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

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　ソフトウェア部品の中には，その部品を利用したプログラムの実行時の環境に依存して，振舞いが変化するものがあり，このような変化も部品とともに仕様化する必要がある．その仕様によって，プログラマが発見した誤りが，部品をプログラムに組み込む方法に原因があるのか，実行時の設定に原因があるのかを判断することが可能となると思われる．本研究では，上記の実例を示すために，Java 言語のソフトウェア部品である RMI クラス群の仕様化を行った．具体的には，最初に，シグニチャおよびクラスの内部状態とメソッドの入出力値をもとにした仕様記述を行い，それだけでは表現できないプログラムの振る舞いがあることを示した．そして，プログラム実行時のクラスファイルの配置などに依存する RMI の動的ロード機構に関する記述を仕様に追加することで，上記の振る舞いを表現することが可能となったことを示した．

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

Specifying Runtime Environments for RMI Components using Z

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　*Reuse can only succeed with formal methods, and formal methods can only succeed if used to develop reusable components. For specifying the reusable components sufficiently, we need the descriptions of runtime environments, as well as these of interface descriptions and contracts with client software. In this paper, we specify a runtime environment for RMI, Remote Method Invocation Components in Java using Z notation. The behavior of RMI components is influenced by the deployment of dynamically loaded classes, because the location of each class affects its authority in runtime. So we specify the original location of each classes as the runtime environment, in addition to the interface and the contract descriptions. With this specification, We exhaustively infer the runtime behavior of a sample implementation.*

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

1 はじめに

Bertrand Meyer はコラム [1] にて以下の提言をしている．

> 形式手法を用いた場合にのみソフトウェアの再利用は成功し，かつ，再利用部品の開発に用いた場合にのみ形式手法は成功を納める．

ソフトウェアに限らず再利用を促進するためには再利用対象の適切なレベルの抽象化が必要である．それを与える手段として形式手法は適切であることを前半は主張している．Ariane-5 ロケットの失敗[1] [2] は適切な仕様を持たない部品を再利用した結果だとしている．

後半は，形式手法が活用できる分野として再利用が適切であることを述べている．形式手法は多くの利点があるにもかかわらず，その適用は十分には浸透していない．それは，形式仕様の数学的記法が利用者に受け入れにくいからではなく，プログラム自身を記述するのに加えて，さらに追加的な記述をする必要があるからだとしている．よって，その追加的な記述をしてもなお利得のある分野でなければ，形式手法は浸透しないとしている．再利用部品は，その部品が不特定多数に数多く利用されるため，形式手法による追加的な記述，すなわち仕様は，その部品の意味を正確に伝えるという観点から十分に利得がある．さらに，部品の利用回数の増加によって，形式記述を行ったことへの平均コストも減少する．

しかし，実際には上記の提言にあった再利用部品の開発は行われていない場合がある．多くのプログラミング言語の部品，ライブラリやコンポーネントなどの仕様として形式的に与えられているものはシグニチャの記述程度である．例えば，Java 言語では，シグニチャおよび自然言語による構造化コメントなどを `javadoc` [2] と言われるツールによりソースコードから取り出すことができる程度である．

再利用部品に関して適切な抽象度の仕様を与えるために，Meyer は「契約による設計」[3] という仕様化の方針を提案している．この方法では，部品を利用するプログラム (クライアント) が部品を利用する際に満たしていなければならない条件 (事前条件)，結果として保証される性質 (事後条件) そして部品が利用されている間に維持され続けなければならない条件 (不変命題) の 3 つの仕様 (表明) を記述することを定めている．この方針は，Meyer 自身が提唱するプログラミング言語 Eiffel[3]や，`javadoc` を拡張して Java 言語にその機能を追加した iContract[4] などに適用されており，プログラム実行中にも表明をチェックする機能も有している．また，UML の OCL[4] にも同様の考え方が導入されている．

