環境要素間の依存関係に基づく部品仕様の構造化法

海谷 治彦　　落水 浩一郎

北陸先端科学技術大学院大学 情報科学研究科
〒 923-1292 石川県 能美郡 辰口町 旭台 1-1

電話: 0761-51-1262　　ファクシミリ: 0761-51-1360
電子メール: kaiya@jaist.ac.jp　　ochimizu@jaist.ac.jp

あらまし　ソフトウェア部品を利用する際には，その利用環境も合わせて仕様化する必要がある．しかし，利用環境の仕様を直接，部品に埋め込んだ場合，仕様の可読性および保守性が低下する．本稿では，Java の RMI 部品の仕様化を例題とし，リモートオブジェクト内部の抽象状態を，そのオブジェクトが利用される実行環境の特徴から分類した．そして，その特徴をもとに RMI 部品の仕様を記述した．これによって，任意のリモートオブジェクトを仕様化する度に，その制約条件を新たに記述する必要がなくなり，かつ，仕様の記述内容も簡潔なものとなった．

キーワード　部品化/再利用，実行環境，Java，リモートメソッド呼び出し，Z 記法

Structuring Components' Specifications using Dependencies among the Runtime Environments

Haruhiko Kaiya　　Koichiro Ochimizu

School of Information Science
Japan Advanced Institute of Science and Technology, HOKURIKU

1-1, Asahidai, Tatsunokuchi-machi, Nomi-gun, Ishikawa, JAPAN, 923-1292
Phone: +81-761-51-1262　　Fax: +81-761-51-1149
Email: kaiya@acm.org　　ochimizu@jaist.ac.jp
http://www.jaist.ac.jp/~kaiya/

Abstract　*When we develop a software system with a resuable component, we should refer not only to its own specification, e.g. its signatures and constraints of input, output and internal values, but also to the relationships between the component and external environments. But the specification with such additional information tends to be a large document, hence it is too hard to read and maintain its descriptions. In this paper, we specify an external environment for RMI, Remote Method Invovation Components in Java using Z notation. And we also introduce a way to simplying and generalizing its specification. As a result, we can systematically describe the specification of any remote objects.*

Key words　*reusable component, runtime environment, Java, RMI, Z notation*

1 はじめに

ソフトウェア部品の仕様を形式手法を用いて記述することにより，その再利用が促進されると思われる [1]．しかし，従来の部品の形式仕様はシグニチャの定義や，部品自体の内部状態や入出力値に依存する条件が記述されている程度である．モバイルコード [2] が広範囲に利用されるようになった現在では，ソフトウェア部品は，その内部状態だけに依存して動作が決定されるのではなく，その部品が配置されるマシンの状況，実行されるマシンの状況，部品コードが配送される経路などに依存して，その振る舞いを変える場合がある．我々はこのような状況や経路を，実行環境，もしくは単に環境と呼ぶことにし，このような環境を考慮した上でソフトウェア部品の仕様記述を行った．具体的には，Java 言語の RMI 部品 [3] を例題とし，その部品と実行環境との関係を Z 記法 [4] で仕様化した．これによって，多種多様な環境下で RMI 部品を利用する際のガイドラインを与えることが可能となった [5].

しかし，[5] での仕様化では，部品の振る舞いと実行環境の関係を個々の場合について記述しており，他の部品の仕様を記述する場合のガイドラインなどがない．また，環境に関する部分の記述が増加したため，仕様の記述量が増加し，仕様を参照する場合や，記述内容を保守・改訂する場合の労力が従来の形式仕様や，自然言語などによる説明などに比べ増加する恐れがある．

そこで本稿では，RMI 部品を利用する際の，実行環境や利用方法の違いなどを考慮し，記述内容を構造化する例を示した．具体的には，アプレット内から RMI を用いる場合の例題を通して，

- 実行環境を考慮した仕様が重要であること
- 実行環境での依存関係をもとに構造化した仕様によって，その可読性が向上すること

を示した．例題のソースコードは付録に示すように，アプレットが RMI サーバーから適当な文字列を受け取り，それを表示するプログラムである．

続く §2 では，RMI と Applet を同時に利用する際の典型的な実行環境の例を示し，それぞれに正常に動作するか否かを説明する．次に，§3 において，構造化を考慮せず記述した部品の仕様と，その仕様をもとに推論できる性質について述べる．そして §4 において，実際に利用される実行環境の種類を考慮して構造化した仕様を示す．

