Inferring Meta-Models for Runtime System Data from the Clients of Management APIs

Hui Song[1], Gang Huang[1⋆], Yingfei Xiong[2], Franck Chauvel[1], Yanchun Sun[1], and Hong Mei[1]

[1] Key Lab of High Confidence Software Technologies (Ministry of Education)
School of Electronic Engineering & Computer Science, Peking University, China
{songhui06,huanggang,franck.chauvel,sunyc,meih}@sei.pku.edu.cn
[2] Generative Software Development Lab, University of Waterloo, Canada
yingfei@gsd.uwaterloo.ca

Abstract. A new trend in runtime system monitoring is to utilize MOF-based techniques in analyzing the runtime system data. Approaches and tools have been proposed to automatically reflect the system data as MOF compliant models, but they all require users to manually build the meta-models that define the types and relations of the system data. To do this, users have to understand the different management APIs provided by different systems, and find out what kinds of data can be obtained from them. In this paper, we present an automated approach to inferring such meta-models by analyzing client code that accesses management APIs. A set of experiments show that the approach is useful for realizing runtime models and applicable to a wide range of systems, and the inferred meta-models are close to the reference ones.

1 Introduction

Monitoring is becoming more and more important for running software systems. The core of monitoring is to retrieve and process the *runtime system data* (such as its memory occupation, instantiated components, user load, etc.), which describe the state, structure and environment of the system. Currently, for most systems, runtime monitoring is still a code-level task. Developers have to write code upon the system's management API, like the JMX API of JEE systems [1].

Runtime model is a promising approach to raising the abstraction level of runtime monitoring [2]. By representing the intangible runtime system data as explicit and well-formed models, system monitoring could be performed in a model-based way, with full semantic basis, utilizing the plenty of model-based techniques such as OCL, QVT, GMF, etc. Runtime model is an extension of the traditional model-driven methodology at runtime.

Since different systems provide totally different runtime data, the first task for supporting runtime model on a system is to construct a proper meta-model, which defines the types of the system data that can be retrieved from the system's

⋆ corresponding author

management API, as well as the association between these data types. To our knowledge, the current generic runtime model supporting tools [3–6] all require users to define such meta-models *by hand*.

However, it is not easy to define such a meta-model. First, many management APIs conform to standard interfaces, like JMX, OSGi, DOM. Such APIs do not provide specific classes for different types of system data, and thus we cannot get the data types from the static type systems of the APIs. Second, many systems do not provide clear documents about what kinds of data they provide, and even if there are such documents, it is still tedious to convert the informal specifications into meta-models. On the other hand, developers may understand the system data from existing API clients (programs using the APIs). Although the clients do not usually contain the direct definition of data types, but they carry the experts' understanding about what data can be obtained and how to use them. The problem is that the client code is often too complicated for human users, especially when it is full of branches and inter-method logics.

In this paper, we present an automated approach to inferring the meta-model of runtime system data under management APIs, by means of static analysis of the API client code. The process is automated, without requiring users to annotate the source code. Our contributions can be summarized as follows.

- We clarify the relation between the API client code and the data types manipulated by it, and provide a novel static code analysis approach to extract such data types automatically.
- We construct an automated tool for meta-modeling a system's runtime data. The output meta-model guides developers in using the API, and is also an input of our previous tool-set to realize the runtime model.
- A set of experiments reveal the usability, wide scope, and effectiveness of our approach, and also reveal that static analysis on API sample clients is sufficient to meta-model the data under management APIs

Our tool and experiments are stored at `http://code.google.com/p/smatrt`.

The rest of the paper is organized as follows. Section 2 gives an overview of the approach. Section 3 to Section 5 explain the approach in three steps. Section 6 introduces our implementation briefly and evaluates the approach based on a set of experiments. Finally, Section 7 introduces related research approaches and Section 8 concludes the paper.

2 Approach Overview

2.1 Motivating example

We take SUN's reference JEE server, Glassfish [7], as an example. Figure 1 shows how to monitor the memory status through its JMX interface. From the two parameters `url` and `account`, the `printMem` method first constructs a JMX connector, and then obtains a management entry `mbsc`. After that, it queries out an `ObjectName` of the MBean (Management Bean, the basic management unit

```
1   public class JMXClient {
2     public void printMem(JMXServiceURL url, Map account) throws Exception {
3       JMXConnector connector = JMXConnectorFactory.connect(url, account);
4       MBeanServerConnection mbsc = connector.getMBeanServerConnection();
5       ObjectName memory = ObjectName.getInstance("*type=Memory");
6       memory = mbsc.queryNames(memory,null).iterator().next();
7       CompositeDataSupport heapUsage=(CompositeDataSupport)
8         mbsc.getAttribute(memory,"HeapMemoryUsage");
9       Boolean verbose=(Boolean)mbsc.getAttribute(memory, "Verbose");
10      String attr="";
11      if(verbose.booleanValue()) attr="max";
12        else attr="used";
13      System.out.println(String.format("%d,␣after␣%d␣GCs",
14        heapUsage.get(attr), countGC(mbsc)));
15    }
16    public long countGC(MBeanServerConnection mbsc) throws Exception{
17      Set<ObjectName> gcs=mbsc.queryNames(
18        ObjectName.getInstance("java.lang:type=GarbageCollector,*"),null);
19      long total=0;
20      for(ObjectName gc: gcs)
21        total+=((Long)mbsc.getAttribute(gc, "CollectionCount")).longValue();
22      return total;
23    }
24  }
```

