XMI parser implemented by EMF, models stored as XMI files can be reified as a set of Java objects under those generated classes. The generated classes and the reused XMI parser forms an architecture adapter. The synchronization engine can automatically manipulate the models represented as Java objects through the standard interface, and the XMI parser ensure that these modifications have effects on the XMI files.

4 Manipulating System States

To construct system adapters under our framework, developers also just need to define the *management states* of the target systems using MOF meta-models. Constructing system adapters is more complex than constructing architecture adapters. Unlike architectural models that are all stored as a unified and standard XMI files, system states hide behind various management interfaces, and each of these interfaces allows a specific method to access them. So the problems are how to provide a simple way for developers to describe such system specific information, and how to automatically generate effect adapters for different management interfaces from such descriptions. In this section, we first give an explanation about management states, and then discuss how to define management states using MOF meta-models. During the defining process, we allow developers to add a series of annotations with system-specific access methods, and this is the solution to the first problem. Finally, we introduce our generation tool for addressing the second problem.

Management states are system states which can be observed or manipulated

* Strictly speaking, EMF is not an implementation of MOF, because its meta-modeling language (named Ecore) is not strictly comply with MOF standard. But many model-driven technologies on MOF have corresponding implementations on Ecore, and thus, in this paper, we ignore the difference, and simply regard Ecore meta models as MOF meta-models

Table 1: Structural definition of management state

Aspects	Meta-model Elements
management states	Package
managed elements	Class
life cycle states	-(instance life cycle)
local state	Attribute
functional dependency	-
connection	Association
containment	Association(containment=true)

by management agents through some kind of management interface. Literature [21] gave a definition of *management states* by listing some common aspects. In the authors' opinion, a system's management state is composed of some *managed elements* which can be retrieved through the management interface provided by the system. Each of the managed elements contains some *local properties* and *life cycle states*, and may have some *functional dependencies* or *common connections* with other managed elements, and a managed element may also *contain* other managed elements. Take a J2EE system as an example. The JMX implementation provided by the J2EE server is the management interface for manipulating the management state. From this JMX interface, we can manage different kinds of MBeans representing the EJBs, data sources or middleware services currently running in the server. These MBeans are the *managed elements*. An MBean for a data source may contain some *local properties* like the maximal pool size of the database, and it may have *connection* with an EJB MBean because the EJB has to retrieve data from this data source.

To define the management states, developers must provide two kinds of information. "Structural definition" specifies the types of the aspects discussed above, using basic MOF meta-model elements. "Access method" specifies the system-specific way for manipulating these aspects. Developers provide this information by adding annotations into the defined meta-model elements. And in the annotations, they can write an extended version of Java code.

Table 1 summarizes a guidance for the structural definition. The first column lists the aspects for management states according to literature [21]. The second column list the meta-model elements to use for defining the left aspects. Note that we do not require explicit definition for "life cycle states", and such states can be reflected by the life cycle of model elements. We also ignored the "functional dependency", because in architecture-based management, such constraint specifications should better be handled at architectural level.

Table 2 lists the annotations a developer can add to specify the system-specific access method. For each kind of annotations, we list the annotation key name, the possible meta-model elements to annotate, and a short explanation about what information developers should provide through this kind of annotations.

Table 3 is a sample meta-model defining the managed elements for data sources in a J2EE system, corresponding to the example we used earlier in this section. Developers can first use a class to define data sources, and then use an Attribute named "jdbcMaxConnPool" to define a local property which can be managed. According to JMX standard, all the managed elements (implemented as MBeans) are identified by a special Java object with the type of ObjectName, and thus developers can annotate that the defined class's "rawtype" is javax.management.ObjectName. At last, they should annotate the class with

Table 2: Annotations for specifying access method

Key	Meta-model Elements	Explanation
mainEntry	Package	get an entry for further management
rawtype	Class	Java class for the managed elements
create	Class	create a new managed element
destroy	Class	destroy a managed elements
equal	Class	check if a model element is the reflection of a managed element
get	Attribute(upper bound=1) Association(upper bound=1)	retrieve a local property retrieve a connected managed element
set	Attribute(upper bound=1) Association(upper bound=1)	update a local property connect to a managed element
list	Attribute(upper bound>1) Association(upper bound>1)	retrieve a list of local properties list connected or contained elements
add	Attribute(upper bound>1) Association(upper bound>1)	add a value into a multi-valued property add a new connected or contained element
remove	Attribute(upper bound>1) Association(upper bound>1)	remove a value from a multi-valued property add a connected or contained element

Table 3: A sample system meta-model

```
Package(name="JOnASJMX"){
  Annotation(source="http://www.sei.pku.edu.cn/rsa"){
    "mainEntry"->{...}
  Class(name="JDBCDataSource"){
    Annotation(source="http://www.sei.pku.edu.cn/rsa"){
      "rawtype"->{javax.management.ObjectName}
      "create"->{...}
      "destroy"->{...}
    }
    Attribute(name="jdbcMaxConnPool", type=Interger, upperbound=1){
      Annotation(source="http://www.sei.pku.edu.cn/rsa"){
        "get"->{
            $featureName=((Interger)$mainEntry
              .getAttribute($core, "$feature")).intValue();
          }
        "set"->{...}
      }
    }
    ...
  }
}
```

the java code for creating a new data source runtime, and annotate the attribute with the Java code for retrieving the maximal size of this data source's connection pool. The text used inside an annotation is written by an extended version of Java. The identifiers starting with "$" are some variables used in generation time, and it will replace by a fix pattern of code during generation. Due to the restriction of paper length, we only present one annotation, and a detailed version of this meta-model can be found in Appendix 8.