上記の例にあるような契約による設計では，再利用部品の内部にある属性値，および入出力値のみを利用して表明を記述している．しかし Java に代表されるような動的ロード機構を持つ言語の部品では，部品の入出力値および内部状態のみでは十分に仕様化できない場合がある．例えば，動的にロードされる部品が実行時において，ローカルシステム上のファイルシステムからロードされるのか，ネットワークを通して他のシステムからロードされるかによって，その部品の振る舞いは変化する．この例のように部品の一部が計算機システムのどの部分に配置されているなどのプログラムを実行する環境も利用しなければ，再利用部品の仕様化を十分に行うことができない場合がある．

そこで，本研究では，実行時の計算機環境も合わせて，Meyer の契約による設計の考え方をもとに再利用部品を仕様化する方法を提案する．実行環境の中でも特に動的ロード機構にかかわる部分に注目し，具体的には Java 言語の RMI クラスライブラリを仕様化する．仕様記述言語としては現在もっとも広く利用されている Z 言語 [5] を用いる．続く 2 節では RMI システムの概要を紹介する．そして，3 節では仕様記述の方針をまとめる．4 節で RMI を利用した例題の仕様記述例を説明する．そして，5 節では，他の仕様記述法との比較を通して本方法の利点や欠点について議論し，最後に今後の方針をまとめる．

2 RMI システムの概要

Java リモートメソッド呼び出し (RMI) は Java のオブジェクトモデルの意味論を保持した Java 専用の分散オブジェクトモデルであり，これによって分散オブジェクトを容易に実装し利用することが可能となる [6]．具体的には，他の Java VM からもメソッドを実行ことができるようなオブジェクトであるリモートオブジェクトを作成するためのクラス群が準備されており，これによってプログラマはプロセス間通信などの知識なしに，リモートオブジェクトを実装し利用することが可能となる．尚，リモートオブジェクトのメソッドを実行する側の Java VM は，異なる計算機上で動作していてもよい．一般にリモートオブジェクトを内蔵するプログラムをサーバープログラム，リモートオブジェクトのメソッドを呼び出すをクライアントプログラムと呼ぶ．

図 1 に RMI のシステムアーキテクチャの例を示す．2 つの異なる計算機に，サーバープログラム，クライアントプログラムそしてネームサーバープログラムが配置されている．図中の ‘メソッド呼び出し関係’ の矢印において，矢先が呼び出される側のオブジェクトである．また ‘オブジェクト’ という名前の楕円は `main` メソッドなどに相当するオブジェクトを表す．

リモートメソッド呼び出しが可能となるまでの手順を概観する．サーバープログラム側で，リモートオブジェクトを生成し，それを `Naming` クラスを利用して，ネームサーバー (`rmiregistry` というサンプル実装が存在する) に適当な名前をつけて登録する．このリモートオブジェクトを利用したいクライアントプログラムは，同様に `Naming` クラスを利用して，名前をキーとしてリモートオブジェクトの参照を取得する．この参照に対してメソッドを実行することによって，リモートメソッドを呼び出すことができる．プログラマにとっては，クライアントプログラム内のローカルオブジェクトのメソッドを呼び出す場合とほとんを変わらない方法でリモートオブジェクトのメソッドを呼び出すことができることに

[1] `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`

[2] `http://www.javasoft.com/products/jdk/javadoc/index.html`

[3] `http://www.eiffel.com/`

[4] UML1.1 Object Constraint Language Specification, `http://www.rational.com/uml/html/ocl/`