Fig. 1. Sample JMX client to monitor the memory status of Glassfish Server

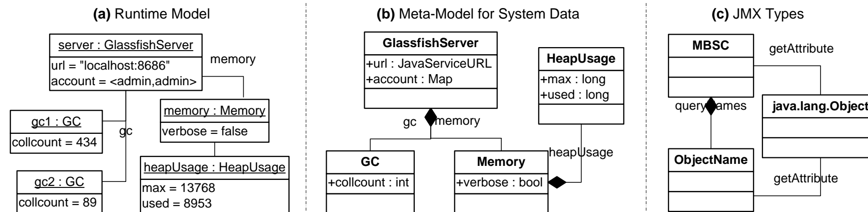

Fig. 2. The runtime model for the memory status of Glassfish

of JMX) for memory (Lines 6-8). Using this MBean, it obtains the heap usage information (7-8), and checks if the memory is verbose (9). If verbose, it prints the maximal heap size (11,14), otherwise, the used heap size (12, 14). Finally, it invokes the other method `countGC` (14), to get the MBean for garbage collection (17-18), and sums up the total number of collections.

Figure 2(a) shows a runtime model that intuitively represents the memory status. Figure 2(b) is the meta-model needed for such a runtime model. This meta-model contains two aspects of information: 1) *What kinds of system data can be obtained through the JMX API.* 2) *The association between the data types.*

The target of this paper is to automatically construct such a meta-model. We cannot simply transform the API static type system into a meta-model. In that way, we will just get a meta-model like Figure 2(c), because JMX provides a general `ObjectName` for all kinds of data. Therefore, we need the knowledge about how people use the general class. The API client carries this information.

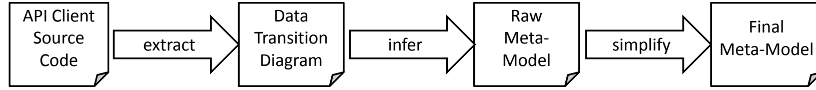

Fig. 3. Approach overview

2.2 Inferring meta-models from the API clients

We can infer meta-models of the system data from API clients because the clients were written by developers according to their understanding about the data types and associations. In other words, when the developer wrote this client, she had a meta-model like Figure 2(b) in her mind.

1. A class or a primitive data type defines a kind of system data, which can be obtained by accessing the API. For example, from the `Memory` class, developers knew there is an MBean for memory status (Lines 3-6).
2. A property of a class type means accessing the API with the help of this type of data could obtain another type of data. For example, from the `heapUsage` association and the `verbose` attribute of `Memory`, developers knew that from a memory MBean, they can get the heap usage information and the verbose flag, and thus wrote the code like Lines 7-9.
3. Different classes and properties indicate different ways to access the API. Here different ways not only mean invoking different methods, but also mean invoking the same method with different parameters.

Reasoning reversely from the above scenarios, to recover the meta-model from client code, we need to find out 1) all the different ways existing in the client for accessing the API, and 2) what other system data are used in each way of API access. The former indicates potential classes or primitive data types in the meta-model, while the latter indicates the properties connecting them.

To answer these two questions, we abstract the API access appeared in the client code as a *transition* from a set of input data to a single output data. For example, `heapUsage.get(attr)` in Line 14 is a transition from `heapUsage` and `attr` to an intermediate value. For each transition, we analyze how the client *feeds* it. Here "feeding a transition" means providing it the required input data. We can get two different slices in the client code to feed the above transition: One is to use the system data `heapUsage` obtained from Line 7 and a constant value `"max"` assigned in Line 11, and the other is to use the same `heapUsage` but a different constant value `"used"` from Line 12. So there may be two types of system data, and they are properties of `HeapUsage`.

We generate the meta-model by investigating all the API accesses, and analyzing how they are fed. The approach has three steps, as shown in Figure 3. We first abstract the source code as a *data transition diagram* (DTG). From the extracted DTG, we analyze all the different slices to feed each transition of API access, and generate classes and properties accordingly. Finally, we simplify the raw meta-model to remove the redundant classes.