In our framework, we implement a code generation engine to automatically generate the system adapter from such meta-models, based on the EMF code generation. Inheriting from EMF, the generated Java classes from our engine also implements the MOF standard reflection interface. Besides these standard methods, this tool also generates a series of specific methods according to the annotations, and these specific methods can actually manipulate the system states. Finally, the framework contains some auxiliary classes which connect the standard methods with the specific methods. We use a sample to show how the generated system adapters work.

Table 4 shows part of the code generated by our framework from the meta-model in Table 3. `eGet` is an example of the the *standard methods*, while

Table 4: A sample generated adapter

```
public class JOnASPackageImpl extends EPackageImpl...{
  ...
  public Management getMainEntry(){...}
}
public class JDBCDataSourceImpl
  extends CommonWrappingEObjectImpl ...{
  public int getJdbcMaxConnPool() {
    jdbcMaxConnPool=((Interger)JOnASPackage.eINSTANCE.getMainEntry()
              .getAttribute(getCore(), "jdbcMaxConnPool")).intValue();
    return jdbcMaxConnPool;
  }
  public Object eGet(int featureID,boolean resolve,boolean coreType){
    switch (featureID) {
      case JOnASPackage.JDBC_DATA_SOURCE__JDBC_MAX_CONN_POOL:
        return new Integer(getJdbcMaxConnPool());
    ...
  }
}
```

`getJdbcMaxConnPool` and `getMainEntry` are examples of the *specific methods*, which are generated according to the "mainEntry" and "get" annotations, respectively. When the synchronization engine wants to know the max pool size of an data source, it will automatically invoke the `eGet` with the proper featureID (it gets the meta information like class name and feature ID from the meta-model), and `eGet` will forward the invocation to `getJdbcMaxConnPool`, where the value will be retrieved from the J2EE server and returned. We choose a very simple example for the ease of comprehension, and other parts of the generated code are more complex. For example, if the synchronization engine wants to get all the data sources and call `eGet` automatically, the generated adapter will finally return an object of a specially implemented List class, which is one of the auxiliary classes we implemented in the framework. This object will call the corresponding `list` method generated from "list" annotation to get all the ObjectNames representing the available data sources in the current system, and instantiate a set of Java objects of the type `JDBCDataSourceImpl` to wrap these ObjectNames, so that the synchronization engine can go on to manipulate these data sources through their wrappers. The implementation of this generation tool is not the emphasis of this paper, and technical details will be discussed in a later literature.

5 Maintaining Causal Connections

Maintaining causal connections is the main function of runtime architecture infrastructures. In particular, maintaining causal connection is the activity of synchronizing the architectural model and the system states when one or both of them have changed, so that both of them reflect changes occurred on the opposite side. Since the architectural model and the system states are heterogeneous, the basis of this synchronization is not simply equivalence, but a particular relation between them expected by developers. Under our framework, developers only need to specify such relation on the model level, using a standard model transformation language, the QVT relational [8], and a general model-to-runtime synchronization engine can maintain the causal connection according to this relation.

Table 5: Sample relation between C2 and JOnAS

```
transformation C2JOnAS(arc:C2, sys:JOnAS){
 top relation Root2Root {
  host : String;
  enforce domain arc s:Architecture{deployedHost=host};
  enforce domain sys t:MBeanServer{serverHost=host};
  when{s.parent.oclIsUndefined();}
 }
 top relation Component2DataSource {
  name:String;
  maxPool:Integer;
  enforce domain arc arch:Architecture{};
  enforce domain arc conn:Connector{
   parent=arch,name='jdbc'};
  enforce domain arc comp:Component{
   below=conn,name=name,maxPool=maxPool};
  enforce domain sys server:MBeanServer{};
  enforce domain sys data:JDBCDataSource{
   name=name,parent=server,
   jdbcMaxConnectionPool=maxPool};
  when{Root2Root(arch,server);}
 }
}
```

Implementing this common synchronization engine is not trivial. First, when a change occurred at architectural model, we have to calculate what this change means for the system state according to the specified relation, and so is the change on system state. Second, when architecture and system change at the meantime, we have to merge these changes and calculate the correct subsequent modifications on architecture and system. Third, for system state, chances are that some modifications do not have expected effects, and for such situations we have to give a reasonable result for architecture-based management. We designed an algorithm based on QVT bidirectional transformation and model difference to address the above issues.

When developing under some framework, developers usually expect that the behavior of their product are predicable. On the basis of the properties of QVT bidirectional transformation and model difference (discussed in literature [22] and [1], respectively), we can demonstrate that our algorithm preserve a series of properties. From these properties, along with the unambiguous semantics of QVT transformation, developers can be clear about the behavior of their infrastructures constructed under our framework.