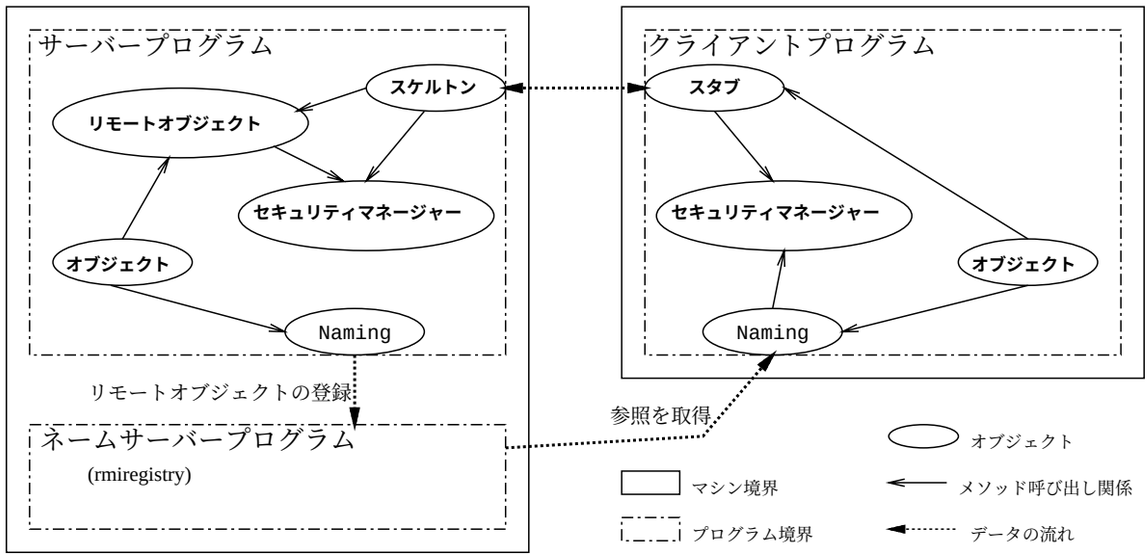

図 1: RMI のシステムアーキテクチャの例

なる．

クライアント側でのリモートメソッド呼び出しは，実際にはスタブと言われるオブジェクトのメソッドの呼び出しになっており，スタブはサーバープログラム側にあるスケルトンとプロセス間通信などを行うことにより，リモートメソッド呼び出しの際の引数をサーバープログラム側に渡したり，返り値を取得したりすることが可能となる．スケルトンから実際のリモートオブジェクトの実装を呼び出すことにより，実際の計算が行われる．尚，スタブおよびスケルトンはリモートオブジェクトをもとに，`rmic` とよばれるプログラムで生成することができるため，プログラマが別途記述する必要はない．

Java では RPC などとは異なり，スタブやスケルトンを含めたクラスをプログラム実行時に動的にロードする機構を有している．これによって例えばスタブやスケルトンをローカルファイルシステム上からではなく，全く別の計算機上から `http` や `ftp` などの標準的なプロトコルを用いてダウンロードすることが可能となる．

しかし，全く別の計算機からロードしたクラスが十分に信頼のおけるものである保証はない．そこでセキュリティマネージャーと呼ばれるクラスによって，ネットワーク越しにダウンロードしたクラスの動作，例えばファイルシステムへの書き込みや，Java VM の停止などを制限することになっている．RMI では，`RMISecurityManager` と呼ばれる標準的なセキュリティマネージャーが用意されており，これを利用するとクラス定義とアクセス以外の全ての機能の実行が制限される．尚，`RMISecurityManager` のサブクラスとして，より制限の少ないセキュリティマネージャーを設計することもできる．

3 記述方針

それぞれのオブジェクトの記述は，Meyer の「契約による設計」のスタイルに従うことにする．まず，オブジェクトの内部状態，および不変命題は，

オブジェクト名

状態

不変命題

とする．

そして，メソッドは以下のような記述とする．

オブジェクト名.メソッド名

Δオブジェクト名; 入力値?; 出力値!

Ξ参照する他のオブジェクトなど

事後条件

通常，スキーマ名には ‘.’ を用いないが，読みやすさのためにここでは用いることとする．

最後に，メソッドに関する明示的な事前条件がある場合は，

pre.オブジェクト名.メソッド名

オブジェクト名; 入力値?

参照する他のオブジェクトなど

事前条件

とする．Z 記法での通常の事前条件は，

Op

$\Delta State$; *Ins*?; *Outs*!

...