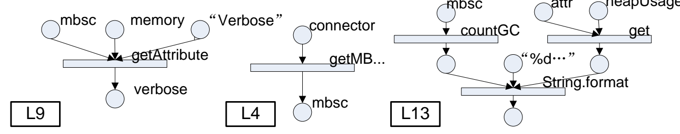

Fig. 4. Sample DTGs

3 Extracting data transition graphs

We model the source code as a data transition graph (DTG). The meaning of DTG is similar to the data-based program dependence graph [8], but with explicit record of variables and API accesses. Specifically, a DTG $D = (P, T)$ is constituted of *places* and *transitions*[3]. *Places* indicate where the data may appear. A place corresponds to an appearance of a variable or an implicit intermediate result inside an expression. A *Transitions* $T : \wp(P) \rightarrow P$, is a function from a set of input data to one output data. A transition corresponds to an assignment, an invocation, or a field access, etc. We name the transitions corresponding to API accesses as **API involved transitions**, notated as $\mathcal{A}$.

Figure 4(L9) shows a sample DTG extracted from Line 9 of Figure 1, and we notate the only transition in this DTG as follows.

$$\tau = \{p(mbsc), p(memory), p('Verbose')\} \xrightarrow{getAttribute} p(verbose)$$

We give each place or transition a signature to link it with the original source code. For the above τ, we use the following notations to retrieve the information it carries. 1) get input places: $\tau.i = \{p(mbsc), p(memory), p('Verbose')\}$, 2) get the output place: $\tau.o = p(verbose)$. We also popularize these notations onto DTGs, i.e., $D.i = \bigcup_{\tau \in D.T} \tau.i$.

We construct DTGs in three steps. We first ignore the control structures such as sequences, branches or loops, and construct a DTG for each of the individual expressions or assignments. Then we merge the disjointed DTGs within the bounds of method declarations, considering the control structures. Finally, we merge the DTGs constructed from different method declarations.

We define a function $\chi : Expr \rightarrow \mathcal{T}$ to construct DTGs from expressions and assignments, based on Sahavechaphan.[9]'s summary on Java expressions.

- **Variable access.** $\chi(var) = \phi \xrightarrow{var} p(var)$
- **Assignment.** $\chi(e_l = e_r) = \{\chi(e_r).o\} \xrightarrow{'='} \chi(e_l).o$
- **Static invocation.** $\chi(cls.fun(e_1, ..., e_n)) = \{\chi(e_1).o, ..., \chi(e_n).o\} \xrightarrow{cls.fun} p()$
- **Field access.** $\chi(e.fld) = \{\chi(e).o\} \xrightarrow{fld} p()$
- **Instance invocation.** $\chi(e.fun(e_1, ..., e_n)) = \{\chi(e).o, \chi(e_1).o, ..., \chi(e_n).o\} \xrightarrow{fun} p()$

[3] we borrowed the names and notations from petri-net, but the semantics are not the same

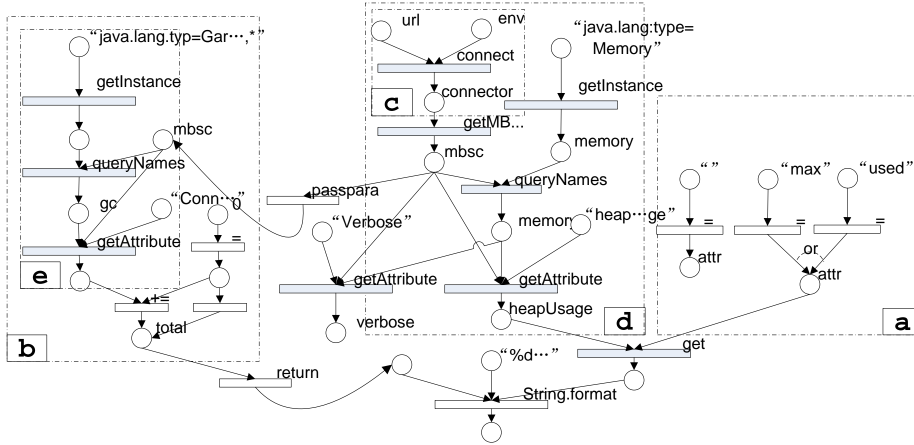

Fig. 5. The DTG for our running example

Figure 4(L13) illustrates the DTG we construct from Lines 13-14 of Figure 1. We first visit the *static invocation* of `String.format`, and construct a transition. After that, we recursively visit its parameters, constructing three other transitions, and appointing their outputs as the inputs of the first transition.

When merging DTGs, we combine an input place π_1 with an output place π_2, if they correspond to the same variable, and the statement of π_2 is an *effective assignment* for the statement of π_1. For example, we combine the output of Figure 4(L4) with the first input of (L9). Here Line 4 is an effective assignment for Line 9 because they belong to the same sequence (so L4 will be executed before L9, assigning value to `mbsc`), and there is no other assignments to `mbsc` between them (so the value will not be flushed). Similarly, Line 11 and Line 12 are two effective assignments to `attr` for Line 14, but Line 10 is not, because it is completely shielded by the above two assignments, and thus we combine them as shown in Figure 5(a). We determine effective assignments using Ramanathan et al.'s work on statement precedence, which is powerful enough to handle sequences, branches and loops [10].