At the last of this section, we will report how we implement this algorithm.

5.1 Specifying relation between architectures and systems

We choose QVT relational language for developers to specify the relation between architecture and system. Unlike operational transformation languages like ATL and QVT operation, QVT relational is not for specifying how to create a new model from an original one, but specifying a relation between two sets of models complying with two meta-models respectively.

Table 5 is a sample QVT transformation specifying the relation between C2 architectural models and JOnAS system states. It is defined between the two meta-models we used as samples in the last two sections, and it is also part of the relation we defined to construct the sample infrastructure we have introduced in Section 2.

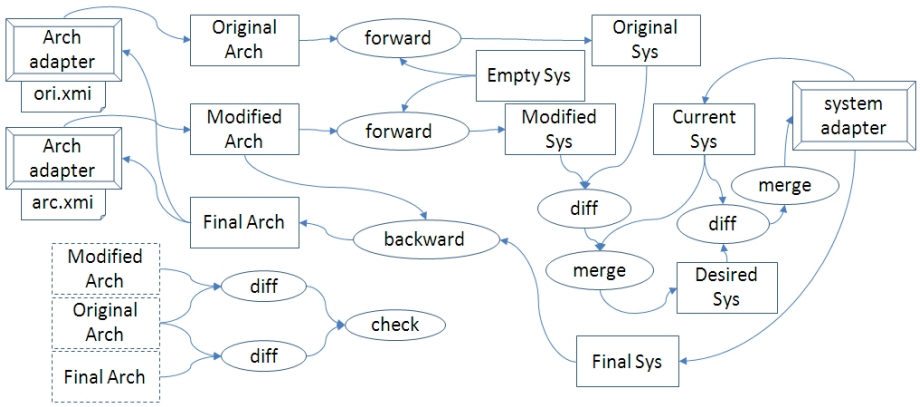

Figure 4: Synchronization algorithm

Here `Root2Root` can be interpreted as follows: for each root `Architecture` (not an inner structure of some component) in the architectural model there must a `MBeanServer` in the system, and vice versa. And the `Architecture`'s `deployedHost` must equal to the `MBeanServer`'s `deployedHost`. The second relation `Component2DataSource` means that if there is a `Component` linked with a `Connector` with type "jdbc", then there must be `DataSource` with the same name. Likewise, for each `DataSource` in the system, there must be a component with the same name, and linked with a `Connector` typed "jdbc".

We do not restrict the use of QVT language, that means any legal QVT relational transformation are acceptable by our framework.

5.2 Synchronization algorithm

An overview of our synchronization algorithm is shown in Figure 4. This algorithm processes two XMI files and a running system through the architecture adapter and system adapter. Before the execution of this algorithm, the two XMI files storing the architectural models before and after the management agent's modifying, and after the execution, the system state is changed according the management agent's modification, and the two XMI files reflect the current system state.

The basic idea of our algorithm is: get the system-side meaning of architectural modifications by transforming the two architectural models into two system models respectively 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.

Before discussing the process, we first introduce the techniques we employed in this algorithm.

- **Architecture adapter and system adapter** are constructed before synchronization, as discussed in the previous two sections. The arrow from an adapter to a model means loading a model through the corresponding adapter. We achieve this by invoking the standard "get" method recursively to retrieve the complete information from XMI files or state state,

and then copy the constructed model. The arrows pointing to architecture adapter means manipulating system states, for which we will discuss later. Except for invoking adapters, the operations on the intermediate models (represented as common rectangles) do not affect XMI files or system states.

- **Forward and backward transformation** are bidirectional transformations on the basis of the relation defined by QVT relational. 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 propagate chages in the opposite direction" [22].

- **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 merger** [1] is the operation for modifying a model according to a difference, and return the modified model. The arrow from a "diff" activity to the system adapter means merging the modifications in the difference result on the system state.

In the rest of this subsection, we use a small example to illustrate our algorithm step by step. This example is about about maintain causal connection between C2 architecture and JOnAS system. The meta-models, adapters and the relation required for this example has already been introduced as examples in previous sections. We suppose that before synchronization, `ori.xmi` only restored an empty architecture, while `arc.xmi` contains a `Component` and a Connector, which are inserted by the management agent. At this time, the JOnAS server only contain a data source.

Step 1. Loading architectural models. According to the example description, we get two models as follows. During the following discussion, we use a simple notation to express models. We express a model element beginning with its type at a non-indent line, and in the following lines, each with one indent, listing the features and their values. For example, the `Modified Arc` shown below contains tree model elements. The second element is a `Component`. Its name is `"MySQL"`, its parent is the #127.0.0.1 and its maximal pool size should be 1000.

```
/*Original Arc*/
Architecture
  deployedHost="127.0.0.1"

/*Modified Arc*/
Architecture
  deployedHost="127.0.0.1",
  component=#MySQL,connector=#jdbc
Component
  name="MySQL",parent=#127.0.0.1,
  below=#jdbc,  maxPool=1000
Connector
  name="jdbc",parent=0,above=#MySQL
```