2 三種類の異なる実行環境

図 1, 2, 3 に異なる三種類の実行環境を示す．図中の楕円は個別のプログラム，矩形は計算機境界を示している．また，実線は RMI 呼び出しを表し，破線はネームサーバーからリモートオブジェクトの参照を取得することを表している．さらに，点破線はアプレットを実行するためのクラスファイルのロードの経路を示している．

図 1 では，アプレットを構成するクラスファイルを開発中のディレクトリなどに配置し，それを `appletviewer` などのツールで実行のテストを行うような環境を表している．プログラムの開発の初期段階では，このような環境下で開発を行うと思われる．付録のプログラムは，この環境下で正しく動作する．

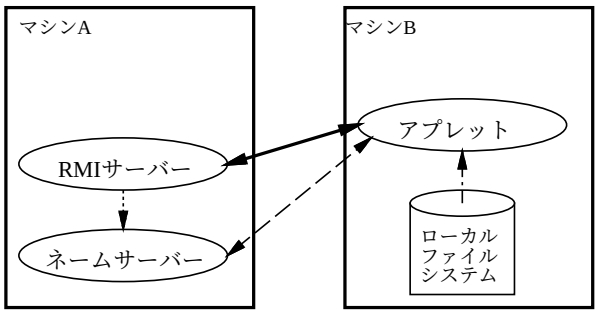

図 1: ローカルファイルシステムからコードをロードする場合 (成功)

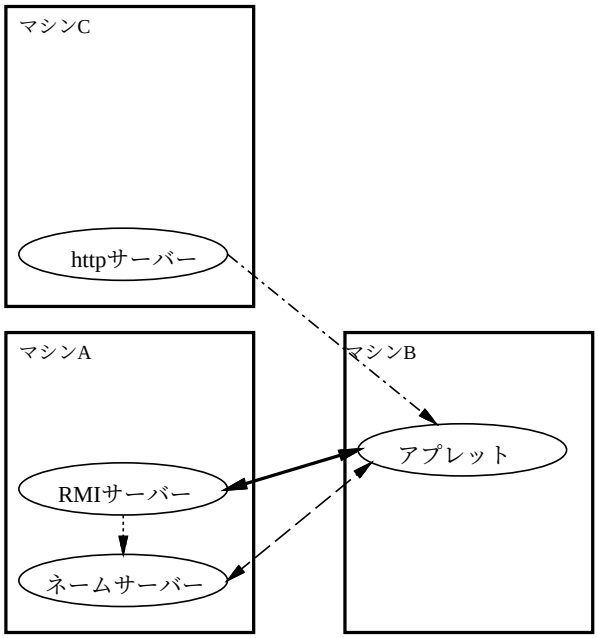

図 2: RMI サーバーが動作するホスト以外で動作する httpd からコードをロードする場合 (失敗)

図 2 では，アプレットを構成するクラスファイルを `httpd` からダウンロードして実行するような実行環境である．ただし，`httpd` の動作するホストと RMI サーバーの動作するホストが同一でない．多くのサイトでは共通の `httpd` を特定のホスト (例えばホスト名 `www` など) で利用する形態をとっているため，プログラム開発者がこのような実行環境下で動作テストをすることが多いと思われる．しかし，アプレットで RMI を用いる場合，コードをダウンロードするための `httpd` と RMI サーバー，そして RMI のネームサーバーは同一のホストで動作していたければならない．これは仕様書 [3, p.28] に説明されてはいるが，開発者にとっては見落としがちな個所である．よって，この環境下では付録のプログラムは正しく動作しないが，開発者がその理由を正しく突き止めることは難しい．

図 3 でも，アプレットを構成するクラスファイルを `httpd` からダウンロードして実行するような実行環境である．ただし，`httpd` の動作するホストと RMI サーバーの動作するホストが同一であるため，付録のプログラムは正しく動作する．