のような操作に対して，

$$preOp \mathrel{\widehat=} \exists State' ; Outs \bullet Op$$

と定義されている．これは，操作 *Op* を適用する前の状態と入力を用いて，操作適用後の状態が存在することを示している．しかし，たとえ適用後の状態が存在するとしても，それが意味ある値か否かはそれぞれの部品に依存する場合がある．よって，ここでは，Z の通常の意味での事前条件に加えて，必要ならば明示的に事前条件を記述できることにする．例えば，Java における例外処理,

```
try{
        メソッド呼び出し
}catch{ExcetionA e}{
        例外処理 A
}catch{ExcetionB e}{
        例外処理 B
}
```

の仕様においては，例外が起こるような状況を事前条件で制限することにする．

リモートオブジェクトの実行環境は，

$$LoadPlace \ ::= \ Local \mid Network$$

環境

$$impl, skel : LoadPlace$$

とし，上記のメソッドおよび事前条件の仕様中の ‘参照する他のオブジェクトなど’ と同様に取り込みを行うこととする．図 1 に示すように，実際のリモードオブジェクトの実装はスケルトンから呼び出される．よって，たとえ実装本体がローカルファイルシステムから呼び出されても，スケルトンがネットワーク越しにダウンロードされた場合には，セキュリティマネージャーの制限を受ける．

4 記述例

本節では，実際のアプリケーションが実行環境を考慮しない部品の仕様 (§4.1) から推論できない動作を行う場合があることを示し (§4.2)，実行環境を考慮した仕様を追加すれば，この不一致を解消できることを示す (§4.3).

セキュリティマネージャーが有効となるのは，スケルトンをローカルファイルシステムからロードする場合ではなく，ネットワーク越しにロードした場合である．よって，スケルトンの実行時のロード場所を ‘実行環境’ としてとらえ仕様化することで，ここでの不一致を解消することが可能となる．

4.1 実行環境を考慮しない仕様

以下に本例題で用いる RMI 関連の部品の仕様を示す．部品の実装例は付録 1 を載せる．これらは `javadoc` の出力や既存の参考書などをもとに記述した．

4.1.1 インタフェース: `MyInterface`

リモートオブジェクトのインタフェースの定義である．インタフェース自身は，

```
interface MyInterface
   extends Remote
```

であり，

```
String readfile(String file)
writefile(String file, String str)
```

のメソッドを持つ．

仕様は以下の通りであり，インタフェースは内部状態がないため状態スキーマはない．

MyInterface.readfile

$$f? : String;\ out! : String$$

MyInterface.writefile

$$f? : String;\ s? : String$$

4.1.2 クラス: `MyRObject`

リモートオブジェクトの実装である．クラスは，

```
class MyRObject
   extends UnicastRemoteObject
   implements MyInterface
```

であり `MyInterface` を実装しているため，メソッドは `MyInterface` と同じである．

`MyInterface` はファイルシステムの読み書きを扱うメソッドを持つため，内部状態としてファイルシステムを仕様化する．

MyRObject

$$serverFS : String \pfun String$$

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．

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

`writefile` の仕様は以下の通りである．

MyRObject.writefile

$$MyInterface.writefile;\ rep! : Report$$
$$\Delta MyRObject;\ \Xi MyManager$$

$$checkTable\ CWRITE \Rightarrow serverFS'\ f? = s?$$
$$\quad \mathrm{dom}\, serverFS \cup \{f?\} = \mathrm{dom}\, serverFS'$$
$$\quad rep! = OK$$
$$\neg checkTable\ CWRITE \Rightarrow$$
$$\quad \theta MyRObject = \theta MyRObject'$$
$$\quad rep! = Exception$$

`readfile` の仕様は以下の通りである．

MyRObject.readfile

MyInterface.readfile; *rep*! : *Report*

Ξ*MyObject*; Ξ*MyManager*

---

checkTable CREAD ⇒

$f? \in \mathrm{dom}\, serverFS$

$\Rightarrow serverFS(f?) = out!$

$f? \notin \mathrm{dom}\, serverFS \Rightarrow "" = out!$

$rep! = OK$

¬*checkTable CREAD* ⇒

$rep! = Exception$

4.1.3 クラス: `MyManager`

ファイルの読み出しの制限のみを緩和したセキュリティマネージャーである．クラスは，

```
class MyManager
   extends RMISecurityManager
```

であり，

```
public synchronized void
   checkRead(String file)
public synchronized void
   checkWrite(String file)
```