Step 2. Forward transformation. We 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.

```
/*Original Sys*/
MBeanServer
  serverHost="127.0.0.1"

/*Modified Sys*/
MBeanServer
  deployedHost="127.0.0.1",
  jdbcDataSource=#MySQL
JDBCDataSource
  name="MySQL",parent=#127.0.0.1,
  jdbcMaxConnectionPool=1000
```

Step 3. Retrieve System State. We invoke a model copy method from system adapter to a 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. Currently, we assume that the JOnAS server only contain a data source named "HSQL".

```
/*Current Sys*/
MBeanServer
  deployedHost="127.0.0.1",
  jdbcDataSource=#HSQL
JDBCDataSource
  name="HSQL",parent=#127.0.0.1,
  ...
```

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.

```
/*Desired Sys*/
MBeanServer
  deployedHost="127.0.0.1",
  jdbcDataSource=#HSQL,
  jdbcDataSource=#MySQL
JDBCDataSource
  name="HSQL",parent=#127.0.0.1,
  ...
JDBCDataSource
  name="MySQL",parent=#127.0.0.1,
  jdbcMaxConnectionPool=1000
```

Step 5. Try to reconfigure the system. Differ `Desired Sys` and `Current Sys`, and the resulted difference contains the modifications we need if we want to change the system state according to `Desired Sys`. The difference is showned below, in the format proposed by literature [1].

```
/*Desired_Sys - Current_Sys*/
[ [ new(JDBCDataSource,#MySQL) ]
  [ insert(#127.0.0.1, jdbcDataSource, #MySQL),
    set(#MySQL, parent, #127.0.0.1)
    set(#MySQL, jdbcMaxConnectionPool, 1000)
  ]
]
```

We merge this difference into the actual system state through the system adapter. This merge process is the same as we used in the last step, i.e. translate the modifications in the difference into invocations to the standard MOF reflection interface. As discussed in Section 4, the system adapter will finally reconfigure the system according to these invocations. We name this step as "*try* to reconfigure the system", because such reconfigurations do not always lead to expected effect. For example, the difference contains `set(#MySQL, jdbcMaxConnectionPool,`

Table 6: Decision table for rechecking results

#	Type	Query	i	a	Condition	Explanation
1	new	type, id	1	0	-	failed to create this element
2	del	type, id	1	0	-	failed to destroy this element
3	set	id, feature	1	0	-	failed to configure this feature
4			1	1	i.value!= a.value	configured with another value
5	insert	id, feature	1	0	-	failed to insert into this feature
6	remove	id, feature	1	0	-	failed to remove from this feature

`1000)`, but this 1000 exceeds the server's ability, then the final maximal pool size will not be that high. After the reconfiguration, we get the final system state as follows.

```
/*Final Sys*/
MBeanServer
  deployedHost="127.0.0.1",
  jdbcDataSource=#2,
  jdbcDataSource=#3
JDBCDataSource
  name="HSQL",parent=#1,
  ...
JDBCDataSource
  name="MySQL",parent=#1,
  jdbcMaxConnectionPool=100
```

Step 6. Get final architectural model. We execute backward transformation between `Modified Arch` and `Final Sys`, to get the architectural model as follows. The `Component` named "MySQL" is also inside the top architecture, but its `maxPool` is 100 not 1000. A new `Component` appears to reflect the original data source in the system. Note that this new component is not a child of the root architecture, because the relation specified in Table tab:samplerelation does not contain enough information to determine the parent of this component. This behavior is reasonable, because as a hierarchical structure, this component may be inside the root architecture or the some of the inner structures, depending on the designer. Finally, we store this `Final Arch` into two XMI files, preparing for the next synchronization.

```
Architecture
  deployedHost="127.0.0.1",
  component=#MySQL,connector=#jdbc
Component
  name="MySQL",parent=#127.0.0.1,below=#jdbc_mysql,
  maxPool=100
Connector
  name="jdbc",  parent=0, above=#MySQL
Component
  name="MySQL",below=#jdbc, maxPool=100
```

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 `acutal` difference from `Original Arch` to `Final Arch`, and then check if the `intended` modifications all remains in the `actual` modification. The decision table used for this check is showed in Table 6.

This table means that for each `Type` of operations [1], we use the `Query` condition (e.g. same element type and id) to find a pair of operations from intended and actual difference respectively. Column `i` and `a` means we find a pair (1,1) or we cannot found a corresponding operation from actual difference (1,0). For a found paire, if the `Condition` is true, then we regard report

an exception. In our example, we have a pair `(set(#MySQL,maxPool,1000), set(#MySL,maxPool,100))` satisfying #4 situation, and this will be reported to the managed agent. Note that we do not consider (0,1) situations as exceptions, for example the actual difference contains a `new(Component,#HSQL)` while the intended one does not a corresponding operation. Although this situation is not the agent's intention, it does not *conflict* with the agent's intention.

5.3 About the properties

In this section, we discuss how this algorithm satisfy the four properties we listed in Section 2.

Property 1. Synchronization leading consistency. According to the "Correctness" property discussed in literature [22], 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.

