Specifying Downloadable Properties for Reusing Software Components: A Case Study of Java

Haruhiko Kaiya, Kenji Kaijiri
Faculty of Engineering, Shinshu University
4-17-1 Wakasato, Nagano City, 380-8553, JAPAN
kaiya@cs.shinshu-u.ac.jp kaijiri@cs.shinshu-u.ac.jp
http://www.cs.shinshu-u.ac.jp/˜kaiya/

Abstract. In this paper, we propose a specification of software components which can be loaded not only from your local system but also from the other systems over the computer network. Because components from the other system are not always enough reliable or safe to act freely in your own system, you should limit their activities to a certain context. Because some existing systems like Java RMI and an Applet provide a mechanism for such limitation implicitly, users sometimes lose sight of the abilities and limitations of such components. Therefore, they fail to reuse the components in the right way. We provide a way to specify such properties, so that component users can precisely understand the abilities and limitations.

1 Introduction

For reusing software components, we need suitable documents of specification. Unfortunately, as many of such documents tend to be long, incomplete, redundant or vague, component users sometimes can not reuse such components adequately. For example, the specification of RMI[1] consists of about 90 pages of documents and it is hard to read completely. But small manuals in many books are too simple to understand the components. One of the suitable tool for reuse is formal method. Though writing a formal specification of components looks like expensive, we can retrieve the costs for using the components and the specification repeatedly [2]. Several formal or semiformal specification of components are already used, e.g. signatures, pre/post conditions and invariants. But we can not simply specify the mobility of components [3], because each component will change its behaviors along with its birthplace, running place and so on.

In this paper, we propose a way of specification for mobile components, which are not autonomous but downloaded by the other components or systems. We call such kind of mobile components as *downloadable components* in this paper. With our specification, users of downloadable components will be able to use them adequately. Though the specification here is developed through the case study of Java, we do not limit the discussion to Java, but discuss how downloadable components should be designed from the users' point of view.

In section2, we introduce the properties and problems of downloadable components. In section3, we summarize what should be specified in addition for such properties, and we give an example in section4 to show the advantages of the specification. In the last section, we summarize our results and discuss the future works.

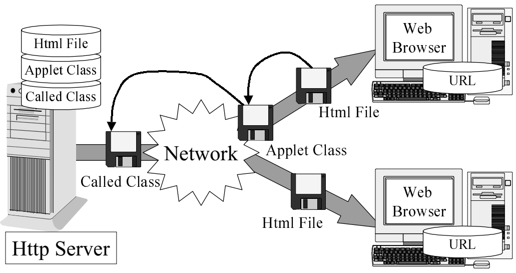

Figure 1: Applet Overview

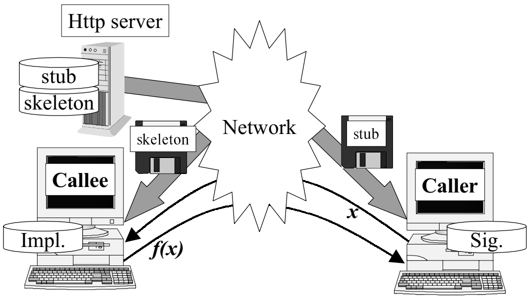

Figure 2: RMI Overview

2 Properties and Problems of Downloadable Components

In this section, we introduce examples, underlying mechanism and problems of downloadable components. Because examples in this section are all about Java system, we regard the term '*class*' in the same light as the term 'component'.

2.1 What is Downloadable Component?

The most famous system using downloadable components is an Applet. In Figure1, we illustrates the typical behavior of an applet. In this paper, the icons of diskette represent downloadable components, and the icons of cylinder represents the components that are deployed in the local system. Web browser running an applet only knows an URL of html file, which is a bootstrap script, where the filename of the applet class is specified. After loading the script, the applet class and other classes called from the applet class are loaded one after another. Therefore, we can use sophisticated reusable class libraries like AWT graphics and Java Beans on any browser. Because the loaded classes are not enough reliable or safe, the classes are not allowed completely to operate system resources, e.g. file system or network connections, among the browser side system.

RMI (Remote Method Invocation) is regarded as object-oriented RPC (Remote Procedure Call) for Java[1]. In Figure2, we illustrates the outline of RMI. For establishing RMI, both a *caller* (or RMI client) program in a machine and a *callee* (or RMI server) program in another machine should be able to use components, *stub* and *skeleton*, for passing the arguments of the procedure and for receiving the results over the network. Because the components are not enough safe to operate the system resources of the caller and callee, their behaviors should be also limited. In RMI system, in contrast with an Applet, programmers should explicitly manage the limitation by the `SecurityManager` in Java. Therefore, you can use the downloadable components without limitation, if you want.