のメソッドをもつが，これらは直接アプリケーション中では実行されず，セキュリティ制限の緩急の設定に利用される．

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．

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

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する．真の場合，実行可能とする．

MyManager

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

このセキュリティマネージャーでは書き込みのみ真にする．

InitMyManager

MyManager′

$checkTable' \; CREAD = \mathrm{true}$

$checkTable' \; CWRITE = \mathrm{false}$

⋮

4.1.4 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，

クライアントプログラムの動作するマシン (クライアントマシンとする) から，リモートメドソッド呼び出しによって，サーバープログラムの動作するマシン (サーバーマシンとする) 中のファイルを読むことができるが，書くことはできないプログラム．

図 2: アプリケーションの非形式的仕様

```
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．

Naming

lookupTable : *String* ⇸ *Remote*

`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*

Δ*Naming*

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

$\mathrm{dom}\, lookupTable \cup \{n?\} = \mathrm{dom}\, lookupTable'$

`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

n? : *URL*; *r*! : *Remote*; *rep*! : *Report*

Ξ*Naming*

$n? \in \mathrm{dom}\, lookupTable \Rightarrow$

$lookupTable \; n? = r!; \; rep! = OK$

$n? \notin \mathrm{dom}\, lookupTable \Rightarrow$

$\theta Naming = \theta Naming'; \; rep! = Exception$

4.2 アプリケーションの仕様と実装

上記の部品仕様をもとに図 2 のような非形式仕様をもつプログラムを実装する．この仕様では，書き込みができないように指定されているため，書き込みメソッドを実装しない方法も考えられる．しかし，ここでは `MyManager` によって読み出しに関する制約のみを緩和することでその実装を行うことにする．実装の例は付録 2 の通りである．

このアプリケーションは，ファイルシステムへの書き込みはできないことが期待され，§4.1 の部品仕様をもとにすると

$$MyRObject.write \mid serverFS \; f? \neq s?$$
$$\vdash serverFS = serverFS'$$

である性質が推論できる．

しかし，実装したプログラムが上記の性質を満たさない場合がある．RMI を行うためのスタブ，スケルトンそしてリモードオブジェクトの実装そのものなどが，ローカルファイルシステム上からロードされた場合，セキュリティマネージャーによる制限は課されないのである．これは，部品を含めたプログラムのソースからでは判断できず，実行時に，必要なクラスがどのように配置されているかの情報を用いて，はじめて判断することができる．

4.3 実行環境を考慮した仕様化

そこで，リモートオブジェクトとそのスケルトンのロード先を実行環境 *LoadMap* として仕様化し，関連するメソッドの仕様に取り込むことにより，部品の仕様を実際の動作にあったものとする．

動的ロードを行うためにクラスファイルを配置する場所を表す型として以下を定義する．

$$LoadPlace ::= Local \mid Network$$

そして実行環境は以下のように定義する．

LoadMap

$$impl, skel : LoadPlace$$

`MyRObject` は §4.2 と変わらない．

MyRObject

$$serverFS : String \pfun String$$

ある実行環境とセキュリティ制限下のみで `MyRObject.writefile` は意味があるため，その条件を事前条件として記述する．

pre.MyRObject.writefile

$$MyRObject;\ MyManager;\ LoadMap$$

$$impl = skel = Local \lor$$
$$(impl = Network \lor skel = Network)$$
$$\land chakTable\ CWRITE$$

事前条件をパスした場合の事後条件のみ記述する．

MyRObject.writefile

$$MyInterface.writefile$$