We combine DTGs across methods, with the help of parameter passing: We first remove the transition for the method invocation. And then we connect the input places from the *actual parameters* of the invocation with the places from the *formal parameters* in the method declaration. Finally, we connect the output place of the original transition with the output place extracted from the returned expression of the method declaration. For example, Figure 5(b) is the DTG extracted and combined from method `countGC`, we substitute it with the original `countGC` (Figure 4(L13)), obtaining this DTG.

Finally, we obtain the DTG for the sample client as shown in Figure 5. The gray bars represent the API involved transitions.

4 Inferring the meta-model

We infer the meta-model by finding out all the potential ways in the client code to feed API accesses, and the system data used in each of these ways.

4.1 Analyzing how to feed the transitions

We analyze all the different ways to feed each API involved transition, by extracting the *valid sub-DTG*s ending up with the transition.

Definition 1. *For a DTG D and a transition $\tau \in D$, D_v is a τ-ended* ***valid sub-DTG*** *(VSD) of D (notated as $D_v \sqsubseteq_\tau D$), iff D_v is a sub graph of D, and it satisfies the following requisitions.*

1. $\tau \in D_v \wedge \tau.o \notin D_v.i$. That means τ is an end of D_v.
2. $\forall \pi \in D_v.o, \pi \notin D_v.i \Rightarrow \pi = \tau.o$ That means τ is the only end of D_v.
3. $\forall \pi \in D_v.i, \pi \in D.o \Rightarrow \pi \in D_v.o$. That means if an input place π is neither a global input nor a constant value (because there are transitions in D that output to π), then there must be a transition inside D_v that outputs to π.
4. $\forall \tau_1, \tau_2 \in D_v.\tau_1.o \neq \tau_2.o$. That means there cannot be different transitions outputting to the same place.

For the sample DTG in Figure 5 (we name it as D), the sub graph marked as `c` (named as D_c) is a VSD ending up with `connect` ($D_c \sqsubseteq_{connect} D$). Similarly, $D_d \sqsubseteq_{getAttribute} D$. However, $D_e \not\sqsubseteq_{getAttribute} D$ because `mbsc`'s source transition is not included. D_a cannot constitute a VSD ending up with `get`, because there are two transitions outputting to the same place `attr`.

A $D_v \sqsubseteq_\tau D$ represents a *sufficient* and *necessary* subgraph to feed τ. *Sufficiency* is ensured by Requisition 3: Any transition in D_v can be fed by transitions inside the subgraph, or by global inputs or constant values. *Necessity* means if any transition in D_v is removed, τ cannot be fed, and it is ensured by Requisition 2 and Requisition 4 together. As a result, *each $D_v \sqsubseteq_\tau D$ represents a different slice to feed τ.*

Definition 2. *For a $D_v \sqsubseteq_\tau D$, we say $\tau' \in D_v \cap \mathcal{A}$ is a* ***direct API involved feeder*** *of τ, if $\nexists \tilde{D_v} \sqsubseteq_{\tilde{\tau} \in D_v \cap \mathcal{A}} D_v, \tau' \in \tilde{D_v}$.*

For example, in Figure 5(d), `queryNames` is a direct API involved feeder of `getAttribute`. $\tau' \in D_v \cap \mathcal{A}$ means the output of τ' (a kind of system data) is useful to feed τ, and $\nexists \tilde{D_v} \sqsubseteq_{\tilde{\tau} \in D_v \cap \mathcal{A}} D_v, \tau' \in \tilde{D_v}$ means that τ' directly feeds τ, without using other system data as intermediary. In a word, *for a specific slice to feed τ, each of its direct feeder represents a system data which is directly used.*

4.2 The meta-model inferring algorithm

We infer a raw meta-model by analyzing all the VSDs for all the API involved transitions, and finding their direct feeders. In this step, the raw meta-model only contains classes and associations. We assume that every different way for an API access corresponds to a different class, and every type of system data directly used to obtain another type corresponds an association.

We construct all the VSDs ending up with any API involved transition, and generate a class for each of them. In Figure 5, for the `getAttribute` transition that outputs `heapUsage`, we can construct one VSD, marked as "d" (named D_d). And thus we generate the class named `Comp--Usage` as shown in Figure 6 for this VSD. Alternatively, for the transition `get`, we can construct two VSDs. Both of them contain D_d and the transition `get`, but one of them contains the assignment transition from `"max"`, while the other contains the one from `"used"`. Thus we generate two classes for this transition, named `Long-1` and `Long-2`.

For each VSD D_v constructed before, we find the direct feeders τ', construct VSDs $D'_v \sqsubseteq_{\tau'} D_v$, and generate an association to connect the class of each D'_v with the class of D_v. In the above example, for the first VSD constructed from `get`, `getAttribute` is a direct feeder. Thus we generate the association from `Comp--Usage` to `Long-1`.