Property 2. Non-interference reading. 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. For *normal conditions*, this algorithm can satisfy 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. But as we have discussed before, there exists some kinds of *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 modified architectural model and the current system state satisfy the relation, then `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 [22], when executing the backward transformation, the `Modified Arch` will not be changed.

5.4 Implementation and discussion

We use a QVT implementation named "mediniQVT" to execute the forward and backward transformation, and reuse our previous model difference and merge implementation developed in "Beanbag" project [23]. As we have tested, these two implementation satisfy the properties discussed in literatures [22] and [1], in most situations. The only exception we found so far is that medini QVT does not reflect the element removal when executing backward transformation. To resolve this problem, we add an additional step into the synchronization process to check what element has been removed in the system, and amend the final architectural model. As a implementation specific step, we do not discuss it in detail.

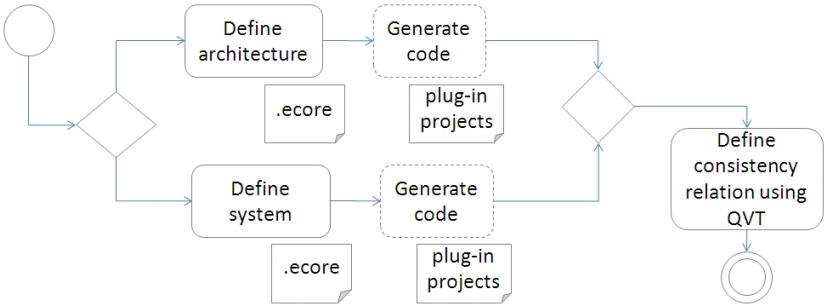

Figure 5: Process for constructing an infrastructure under our framework

Note that our algorithm depends on one assumption, i.e. during the synchronization process, the system state does not change. This assumption is reasonable if the synchronization can be executed quickly enough, and in our case study, this situation is satisfied. If in the actual usage, the system state change too frequently, management agents have to lock the system state before execute the synchronization.

6 Case Study

In this section, we report our case study for constructing an infrastructure to support runtime management of JOnAS systems based on C2-styled architecture model. We choose JOnAS as a target platform because JOnAS is a widely used open source J2EE application server, achieving the scale of systems in actual use. Moreover, we can use a well know application, JPS, to evaluate the effect of the generated infrastructure, and use the build-in JOnAS admin tool as a reference. We choose C2 styled architectural model in this case study because C2 is a well-researched architecture style and its advantages on managing GUI systems are widely accepted.

We have briefly revealed the effect this constructed infrastructure in Section 2.1, and in this section, we will focus on how we construct this infrastructure based on our framework, demonstrating that our framework can improve this constructing process to be productive.

6.1 Constructing the runtime architecture infrastructure

Our framework is implemented as several Eclipse plug-ins, depending on a number of reusable plug-ins or libraries like EMF, medini QVT, Beanbag, etc. The whole constructing process is performed under Eclipse environment, as shown in Figure 5.

First, we use a graphical Ecore editor to construct an Ecore meta-model. The architecture meta-model is shown in the left part of Figure 6. According to C2 style, this meta-model currently includes three classes, i.e. `Component`, `Connector` and `Architecture`. `Components` have some attributes, may link to connectors through "above" or "below" references, and may contain some inner architecture. From this meta-model, we generate three plug-in projects under the help of EMF.

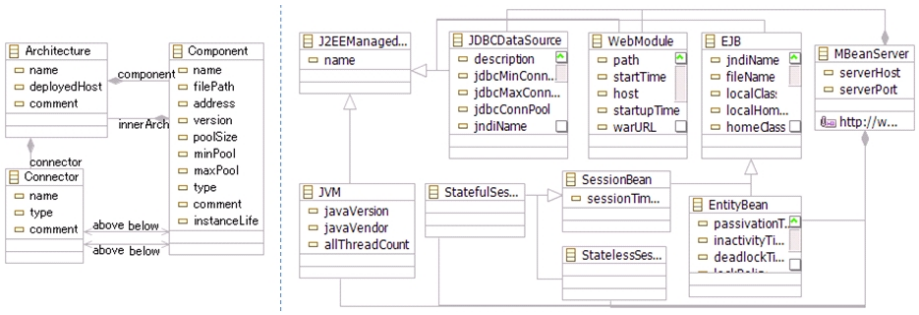

Figure 6: Meta-models for architecture and system

Table 7: Manual, generated and framework artifacts

	item	artifact	quantified workload	
Manual	define C2	Ecore model	29	model elements
	define JOnAS	Ecore model	61	model elements
	add annotation	Java code	474	LOC
	define relation	QVT text	207	LOC
Fra.Gen.	for arc	Java code	8761	LOC
	for sys	Java code	18263	LOC
	common meta-model	Ecore model	17	model elements
	Synch. and utility	Java code	2130	LOC*
*exclude reused libraries and plug-ins				

Second, we construct another Ecore meta-model to specify the kind of system states as show in the right part of Figure 6. This meta-model contains classes like `EntityBean`, `WebModule`, `DataSource`, etc. according to the types of MBeans provided by JOnAS JMX with same names. We add some attributes into these classes according to the attributes of corresponding MBean types. The inheritance structure between these classes agrees with the class hierarchy defined between the MBean classes. `MBeanServer` is a special class defining the JOnAS server itself. We also add annotations to these meta-model elements with the JMX specific code for accessing JOnAS server. An example of these annotations can be found in Table 3. We also use EMF to generate three projects.