2.2 User-defined Class Loaders

Thanks to *user-defined class loaders* in Java Virtual Machine (JVM) [4], we can use the downloadable components and manage their behaviors in Java. The purpose of class loaders is to support dynamic loading of software components on the Java platform [5]. JVM has two different kind of class loaders, *a system class loader* and user-defined class loaders [6]. Because each user-defined class loader can limit the behaviors of components loaded by the loader respectively, a component can behave differently with respect to its loader.

In Java application, the system class loader is used to load a class, whose `main` method act as a bootstrap, from the local file system. All classes used directly in the class are also

```
import java.net.*;
import java.io.*;

public class DLoader extends ClassLoader{
    // several definitions are omitted .....

    protected Class loadClass(String name, boolean resolve)
    throws ClassNotFoundException{
        Class clazz = null;

        try{ return findSystemClass(name); }
        catch(ClassNotFoundException e1){}
        catch(NoClassDefFoundError e2){}

        clazz=findLoadedClass(name);
        if(clazz != null){ return clazz; }

        clazz=findClass(name);
        if(clazz == null ){ throw new ClassNotFoundException(name); }

        return clazz;
    }

    private synchronized Class findClass(String name){
        // finding byte code from the network resource and define class.
        // .....
              return defineClass(name, data, 0, total);
        // .....
    }

    // several definitions are omitted .....

    public static void main(String args[]){
        try{
            DLoader loader=new DLoader(new URL(args[0]));
            Runnable cmd=(Runnable)loader.loadClass(args[1]).newInstance();
            cmd.run();
        }catch(Throwable e){
            e.printStackTrace();
        }
    }
}
```

Figure 3: Example: Simple User-defined Class Loader

loaded from the loader[1]. An user-defined class loader is implemented as a subclass of `ClassLoader`, and used as shown in Figure3. A class loader `DLoader` defined in this figure is loaded a component `args[0].class`, which is implemented the `interface Runnable`, from the network stream.

We can decide the policy of security in the following steps.

1. Decide whether your class loader takes account of the `SecurityManager` or not. For example, the loader in Figure3 does not, but loaders of RMI and an Applet does.
2. Decide each security policies listed in class `SecurityManager`. The manager class has about 30 numbers of check lists for limiting the operations for the system resources, e.g. file systems and network connections[1]. You can implement your own manager as a subclass of `SecurityManager`. For example, the manager for RMI, i.e. `RMISecurityManager` disallows most of all system operations.

Then, the decision affects the behaviors of loaded classes as follows.

- If a class loader is designed to require its `SecurityManager` but the manager is not given, the class loader itself can not run. As a result, classes loaded by the loader also can not run.
- If a loader is not designed to require its `SecurityManager`, the loader itself and classes loaded by it can run.
- Methods in Java class libraries for operating the system resources are designed to refer the related check lists in the `SecurityManager`, if the manager is defined. Therefore, the activities of each method is limited by the manager.

2.3 The Problems: Loader Selection & Deployment

As mentioned above, Java has flexible but complex mechanism for loading its components. As a result, without deep understanding of class loading and security system, component users tend to lose sight of the abilities and limitations of such components, especially those downloaded from the other system over the network. Here we summarize the problems.

As explained above, a component of Java changes its behaviors along with its class loader. Therefore, component users should know the specification of components with their loaders. But it is not so easy because

- application programmers normally do not care about the class loaders,
- and the *defining loader* is dynamically decided according to both the logic of the *initiating loader*. and the deployment of components.

If a component `C` is the result of `L.loadClass()`, a loader `L` is called initiating loader of C. If a component `C` is the result of `L.definClass()`, a loader `L` is called defining loader of C [5].

The logic of the initiating loader specifies the order for selecting defining loader. For example, `DLoader` in Figure3 uses the following order to select its defining loader; 1. system class loader (line 11-13), 2. cache of this loader (line 15-16), 3. `defineClass` method of this class via the method `findClass` (line 18-19).

As a result, components in the local system, which will be loaded by the system class loader, have the priority in `DLoader`, therefore the security manager sometimes gives no effect to the components. Loaders of RMI and an Applet have similar logic of selection.