We write our meta-model inferring algorithm as the pseudocode shown in Algorithm 1. The algorithm first constructs a root class (Line 1). Then it enumerates all the API involved transitions (Lines 2-23) to 1) construct all the VSDs ending up with this transition, and then 2) generate classes and associations.

The first step is to trace back from the current transition τ, and construct a set V of all VSDs ending up with τ. It initials V with only one DTG which has only one transition τ (Line 3). If there are still invalid DTGs in V, it takes a D_t that is still not valid (Line 5), finds a place π that make D_t invalid. If π is the output of an involved transition τ' (Line 8), it gets the VSDs ending up with τ', uses them to expand D_t, and puts the results back into V. If π is the output of n non-involved transitions, it grows D_t with each of these transitions, constructing n new subgraphs.

The second step is to create classes and associations. For each constructed VSD D_t stored in V, the algorithm creates a new class c_n(Line 15). The method `CName` calculates the class name by combining the names of the variable and its Java type, like `ObjectNameMemory`. After that, it enumerates all the direct feeders. For each feeder τ', it constructs a VSD $D'_v \sqsubseteq_{\tau'} D_v$, finds the class c_p generated from D'_v (Line 17), and creates an association to link c_p with c_n (Line 18). The name is calculated by `RName`, combining the transition signature and the constant inputs, like `getAttributeVerbose`.

Figure 6 shows the complete raw meta-model of our running example.

5 Simplifying the raw meta-model

We summarize five automated rules (antipatterns followed by refactoring actions) to simplify the raw meta-model, remove the non-significant or duplicated

Algorithm 1: Inferring the meta-model

Input: A DTG D extracted and combined from the sample code
Output: A set $Pack$ of classes, constituting the meta-model

```
1  c_r ← class : {name ↦ 'Root' a : input ↦ Type}, Pack ← {c_r}
2  for τ ∈ A ∩ D do
3      V ← {DTG(τ)}
4      while ∃D_t ∈ V, D_t ⋣_τ D do
5          D_t ← any({D_t ∈ V | D_t ⋣_τ D})
6          π ← any({π ∈ D.o | π ∉ D_t.o})
7          V ← V − {D_t}
8          if ∃τ' ∈ A_D, τ'.o = π then
9              foreach D'_t ⊑_τ' D do V ← V ∪ {D_t ∪ D_i}
10         else
11             foreach τ_i ∈ D ∧ τ_i.o = π do V ← V ∪ {D_t ∪ {τ_i}}
12         end
13     end
14     for D_t ∈ V do
15         c_n ← class:{name ↦ CName (τ.o) }
16         foreach τ' as a direct API involved feeder of D_t do
17             c_p ∈ Pack is generated from D'_v ⊑_τ' D_v
18             c_p ← c_p ∪ {r : RName(D_t, D_av) ↦ c_n}
19         end
20         if ∄τ' as a direct feeder then c_r ← c_r ∪ {r : RName(D_t) ↦ c_n}
21         Pack ← Pack ∪ c_n
22     end
23 end
24 return Pack
```

classes and associations, and weaken the classes into simple data types if they are never used to obtain other data.

To formalize these rules, we introduce a new notation named *reachability*. For two classes c and c' in the same meta-model, we say c' is reachable from c by r (noted as $c \xrightarrow{r} c'$), if there is an association r in c and its type is c'. We say c' is reachable from c, if there is a sequence of classes to transfer reachability from c to c', noted as $c \xrightarrow{*} c'$.

Rule 1. *Merge equivalent references.* $\dfrac{c \xrightarrow{r_1} c_1,\ c \xrightarrow{r_2} c_2,\ r_1 \overset{name}{=} r_2,\ c_1 \overset{name}{=} c_2}{c \leftarrow c - \{r_2 \mapsto c_2\}}$

According to our generation algorithm, if a class contains two associations with the same name, then that means the client invokes the same method with the same parameter. If the two target classes also have the same name, then that means the developer appoints the same variable name to the invocation results. So we determine the two associations as duplicated, and delete one of them.

Rule 2. *Remove the forwarding classes.* $\dfrac{c_1 \xrightarrow{r_1} c_2,\ c_2 \xrightarrow{r_2} c_3,\ |c_2.r|=1,\ c_2.a=\phi}{c_1 \leftarrow c_1 - \{r_1 \mapsto c_2\} \cup \{r_1 \mapsto c_3\}}$

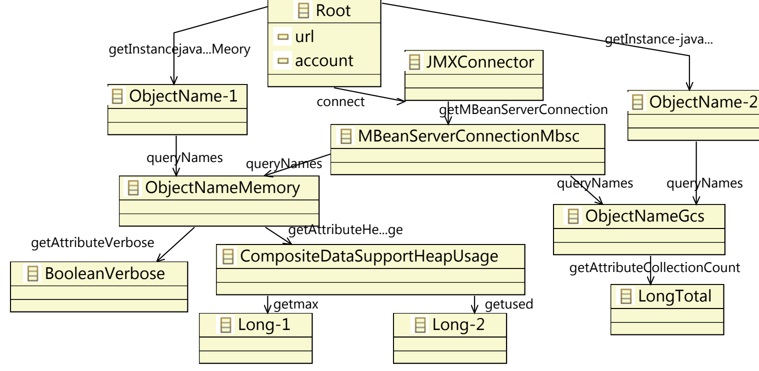

Fig. 6. The raw meta-model

For example, the class `ObjectName-1` (as c_2 in the above formula) has no other usage except serving as an intermediate during the access from `Root` to `ObjectNameMemory`. We remove this *forwarding class*, and link the two classes.