Third and last, we define a QVT transformation to specify the consistency relation we expect between architectural models and system states. In Table 5, we list 2 of the 6 relations we defined.

Table 6.1 lists the quantified manual workload for implementing this case, comparing with the size of generated code and framework. From the simple comparison we can at least conclude that that our framework does save some effort for developers. We have not found a manual-developed runtime architecture infrastructure with same capabilities as a control sample, but we believe that seven hundred lines of code is not a heavy load for a developer familiar with JOnAS JMX, and neither is the one hundred of model elements for a developer familiar with EMF and QVT.

6.2 Using the constructed infrastructure

To use the constructed infrastructure, we can export the generated projects into plug-in jar files, and copy them into the "plugins" directory of an Eclipse

Table 8: Synchronization time spent

No.	modifications on arch. model	sync. time spend
1	-	5137 ms
2	-	3012 ms
3	change 1 attribute value	2917 ms
4	change 10 attribute value	3105 ms
5	add 1 WebModule	3262 ms
5	remove 1 WebModule and add 2 EJB	3327 ms
6	relink between WebModule and EJB	3329 ms

platform. After launching this Eclipse, we can create a empty project, and copy the two meta-models and the QVT file into this project. Now we can import an architecture model or create an empty one, modify it with text editor or a tree-based model editor provided by eclipse, and synchronize it with a running JOnAS system by clicking a tool-bar button.

In this case study, to intuitively reveal the effect of this infrastructure, we also use Eclipse GMF to generate a graphical editor for the architectural model, as shown in Figure 1. To generate this graphical editor, we created 3 GMF required models with 34 model elements in total, without writing a single line of code. Notice that is work is actually outside the scope of our framework, since it is only in charge of maintaining the causal connection.

We have briefly shown the effect of using this infrastructure in Section 2, through some basic scenarios for runtime management of the JPS application.

We did an experiment on the infrastructure constructed in this case study, on a laptop computer with an Intel Pentium M 1.6GHz process, and 1.5G memory. The Eclipse 3.4 platform, which is the host of architecture editor and our infrastructure, and a JOnAS 4.9.7 application server are all running on this same computer. The architectural model contains 69 components and connectors, while the JOnAS server contains over 300 manageable elements (MBeans), and 47 elements among them, along with over 100 attributes will be reflected to the architecture model. The experiment result is listed in Table 8.

The number in the first column represents the order for executing "synchronization". The first synchronization took quite a long time, and we doubt it is because the JVM has to load many classes from the relevant plug-ins and libraries. The 6th synchronization took the second long time because in our current wrapping, to change a EJB reference, we have to reload the EJB or the Web Module.

6.3 Discussion on some other concerns

Performance As an initial attempt, we did not pay too much attention on the performance of constructed infrastructures, although performance is also an important concerns. The experiment above shows that although this infrastructure's performance is not good, but it is still acceptable. Several seconds of time spent are reasonable for manual management. We have tried to use the build-in JOnAS Admin web page to deploy a WebModule, and after clicking the "confirm" button, we also have to wait for more than a second until the whole page is refreshed.

Actually, there are many potential ways to improve the performance, like caching the retrieved system state, employing incremental transformation [12], etc. and these improvement may be important for the infrastructures to support

automated management agents, like self-adaptation engine. We will emphasize on the performance in our further work.

Maintainability and incremental development We are not very familiar with JOnAS JMX interface, and thus we made some mistakes when filling the hook method. So, the first version of the constructed infrastructure also have some defects, like receiving exception when reading some attribute. Since the specific code are directly mapped to elements, attributes or references, we can easily locate the mistakes from a few lines of code in annotations. We think this is a reflection of the maintainability of infrastructures constructed under our framework. We also tried to reflect extra kinds of manageable elements into architectural models, and the experiment confirm that we just need to *supplement* the meta-models, QVT transformation, without breaking the existing effort. So we can also say that our framework is good for incremental development.

7 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 [5], 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 [17], for automatically executing or evaluating the reconfiguration [18], or even for self adaptation [11]. 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 [11] 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 [9], M. Antkiewicz and K. Czarnecki's model-code synchronization [2], and H. Giese and R. Wagnerand's incremental model synchronization [12]. Our approach attempt to apply bidirectional transformation technologies to a new area.

8 Conclusion

In this paper, we report our initial attempt towards a model-driven framework for constructing runtime architecture infrastructures. Under this framework, developers only need to write two MOF meta-models and a QVT transformation between them. The tools we implement will automatically generate code from the two meta-models for accessing architectural models and system states, and a common synchronization engine will maintain the causal connection according to the QVT transformation.

We do not wish to provide complete support for runtime architectures, but only put emphasis on maintaining causal connections. Developers still have to design the proper architecture style for their usage, and instrument their target systems with management capability at runtim. As there are already many research approaches on architecture styles and management capabilities, subsequent researchers can use our approach combining with these existing ones.