3 Additional Specification for such Properties

For specifying the downloadable properties and its problems mentioned in section2. We should describe the following items in the specification of the components.

1. **Security policies for each components**: Because downloadable components are not enough reliable or safe to act freely, we should specify how a component may act and may not.
2. **Birthplace of each component**: Because the birthplace of a component is good indicator for its reliability, it should be also specified.
3. **Location for running the components**: Because relative location of a birthplace from the location where the component will run defines its reliability, we should also specify where the component will run.
4. **Deployment of components in each location(or machine)**: Under the environment connected by the network, the deployment of each component are dynamically changed in every moment. Therefore we should globally specify such deployment.
5. **Logic for selecting defining-loader**: In Java, because users can define the logic for loading components, the users can use flexible and complex mechanism for downloading them. Therefore we should specify such logic.

4 Example

In this section, we introduce an example to show how the specification in section3 is useful for component users to understand the properties of downloadable components. We use Z notation for our specification in this paper because it is widely known in the software engineering field, and because it is fit for object-oriented mechanism.

4.1 Stub with Cracking Code: Counter Example

Suppose we should use the RMI call for our system, even though the call contains cracking codes which will steal our password. For limiting such stealing, we use a `SecurityManager`. Moreover, we carefully deploy current version of stub codes so as to stop the progress of cracking in the codes. Here we check the safety of this situation. Note that we only take account of the item 1 in Section3 in the following specification.

First, we specify the system resources and their security limitation in *SysRes*. As the security limitation can change only at once by `SetSecurityManager` method in Java, we model this method in *SetLimit*.

$$\begin{array}{|l}
\mathit{SysRes} \\ \hline
\mathit{res} : R \mathrel{+\!\!\!\!\to} \mathit{Bool};~ \mathit{limit} : \mathbb{P}\,R \\ \hline
\mathit{limit} \subseteq \operatorname{dom} \mathit{res};~ \forall x : \mathit{limit} \bullet (x, \mathrm{false}) \in \mathit{res} \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\mathit{SetLimit} \\ \hline
\Delta \mathit{SysRes};~ l? : \mathbb{P}\,R \\ \hline
\mathit{limit} \neq \varnothing \Rightarrow l? = \mathit{limit}' \\ \hline
\end{array}$$

R is basic type which represents a set of resources. Schema *SysRes* represents a security level of this machine.

A method itself and the cracking codes can be represented as follows;

$\mathit{Func} \mathrel{\widehat{=}} [x?, y! : \mathbb{Z} \mid y! = f(x?)]$ $\qquad \mathit{Crack} \mathrel{\widehat{=}} [\mathit{pas}! : R, \Xi \mathit{SysRes} \mid (\mathit{pas}!, \mathrm{true}) \in \mathit{res}]$

All the method will be observed from the component users as follows;

$F \mathrel{\widehat{=}} (\mathit{Crack} \fatsemi \mathit{Func} \land \Xi \mathit{SysRes}) \setminus \{\mathit{pas}!\}$

Then we check the following expression,

$\mathit{SetLimt} \fatsemi \mathit{Crack} \fatsemi (\mathit{Func} \land \Xi \mathit{SysRes}) \mid \mathit{pas}! \in l?$

This represents that cracking is established even if corresponding operation is protected by the security manager.

This schema is inconsistent because both $(\mathit{pas}!, \mathrm{true}) \in \mathit{res}$ and $(\mathit{pas}!, \mathrm{false}) \in \mathit{res}$ are satisfied at the same time even if *res* is function. Therefore, we can conclude that the cracking is never established and our protection is enough safe.

Unfortunately, the cracking can be established in fact because the policy of security checking is changing along with the birthplace of components. We will resolve this problem in the next.

4.2 Stub with Cracking Code: Resolved

We introduce two additional basic types, *Loc* for location of byte codes, and *ByteCode* for realizing a class and an instance in run time. Then we define the deployment of byte codes over the network (the item 4 in Section3) as follows;

$$\begin{array}{|l}
\mathit{deploy} : \mathit{Loc} \mathrel{+\!\!\!\!\to} \mathbb{P}\,\mathit{ByteCode}
\end{array}$$

We extend *SysRes* schema with the location where the components are running (the item 3 in Section3), and introduce *Class* schema for representing the birthplace, byte code and the logic for selecting the birthplace of the class (the item 2 & 5 in Section3).