Rule 3. *Remove shortcut references:* $\dfrac{c_1 \xrightarrow{*} c_2,\ c_2 \xrightarrow{r_1} c_3,\ c_1 \xrightarrow{r_2} c_3}{c_1 \leftarrow c_1 - \{r_2 \mapsto c_3\}}$

After removing `ObjectName-1` (according to Rule 1), `Root` (as c_1) could reach `ObjectNameMemory` (c_3) in two ways: one is through a direct association, and one is through an intermediate class `MBSC` (c_2) . That means to obtain an `ObjectNameMemory` instance, we need both an `MBSC` instance and its parent, a `Root` instance. Since an instance's parent always exists, the second path implies the first one, and thus we remove the direct association.

Rule 4. *Weaken the leaf classes to attributes.* $\dfrac{c_1 \xrightarrow{r_1} c_2,\ c_2.a{=}\phi,\ c_2.r{=}\phi}{c_1 \leftarrow c_1 - \{r_1 \mapsto c_2\} \cup \{a \mapsto O(c_2)\}}$

From an instance of `ObjectNameMemory` (c_1), we can access an association (r_1) to obtain an instance of `BooleanVerbose` (c_2). This instance does not have its own states nor points to other instances, but only depicts if the memory is verbose or not. Therefore, we regard it as an attribute of `ObjectNameMemory`. The function O means getting original primitive data type.

Rule 5. *Removing dangling classes.* $\dfrac{root \overset{*}{\not\rightarrow} c}{Pack \leftarrow Pack - \{c\}}$

After applying the previous rules, some associations may be removed, leaving some classes unreachable from `Root`. We simply remove such dangling classes.

For a generated meta-model, we apply the rules iteratively, until it does not satisfy any anti-patterns. Figure 7 shows the simplified result of our running example. This result is quite close to Figure 2(b), ignoring the difference on names. The only structural difference is that we generate a redundant class, `MBean--tion`, because the current algorithm cannot differentiate control element and data element of the API.

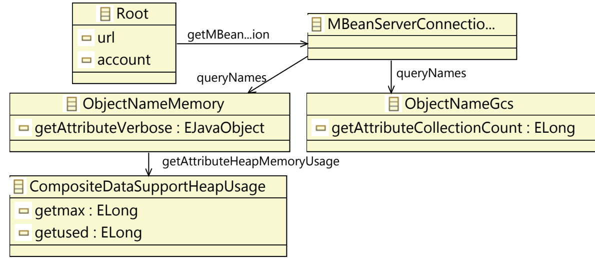

Fig. 7. The simplified meta-model

6 Implementation and evaluation

We implement the whole approach on the Eclipse platform, providing an automated meta-modeling tool for runtime system data. Specifically, we utilize Eclipse Java Development Tools (JDT) to represent the source code as an Abstract Syntax Tree, utilize Eclipse Modeling Framework (EMF) to represent the meta-model, and define a simple model to represent DTGs. We also develop an Eclipse plug-in to implement all the algorithms presented in this paper.

We evaluate this approach from three aspects. We first present a case study to show that *it can be used for realizing runtime model*. Then we report a set of experiments to show that *it applies to a wide range of systems*. Finally, we use two quantitative criteria, namely accuracy and efficiency, to evaluate the above experiments, revealing that *the approach generates good meta-models*.

6.1 The JOnAS case study

Utilizing the meta-modeling tool presented in this paper, we upgrade an original runtime model case on JOnAS [4] with a more complete meta-model, which is automatically inferred from two API clients: 1) an official sample shipped with the release version of JOnAS, and 2) the official web-based JOnAS administration console, *jonasAdmin*, which organizes and displays the runtime data by HTML pages. The clients and the meta-models are described in Table 1. These generated meta-models work well with the original SM@RT tool, maintaining a runtime model representing the running JOnAS server.

We evaluate the validity of this runtime model by a complete traversal on it. This traversal successfully passes through at least one model element for every class[4].The traversal successfully receives a value (including “null”) for almost every attribute, except for 17 ones (out of 424) where the system throws exceptions. The average number of attributes contained by the elements is about 7, and most elements contain less than 10 attributes. The traversal reaches the attributes from the root in about 5 steps on average. A careful analysis reveals that this meta-model covers 46 different types of MBeans, and 361 effective and unique attributes of them.