As an initial attempt towards flexible approach on runtime architecture, we current ignored some technical issues. One of these issues is about fault-tolerance. The mistake in the specification of causal connection or the inaccurate wrapping of management capability will all cause unexpected behavior of the infrastructure. We had a discussion about these exceptions in Section 5, but our current solution still remains at the step of "informing something is wrong" In the future, we'll try to find effective approaches to tolerating the faults in specifications, providing constructive information from the analysis of exceptions, or assisting developers in deriving correct specifications. Currently, we also paid little attention on the performance, which is a key issue for the scalability of the infrastructure.

A model-based extensible infrastructure also has some additional features. First, using MOF to define architecture styles will help reuse management policies between different approaches. Second, by explicitly specifying the causal connections under QVT, it is possible to inspect, analyze, or even verify the runtime architecture infrastructures before they are actually used. We plan to give further research on these concerns.

Acknowledgment

The research was supported in part by the Natural Science Foundation of China under Grant No. 60528006, the National Basic Research Program of China (973) under Grant No. 2005CB321805the High-Tech Research and Development Program of China under Grant No. 2007AA010301, the Science Fund for Creative Research Groups of China under Grant No. 60821003

References

[1] Marcus Alanen and Ivan Porres. Difference and union of models. In *The 6th Unified Modeling Language (UML)*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 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, LNCS,*, 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 (ICSE)*, 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*. Springer-Verlag, 1998.

[6] Jeremy S. Bradbury, James R. Cordy, Jürgen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *The 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS)*, pages 28–33, 2004.

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

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

[9] 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.

[10] 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, Washington, DC, USA, 2007. IEEE Computer Society.

[11] 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.

[12] 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.

[13] 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.

[14] Java Management Extensions, `http://www.jcp.org/en/jsr/detail?id=77`.

[15] Java PetStore, `http://java.sun.com/developer/releases/petstore/`.

[16] JOnAS Project. Java Open Application Server, `http://jonas.objectweb.org`.

[17] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[18] 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, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 899–910, New York, NY, USA, 2008. ACM.

[20] 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.

[21] Sylvain Sicard, Fabienne Boyer, and Noel De Palma. Using components for architecture-based management: the self-repair case. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 101–110, New York, NY, USA, 2008. ACM.

[22] Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, volume 4735 of *Lecture Notes in Computer Science*, pages 1–15, 2007.

[23] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, Hui Song, and Hong Mei. Beanbag: Operation-based synchronization with intra-relations. In *GRACE Technical Report GRACE-TR-2008-04*, 2008.

[24] 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.

Appendix A.

Sample meta-model for defining data sources in JOnAS

```
Package(name="JOnASJMX"){
  Annotation(source="http://www.sei.pku.edu.cn/rsa"){
    "mainEntry"->{
      ...
      mgmtHome = (ManagementHome)
        PortableRemoteObject.narrow(
          initialContext.lookup("ejb/mgmt/MEJB"),
          ManagementHome.class);
      return mgmtHome.create();
    }
  Class(name="JDBCDataSource"){
    Annotation(source="http://www.sei.pku.edu.cn/rsa"){
      "rawtype"->{javax.management.ObjectName}
      "create"->{
          ...
          $mainEntry.invoke(dbServerce,
              "loadDataSource",
              new String[]{getName(),pro,new Boolean(true)},
              new String[]{"java.lang.String",
                "java.util.Properties",
                "java.lang.Boolean"}
              )
        }
      "destroy"->{...}
    }
    Attribute(name="jdbcMaxConnPool", type=Interger, upperbound=1){
      Annotation(source="http://www.sei.pku.edu.cn/rsa"){
        "get"->{
            if($core==null) return "";
            $featureName=$mainEntry
              .getAttribute($core, $feature);
          }
        "set"->{...}
      }
    }
    ...
  }
  Class(name="MBeanServer"){
    Association(name="jdbcDataSource", containment=true){
      Annotation(source="http://www.sei.pku.edu.cn/rsa"){
        list->{
          ObjectName query=new ObjectName("jonas:j2eeType=$featurName");
          Set sets=$mainEntry.queryNames(query, null);
          $return.addAll(sets);
        }
        add->{...}
  ...
}
```

Code generated from the above meta-model

```
public class JOnASPackageImpl extends EPackageImpl...{
  ...
  public Management getMainEntry(){
    ...
  }
}

public class MBeanServerImpl extends WrappingEObjectImpl...{
  public List listSubCores(int featureID){
    switch(featureID){
    case JOnASPackage.MBEAN_SERVER__JDBC_DATA_SOURCE:
      ...
      pack.getMainEntry.invoke(dbServerce,
          "loadDataSource",
          new String[]{getName(),pro,new Boolean(true)},
```

```
          new String[]{"java.lang.String",
            "java.util.Properties",
            "java.lang.Boolean"}
          )
      break;
    ...
  }
  public EList<JDBCDataSource> getJdbcDataSource() {
    if (jdbcDataSource == null) {
      jdbcDataSource = new
        EObjectContainmentEListForWrapping<JDBCDataSource>(
          JDBCDataSource.class,
          this,
          JOnASPackage.MBEAN_SERVER__JDBC_DATA_SOURCE,
          JOnASPackage.eINSTANCE.getJDBCDataSource()
        );
    }
    ((EObjectContainmentEListForWrapping<JDBCDataSource>)
      jdbcDataSource).refreshWrap();
    return jdbcDataSource;
  }
  public Object eGet(int featureID,boolean resolve,boolean coreType){
    switch (featureID) {
    case JOnASPackage.MBEAN_SERVER__JDBC_DATA_SOURCE:
        return getJdbcDataSource();
    ...
  }
  ...
}

public class JDBCDataSourceImpl ...{
```