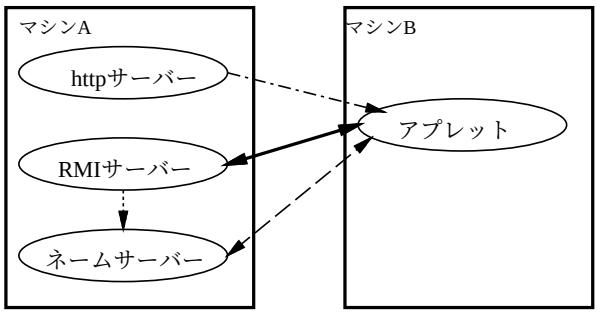

図 3: RMI サーバーが動作するホストで動作する httpd からコードをロードする場合 (成功)

3 構造化しない仕様

3.1 セキュリティマネージャー

Applet を利用する場合，RMI の振る舞いも `Applet-SecurityManager` に制約される．よって通常のアプリケーションとは異なり，アプレットを用いたクライアントでは，独自にセキュリティマネージャーを設定する必要はなく，また設定しなおすこともできない．

セキュリティマネージャーによって制約される動作の型を定義する．

$$Actions ::= CCONNETCT \mid CREAD \mid \ldots$$

実際には制約をチェックされる 23 の動作が規定されている．

それぞれの制約に対してブール値で設定状況を記録し，真の場合，実行可能とする．

さらに，クラスファイルをダウンロードする元のホストを dlhost，RMI コールをする先のホストを cldhost，そしてこのセキュリティマネージャーが適用されるホストを slfhost とする．

AppletSecurityManager

$checkTable : Actions \to \mathbb{B}$

$dlhost, cldhost, slfhost : Host$

$checkTable\ CCONNECT = (dlhost = cldhost)$

$checkTable\ CREAD = \text{false}$

$\vdots$

3.2 クラス: `HelloImpl`

リモートオブジェクトの実装である．クラスは，

```
class HelloImpl
   extends UnicastRemoteObject
   implements Hello
```

である．

`HelloImpl` は適当な文字列を返すメソッドを持つため，内部状態として文字列を仕様化する．

HelloImpl

$message : String$

さらに，セキュリティ制限による例外発生を報告するため以下の型を導入する．

$$Report ::= OK \mid Exception$$

`sayhello` の仕様は以下の通りである．

HelloImpl.sayhello

$out! : String;\ rep! : Report$

$\Xi HelloImpl;\ SecurityManager$

$out! = message;\ rep! = OK$

本メソッドでは，たまたま SecurityManager の 23 のチェック項目のどれにも無関係のため，SecurityManager 内の属性は参照されていない．

3.3 クラス: `Naming`

リモートオブジェクトをネームサーバーに登録するためのクラスである．クラスは，

```
public final class Naming
   extends Object
```

であり，

```
public static Remote
   lookup(String name)
public static void
   rebind(String name, Remote obj)
```

が例題に関係するメソッドである．

リモートオブジェクトの名前とそのオブジェクトの参照の表を持つように仕様化する．また，ネームサーバーが動作しているホスト名も nmhost として保持する．

Naming

$lookupTable : String \pfun Remote$

$nmhost : Host$

`rebind` は，名前をつけてリモートオブジェクトを登録するメソッドである．

Naming.rebind

$n? : String;\ o? : Remote;\ rep! : Report$

$\Delta Naming;\ SecurityManager$

$dlhost = nmhost \Rightarrow$

$\quad lookupTable'\ n? = o?;\ rep! = OK$

$\quad \text{dom}\ lookupTable \cup \{n?\} = \text{dom}\ lookupTable'$

$dlhost \neq nmhost \Rightarrow$

$\quad rep! = Exception$

`lookup` は，名前からリモートオブジェクトの参照を取得するメソッドである．

Naming.lookup

$shost? : String;\ sname? : String$

$r! : Remote;\ rep! : Report$

$\Xi Naming;\ SecurityManager$

$cldhost = shost?$

$(checkTable\ CCONNETCT$

$\vee slfhost = dlhost) \Rightarrow$

$\quad sname? \in \mathrm{dom}\ lookupTable \Rightarrow$

$\quad\quad lookupTable\ sname? = r!$

$\quad\quad rep! = OK$

$\quad sname? \notin \mathrm{dom}\ lookupTable \Rightarrow$

$\quad\quad \theta Naming = \theta Naming'$

$\quad\quad rep! = Exception$

$\neg(checkTable\ CCONNETCT$

$\vee slfhost = dlhost) \Rightarrow$

$\quad rep! = Exception$

3.4 部品の性質

AppletSecurityManager 下で sayhello が正しく実行されることは，以下を証明することに相当する．

$$HelloImpl.sayhello\ [SecurityManager/AppletSecurityManager] \vdash out! = message \wedge rep! = OK$$