[4] We configured the server carefully, so that it contains all kinds of system data

Table 1. Experiments and results. In *experiment design*, we list the target systems, their APIs, the clients and their sizes. In *result*, we show the sizes of the generated meta-models (including the numbers of classes, attributes and references), and the time spent for analyzing. In *evaluation*, we list the meta-models we choose as references, and the accuracy and efficiency of the generated meta-model, comparing with this reference.

	Experiment design				Result				Evaluation		
#	System	API	Client	LOC	cls	attr	ref	Time	Reference	Acc	Eff
1	Glassfish	JMX	Home-made	24	5	6	4	2.2	Figure 2	1	0.87
2	JOnAS	JMX	J2EEMgmt	870	9	22	10	2.8	MBeans	1	0.71
3	JOnAS	JMX	jonasAdmin	16K	59	424	58	15.2	MBeans	0.96	0.89
4	Equinox	OSGi	Eclipse console	5K	13	42	12	19.3	Internal classes	0.80	0.56
5	Jar	BCEL	JarToUML	3K	21	62	20	4.9	Official Doc	0.84	0.31
6	WSDD	DOM	Axis Admin	813	8	23	7	1.8	Official Schema	0.89	0.88

6.2 Experiments and results

We undertook a set of experiments to see the applicability of this approach, as summarized in Table 1. The first row describes the motivating example. #2 and #3 are the cases mentioned in the above case study. #4 is an attempt to model the bundle information of Equinox platform, and the client is the command-line console provided by Eclipse. #5 is originally a use case of MoDisco [6], aiming to model the Java class structure through byte code files. We utilize the source code of the original implementation on BCEL. #6 is an attempt to model the web service deployment descriptor, utilizing the WSDD analyzer in Axis.

For all the experiments, the tool generates meta-models. The size of the meta-model depends on both the size and the function of the client. Regarding the performance, these experiments take 2 to 20 seconds. Since the analysis task only need to performed once for a system, this performance is tolerable.

6.3 Experiment evaluation

This section evaluates the experimental results, to see whether the approach infers *good* meta-models for the above experiments. We found a reference meta-model for each system. For the running example, the reference meta-model was the one shown in Figure 2. For #2 and #3, the references were reversely constructed by us according to the final runtime model and our knowledge about JOnAS. When doing this, we carefully examined the runtime model to filter out wrong or duplicated classes and attributes. For #4 to #6, we derived the reference meta-models manually from the internal implementation, the official documents or the XML schema.

We evaluate the inferred meta-models by checking if they are *close* to the reference ones, using two quantitative criteria named *accuracy* and *efficiency.* **Accuracy** assesses how much content in the inferred meta-model also appears in the reference one. For a generated meta-model M and its reference R, we regard their attributes ($M.a$ and $R.a$) as the definitions to actual system data.

We determine their subsets $\overline{M.a}$ and $\overline{R.a}$ which are the common parts between the two sets of attributes (ignoring difference between names), and define $acc(M,N) = \left|\overline{M.a}\right| / |M.a|$. **Efficiency** assesses for the same data, the ratio of elements used in the reference meta-model divided by the ones used in the generated one. For $\overline{M.a} \subseteq M.a$, we construct a sub meta-model $Prune(M, \overline{M.a})$ by pruning M of the attributes that are not in $\overline{M.a}$, as well as the classes from which any attributes in $\overline{M.a}$ cannot be reached. We prune R in the same way, and define efficiency as $eff(M,N) = \left|Prune(R, \overline{R.a})\right| / \left|Prune(M, \overline{M.a})\right|$.

If M and N are isomorphic, accuracy and efficiency all equal to 1. The accuracy declines if we infer wrong definitions. The efficiency declines if we construct duplicated attributes, or redundant classes. We do not evaluate "completeness" here, because the inputted API clients cannot cover all types of system data.

As shown in Table 1, *The accuracy is generally good*: more than 80% of the inferred data types really define the system data. The wrong data are usually caused by regarding control-purpose API accesses as data retrievals. For the clients mainly used for representing data (#3), the accuracy is higher. Although the reference model for #3 is not official defined, the accuracy score is not overestimated, because all the attributes are strictly examined according to the real JOnAS system. *The efficiency is not ideal.* We infer many duplicated data types, if the client developer used different ways to obtain the same type of data. Take #5 for example, the developer obtains Java classes from different sources, like Jar file, eclipse project, file system, etc., and we wrongly regard them as different types.

From the experiments, we learn two things. First, the quality of inferred meta-model depends on the purpose of the client. It is better to choose the client that purely represents system data (like #3). Second, we need further techniques to differentiate data access operations and the configuration operations, as well as the techniques to identify and merge duplicated data types.