The complete QVT specifying relation between C2 and JOnAS

```
transformation C2JOnAS(arc:C2, sys:JOnAS){
 key C2::Connector{name};
 key C2::Component{name};
 key C2::Architecture{deployedHost};
 key JOnAS::MBeanServer{serverHost};
 key JOnAS::EJB{name};
 key JOnAS::WebModule{name};

 top relation Root2Root {
  host : String;
  enforce domain arc s: C2::Architecture {
   deployedHost=host
  };
  enforce domain sys t: JOnAS::MBeanServer {
   serverHost=host
  };
  when {
   s.parent.oclIsUndefined();
  }
 }

 top relation Component2EntityBean {
  name : String;
  address : String;
  filePath : String;
  jdbc:String;
  checkonly domain arc arch:C2::Architecture{};
  enforce domain arc conn:C2::Connector{
   name=address
  };
  enforce domain arc comp : C2::Component{
   name=name,
   address=address,
   filePath=filePath,
   type='Entity',
   below=conn,
   above = conn2 : C2::Connector{
```

```
       name=jdbc
      }
 };

 checkonly domain sys mbeanserver :JOnAS::MBeanServer{
 };
 enforce domain sys entitybean : JOnAS::EntityBean{
  name=name,
  jndiName=address,
  fileName=filePath,
  parent=mbeanserver,
  dataSourceJNDI=jdbc
 };
 when {
  Root2Root(arch,mbeanserver);
 }
}

top relation Component2StatefulSessionBean {
 name : String;
 address : String;
 filePath : String;
 instanceLife : Integer;
 checkonly domain arc arch:C2::Architecture{};
 enforce domain arc conn:C2::Connector{
  name=address
 };
 enforce domain arc comp : C2::Component{
  name=name,
  address=address,
  filePath=filePath,
  type='DurativeOperation',
  below=conn,
  instanceLife=instanceLife
 };
 checkonly domain sys server :JOnAS::MBeanServer{
 };
 enforce domain sys sb : JOnAS::StatefulSessionBean{
  name=name,
  jndiName=address,
  fileName=filePath,
  parent=server,
  sessionTimeOut=instanceLife
 };
 when {
  Root2Root(arch,server);
 }
}

top relation Component2StatelessSessionBean {
 name : String;
 address : String;
 filePath : String;
 instanceLife : Integer;
 jdbc : String;
 checkonly domain arc arch:C2::Architecture{};
 enforce domain arc conn:C2::Connector{
  name=address
 };
 enforce domain arc comp : C2::Component{
  name=name,
  address=address,
  filePath=filePath,
  type='OneStopOperation',
  instanceLife=instanceLife,
  below=conn,
  above = conn2 : C2::Connector{
       name=jdbc
      }
 };

 checkonly domain sys server :JOnAS::MBeanServer{
 };
 enforce domain sys sb : JOnAS::StatelessSessionBean{
```

```
  name=name,
  jndiName=address,
  fileName=filePath,
  parent=server,
  sessionTimeOut=instanceLife,
  dataSourceJNDI=jdbc
 };

 when {
  Root2Root(arch,server);
 }
}

top relation Component2WebModule {
 name : String;
 address : String;
 filePath : String;
 refjndi : String;
 checkonly domain arc arch : C2::Architecture{};
 enforce domain arc conn : C2::Connector{
  name='HTTP',
  parent=arch
 };
 enforce domain arc comp : C2::Component{
  name=name,
  address=address,
  filePath=filePath,
  below=conn,
  above=refconn : C2::Connector{
     name=refjndi
    }
 };
 checkonly domain sys server : JOnAS::MBeanServer{};
 enforce domain sys webModule : JOnAS::WebModule{
  name=name,
  path=address,
  warURL=filePath,
  parent=server,
  ejbref=refjndi
 };
 when {
  Root2Root(arch,server);
 }
}


top relation Component2DataSource {

 name:String;
 address:String;

 checkonly domain arc arch : C2::Architecture{};
 enforce domain arc conn : C2::Connector{
  name=address,
  parent=arch,
  type='jdbc'
 };
 enforce domain arc comp : C2::Component{
  below=conn,
  name=name,
  address=address,
  parent=arch
 };
 checkonly domain sys server : JOnAS::MBeanServer{};
 enforce domain sys data : JOnAS::JDBCDataSource{
  jndiName=address,
  name=name,
  parent=server
 };
 when{
  Root2Root(arch,server);
 }
}
```

```
}
```