sayhello は，SecurityManager のチェック項目のどれにも依存していないため，この結果は自明である．

コードをダウンロードしたホストとネームサーバーが動作するホストが一致し，名前を検索する先のホストがネームサーバーの動作するホストと一致するとする (図 3 の場合)．この条件下において，登録された名前でリモートオブジェクトの参照を獲得しようとすれば，それが獲得できる．この性質が成り立つことを示すことは以下を証明することに相当する．

$$Naming.rebind \wedge Naming.lookup \mid dlhost = nmhost \wedge nmhost = shost? \vdash n? = sname? \Rightarrow o? = r!$$

また図 1 の場合は以下のようになる．

$$Naming.rebind \wedge Naming.lookup \mid slfhost = dlhost \vdash n? = sname? \Rightarrow o? = r!$$

4 構造化した仕様

§3 では個々の記述毎に実行環境との関係を記述しており，その記述方針に一般性がない．そこで以下のような方針で実行環境の仕様を構造化する．

- リモートメソッド内の操作対象となる抽象状態を，セキュリティの観点から分類し，任意の抽象状態に対する性質を系統的に記述することを可能とする．

RMI では，操作する資源の型が，プログラム内の変数か，プログラム外のファイルシステムか，ネットワークポートかによって扱いが異なる．そのため，どの型に属するかの値を個々の資源とともに保持することによって，セキュリティマネジャーの扱いを一般化する．

$$ResType ::= Strage \mid FileSystemR \mid FileSystemW \mid Port$$

4.1 AppletSecurityManager の場合

`HelloImpl` は適当な文字列を返すメソッドを持つため，内部状態として文字列を仕様化する．

HelloImpl

$message : String \times ResType$

`sayhello` の仕様は以下のようになる[1]．

HelloImpl.sayhello

$out! : String;\ rep! : Report$

$\Xi HelloImpl$

$SecurityManager[a/message] \Rightarrow$

$\quad (out!, Strage) = message$

$\quad rep! = OK$

$\neg SecurityManager[a/message] \Rightarrow$

$\quad rep! = Exception$

セキュリティマネージャーは以下のようなデータ部を持つスキーマと仕様化する．例えば AppletSecurityManager の場合，

AppletSecurityManager

$a : String \times ResType$

$dlhost, cldhost, slfhost : Host$

$\forall x : String \bullet a = (x, Strage)$

$\forall x : String \bullet a \neq (x, FileSystemR)$

$\forall x : String \bullet a \neq (x, FileSystemW)$

$\vdots$

と定義する．これによって，個々のリモートオブジェクトの抽象状態の数や種類が増加しても，それぞれの状態に対してセキュリティマネージャーにどのように制約されるかを記述する必要がなくなる．

[1] スキーマの否定には定義部の型についての問題があるが，ここではその点には触れない．

この例では,

$$\begin{array}{l} \mathit{HelloImpl.sayhello} \\ \quad [\mathit{SecurityManager} \\ \quad /\mathit{AppletSecurityManager}] \end{array} \vdash \mathit{rep}! = \mathit{OK}$$

となる.

4.2 ファイルシステムを読む許可を与えたセキュリティマネジャー

ファイルシステムを読む許可を与えたセキュリティマネジャーでは,

$$\begin{array}{|l} \mathit{FSReadSecurityManager} \\ \hline a : \mathit{String} \times \mathit{ResType} \\ \mathit{dlhost}, \mathit{cldhost}, \mathit{slfhost} : \mathit{Host} \\ \hline \forall x : \mathit{String} \bullet a = (x, \mathit{Strage}) \\ \forall x : \mathit{String} \bullet a = (x, \mathit{FileSystemR}) \\ \forall x : \mathit{String} \bullet a \neq (x, \mathit{FileSystemW}) \\ \vdots \\ \hline \end{array}$$

などとなり，ファイルを読むことを含むリモートメソッド `readfile` [2]などに対して,

$$\begin{array}{l} \mathit{HelloImp.readfile} \\ \quad [\mathit{SecurityManager} \\ \quad /\mathit{FSReadSecurityManager}] \end{array} \vdash \mathit{rep}! = \mathit{OK}$$

となり,

$$\begin{array}{l} \mathit{HelloImp.readfile} \\ \quad [\mathit{SecurityManager} \\ \quad /\mathit{AppletSecurityManager}] \end{array} \vdash \mathit{rep}! = \mathit{Exception}$$

となる.