7 Related Work

Runtime model is a hot research topic [2]. Its original intention is to introduce model-driven methodology into the runtime stage of the software life-cycle. P. Oreizy et al. [11] utilize software architecture to support the monitor and control of system structures. H. Goldsby et al. [12] use models to help represent and reason about the system and environment states. Many other related approaches can be found in the annual MoDELS workshop named Models@Run.Time. To support such runtime model applications, some researchers are trying to provide generic tool to realize runtime models. Rainbow [3] models the system data based on the input architecture style. Jade [5] represents system data as a Fractal model, guided by the provided Fractal ADL. The MoDisco project [6] provides reusable components for "model discovers" that represent system data as MOF compliant models, according to the MOF meta-models. The authors of this paper also present a SM@RT tool [4] to generate a runtime model synchronizer from

a MOF meta-model. These tools all require users to provide a meta-model first. The approach in this paper attempts to provide such meta-models automatically.

There is some related work targeting at inferring the data types for XML data [13] or plain text data [14]. The basis for their inference is the "shape" of the target data. Since the runtime system data are intangible, we cannot infer the type based on the data themselves, but on the usage of data.

Turning to the techniques, the kernel of our meta-model inference is the static code analysis on API clients. There are related approaches on analyzing the source code of API clients to establish the relation between API types, like Prospector [15] and XSnippet [9]. These approaches tell developers how to reach a specific type of API instance (an object of a class defined by the API). They are not enough for inferring the underlying data types and relations of the management APIs, because API instances of the same type may stand for different kinds of system data.

Our idea is also related to the engineering of domain-specific modeling languages: We want to model runtime system data, but different systems provide different types of runtime data. This approach constructs proper meta-models specific to the systems. Our solution of constructing such meta-models from API clients is also similar to FSML [16]. The difference is that FSML focuses on the APIs themselves, but we target at the data hidden behind the APIs.

8 Conclusion

In this paper, we present an automated approach to inferring the type of runtime system data by analyzing the client code of the management APIs. The result data types are defined as MOF meta-models that can be used to construct runtime models for the systems. We successfully use the approach with our previous runtime model supporting tool, and further experiments illustrate that the approach applies to a wide range of systems and the inferred meta-models are close to the reference ones. Note that the inferred meta-model are not complete, only defining the system data covered by the chosen API client. But since many of the clients are the official and widely used management tools, the data covered by them are enough in most cases.

The major limitation of the current approach is that its effect depends on the input clients. For the clients that have other functions than representing system data, or retrieving same data in different ways, the approach may infer wrong or redundant data types. We plan to improve this by precise classification of API accesses, as well as more effective duplicate checking. Knowing what kind of data can be accessed is only the first step, we will investigate the way to automatically extract how to access each type of data by slicing the client code according to the meta-model.

ACKNOWLEDGMENT. This work is sponsored by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60873060, 60933003,

60873060; the High-Tech Research and Development Program of China under Grant No. 2009AA01Z16, 2008AA01Z139; the EU Seventh Framework Programme under Grant No. 231167; the Program for New Century Excellent Talents in University; the Science Fund for Creative Research Groups of China under Grant No. 60821003. Thank Prof. Zhenjiang Hu in National Institute of Informatics (Japan) for his early discussion and suggestion on this work.

References

1. Shannon, B.: Java Platform, Enterprise Edition 5, Specifications (April 2006)
2. Blair, G., Bencomo, N., France, R.: Models@ run.time. Computer **42**(10) (2009) 22 –27
3. Garlan, D., Cheng, S., Huang, A., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
4. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating synchronization engines between running systems and their model-based views. In: MoDELS Workshops 2009, LNCS 6002. (2009) 140–154
5. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: International Conference on Software Engineering (ICSE). (2008) 101–110
6. MoDisco Project `http://www.eclipse.org/gmt/modisco/`
7. Sun Microsystems `http://java.sun.com/javaee/community/glassfish/`
8. Horwitz, S., Reps, T.: The use of program dependence graphs in software engineering. In: International Conference on Software Engineering (ICSE). (1992) 392–411
9. Sahavechaphan, N., Claypool, K.: XSnippet: Mining for sample code. In: International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA). (2006) 413–430
10. Ramanathan, M., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: International Conference on Software Engineering (ICSE). (2007) 240–250
11. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: International Conference on Software Engineering (ICSE). (1998) 177–186
12. Goldsby, H., Cheng, B.: Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In: Model Driven Engineering Languages and Systems (MoDELS). (2008) 568–583
13. Garofalakis, M., Gionis, A., Rastogi, R., Seshadri, S., Shim, K.: XTRACT: a system for extracting document type descriptors from XML documents. ACM SIGMOD Record **29**(2) (2000) 165–176
14. Fisher, K., Walker, D., Zhu, K., White, P.: From dirt to shovels: Fully automatic tool generation from ad hoc data. In: Principles of Programming Languages (POPL). (2008) 421–434
15. Mandelin, D., Xu, L., Bodik, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: Programming Language Design and Implementation (PLDI). (2005) 48–61
16. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of framework-specific modeling languages. IEEE Trans. Software Eng. **35**(6) (2009) 795–824