$$\begin{array}{|l}
\mathit{SysRes} \\ \hline
\mathit{res} : R \mathrel{+\!\!\!\!\to} \mathit{Bool};~ \mathit{limit} : \mathbb{P}\,R;~ \mathit{here} : \mathit{Loc} \\ \hline
\mathit{limit} \subseteq \operatorname{dom} \mathit{res} \\
\forall x : \mathit{limit} \bullet (x, \mathrm{false}) \in \mathit{res} \\
\mathit{here} \in \operatorname{dom} \mathit{deploy} \\ \hline
\end{array}
\qquad
\begin{array}{|l}
\mathit{Class} \\ \hline
\mathit{birth} : \mathit{Loc};~ \mathit{byte} : \mathit{ByteCode} \\
\mathit{lslctr} : \operatorname{seq} \mathit{Loc} \\ \hline
\mathit{birth} \in \operatorname{ran} \mathit{lslctr} \\
\mathit{birth} \in \operatorname{dom} \mathit{deploy} \\ \hline
\end{array}$$

Then the attribute *birth* in schema *Class* is defined the following schema.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SetLoader}} \\
\hline
\mathit{sl}?;\ \mathrm{seq}\,\mathit{Loc};\ \Delta \mathit{Class} \\
\hline
\mathit{lslctr}' = \mathit{sl}? \\
\forall x, y : \mathbb{N} \bullet \mathit{byte} \in \mathit{deploy}\ \mathit{lslctr}'\ x \land x \in \mathrm{dom}\,\mathit{lslctr}' \land \mathit{lslctr}'\ y = \mathit{birth}' \Rightarrow y \leq x \\
\hline
\end{array}
$$

The second line of predicate part (the item 5 in Section3) is slightly complex, but it simply says that the first location which involves *byte* in *lslctr* is its *birth*.

The schema *Crack* is modified as follows for representing the effect of birthplace.

$$\mathit{Crack} \mathrel{\widehat{=}} [\mathit{pas}! : R;\ \Xi \mathit{SysRes};\ \Xi \mathit{Class} \mid \mathit{here} \neq \mathit{birth} \Rightarrow (\mathit{pas}!, \mathrm{true}) \in \mathit{res}\ ]$$

Then the following expression becomes consistent,

$$\mathit{SetLimit} \fatsemi (\mathit{SetLoader} \land \Xi \mathit{SysRes} \fatsemi \mathit{Crack} \fatsemi \mathit{Func} \land \Xi \mathit{Class} \land \Xi \mathit{SysRes}) \setminus \mathit{Class}$$
$$\mid \mathit{pas}! \in l? \land \mathit{sl}? = \langle \mathit{here}, \mathit{there} \rangle$$

under the situation of $\mathit{deploy} = \{(\mathit{here}, \{\mathit{byte}, \cdots\}), (\mathit{there}, \{\mathit{byte}, \cdots\}) \cdots\}$. In Figure2, *here* corresponds to 'Call', *there* to 'Http server', and *byte* to 'stub'. As a result, cracking can be established even if corresponding operation is protected by the security manager under this situation. So if you want to stop the cracking, you should change *sl*? or *deploy*.

5 Discussion

In this paper, we discuss and propose how to specify downloadable properties of software components for suitable reuse, through the case study of Java. Though the style of specification here is not different from the style of traditional specification, we can clarify how and what kind of items should be described. Formalization for JVM is already proposed [7] and the security issue of Java is also reported [8], but these researches are not intended to encourage the reuse of components.

From our case study, we can discuss how downloadable components should be designed from the viewpoint of users. For example, in a system where each component can have different security policy, we can develop more flexible but complex system. Also the reliability of each component can be decided not only by its birthplace but also its frequency of use. Now we have no metrics for measuring the trade-off between the flexibility of the system and the requirements of users, we can not rationally select the suitable language system for developing a system for the users. Formal description of components may also contribute to make such metrics.

References

[1] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, Feb. 1997. Revision 1.4, JDK1.1 FCS.

[2] Bertrand Meyer. The Next Software Breakthrough. *COMPUTER*, 30(7):113–114, Jul. 1997.

[3] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, Sep. 1997.

[4] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley Longman, second edition, Apr. 1999.

[5] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proc. of OOPSLA*, pages 36–44, Oct. 1998.

[6] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, first edition, Mar. 1997.

[7] T. Jensen, D. Le Metayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalization. In *Proc. of Int. Conference on Computer Languages*, pages 4–15, May 1998.

[8] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proc. of IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.