5 おわりに

本稿では，RMI 部品を利用する際の，実行環境や利用方法の違いなどを考慮し，記述内容を構造化する例を示した．構造化に際しては，仕様化対象の部品の内部構造，具体的にはセキュリティをチェックする項目をそのままに仕様化するのではなく，リモートオブジェクト内の抽象状態をセキュリティ制限の観点からの分類を利用して仕様化することとした．これによって，任意のリモートオブジェクトを仕様化する度に，その制約条件を新たに記述する必要がなくなり，かつ，仕様の記述内容も簡潔なものとなった．

[2]ソースコードは省略する

参考文献

[1] Bertrand Meyer. The Next Software Breakthrough. *COMPUTER*, Vol. 30, No. 7, pp. 113–114, Jul. 1997. IEEE/CS.

[2] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 213–239, Sep. 1997.

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

[4] J. M. Spivey. *The Z Notation, A Reference Manual, Second Edition*. Prentice Hall, 1992. `http://spivey.oriel.ox.ac.uk/~mike/zrm/`.

[5] 海谷治彦, 落水浩一郎. 実行環境に依存する部分を 含めたソフトウェア部品の形式的仕様の記述. 電子情報通信学会 技術研究報告, Vol. 98, No. 439, pp. 7–14, Dec. 1998. ソフトウェアサイエンス研究会 SS98-31.

付録: 例題のソース

インタフェースの定義

```
 1  // $ 1999-04-15 19:35:57+09 kaiya Exp $
 2
 3  import java.rmi.*;
 4
 5  public interface Hello extends Remote{
 6    public String sayHello() throws java.rmi.RemoteException;
 7  }
 8
```

リモートオブジェクトの実装

```
 1  // $ 1999-04-15 19:35:57+09 kaiya Exp $
 2
 3  import java.rmi.*;
 4  import java.rmi.server.UnicastRemoteObject;
 5  import java.net.*;
 6
 7  public class HelloImpl extends UnicastRemoteObject implements Hello{
 8  int cnt=0;
 9
10    public HelloImpl() throws RemoteException{
11      super();
12    }
13
14    public String sayHello() throws RemoteException{
15      cnt++;
16      System.err.println(cnt+": Hello Jaist!");
17      return "Hello Jaist!";
18    }
19
20    public static void main(String args[]){
21    String n=args.length>0? args[0]: "hello";
22      try{
23        System.setSecurityManager(new RMISecurityManager());
24      }catch(Exception e){
25        System.err.println("can not set SM, "+e);
26      }
27
28      try{
29        HelloImpl h=new HelloImpl();
30        Naming.rebind(n, h);
31        System.out.println(n+" ready as hello");
32      }catch(RemoteException re){
33        System.out.println("RemoteException ");
34      }catch(MalformedURLException e){
35        System.out.println("URL Excetion ");
36      }
37    }
38  }
39
```

Applet 起動のための html ソース

```
 1  <html>
 2  <title>Hello jaist</title>
 3  <applet codebase="." code="HelloApplet" width=200 height=100>
 4    <param name=server value=RMI サーバーが動作するホスト>
 5    <param name=service value=jaist>
 6  </applet>
 7  </html>
```

Applet 本体の java ソース

```
 1  // $ 1999-04-15 19:35:57+09 kaiya Exp $
 2  import java.awt.*;
 3  import java.rmi.*;
 4  import java.applet.*;
 5
 6  public class HelloApplet extends Applet{
 7    String message="", serv;
 8    public void init(){
 9      try{
10        serv="rmi://"+getParameter("server")+"/"+getParameter("service");
11        Hello obj=(Hello)Naming.lookup(serv);
12        message=obj.sayHello();
13      }catch(Exception e){
14        System.out.println(""+e);
15      }
16    }
17    public void paint(Graphics g){
18      // System.err.println(message);
19      g.drawString(serv+" said, ", 0, 10);
20      g.drawString(message, 0, 20);
21    }
22  }
23
```