$$\Delta MyRObject;\ \Xi MyManager$$

$$serverFS'\ f? = s?$$
$$\mathrm{dom}\ serverFS \cup \{f?\} = \mathrm{dom}\ serverFS'$$

これによって，

$$MyRObject.write$$
$$\mid skel = Local \land serverFS\ f? \neq s?$$
$$\vdash serverFS \neq serverFS'$$

や，

$$MyRObject.write$$
$$\mid skel = Network \land serverFS\ f? \neq s?$$
$$\vdash serverFS = serverFS'$$

となり，実際のプログラムの動作と仕様の上での推論が一致する．

5 議論

図 1 にも示すように，リモートメソッド呼び出しは，クライアントからスタブ，スケルトンなどを介して実際のリモートオブジェクトへとメソッドが委譲される形で実装されている．それをそのままに仕様化すると，

$$\text{スタブ} \gg \text{スケルトン} \fatsemi \text{実装}$$

のようになる．しかし，このような仕様ではプログラマにとっては知る必要のない内部情報まで記述することになり，部品を利用するための説明としての仕様には必要のない記述である．よって，本研究では直接にリモートメソッドを呼び出す形で仕様化した．

スタブ，スケルトン，実装などがプログラム実行時にどこからロードされるかによって，セキュリティマネージャーの制約が変化し，それによって，実行結果も変化する．よって，この部分については部品を利用するために必要な情報として仕様に加えた．特に，実装とスケルトンの両方がローカルファイルシステムからロードされた場合には，セキュリティマネージャーは無視されるが，そうでない場合にはセキュリティマネージャーの制約を受けることになる．この条件をリモードオブジェクトのメソッドの事前条件として記述した．

本研究での記述法では，Meyer の「契約による設計」のコンセプトをもとにした．しかし，Eiffel や iContract[4] などのように，実行時の表明のチェックではなく，部品の ‘マニュアル’ として表明を記述した．本研究の仕様化は，プログラマがその部品を利用する場合に必要な知識を提供することを目的としているため，実際の実装に即した表明とはなっていない．例えば，図 1 にあるようなアーキテクチャに完全に即してリモードメソッドを仕様化していない．しかし，そうすることによって，部品の実装やアーキテクチャに忠実な仕様よりも簡潔な記述になっていると思われる．

Meyer は文献 [3, p.400] で表明の記述に記述能力の高い言語，Z や VDM やそのオブジェクト指向拡張を利用するか否かについて議論している．Meyer の Eiffel では，Eiffel 自身を多少拡張した言語で表明を記述する．この方法は，実行時のチェックも可能となり，分かりやすさ，および表現力としても，それほど劣るものではないと Meyer は主張している．本研究では，曖昧性のない ‘マニュアル’ としての側面を重視するために，既存の形式的仕様記述言語を利用した．

文献 [7, §4.1, p.7] では,「契約による設計」の欠点として，事後条件に手続き呼び出しを利用できないことを指摘し，よって仕様の煩雑さを押さえることができないと指摘している．たしかに文献 [3, p.402 下] において，表明内で呼ばれた関数自体の表明のチェックから容易に無

限ループに陥る危険を認めている．しかし，表明は常に表明が添付されているルーチンよりも高い信頼性をもっているべきであり，それゆえその中にある関数呼び出しも同様に高信頼性であるべきであるとしている．よって，表明内の関数呼び出しの表明は評価すべきでないとしている [3]．‘マニュアル’ としての部品の仕様では，この条件をさらに緩和しても良いと思われる．

文献 [7] の筆者は，‘アブストラクトステートメント (グレーボックススタイル)’ という仕様記述法を提案しており，これは従来の事前条件/事後条件を記述する「契約による設計」よりも，ソフトウェア部品の構成と文書化に適しているとしている [8]．この記法では，Refnement Calculus[9] を理論基盤とし，サービスの操作的な記述を行うため，その記述は ‘疑似コード’ に類似している．また，文献 [10] では Meyer の契約とは異なる考え方の ‘契約’ による仕様化を行っている．ここでの ‘契約’ は，プログラム内で同時につかう部品を関係付ける概念であり，部品の振る舞いに関する依存関係をもとに関係付けを行っている．これらの方法では通常の事前条件/事後条件型記述では困難であった，他の関数や手続きを呼び出す問題をある程度解決しているように思われる．一方，本研究では，Z 言語のスキーマ取り込みを利用することで，ある程度の仕様間の依存関係などを記述することが可能となったが，その記述が理論的に不備がないかは今後検討する必要がある．

6 おわりに

本研究では，Java RMI クラス群の仕様化を通し，実行環境に依存する部分の仕様化の実例を示した．これによって，ソフトウェア部品を利用する際の実行環境に依存して，その部品の振舞いが変るような場合でも，部品利用者は，その変化を仕様から得ることができ，誤解のないプログラミングやテストを行うことが期待できる．記述の方針としては「契約による設計」を利用し，記述言語は Z を用いた．

本研究で提案する仕様は，部品利用者が，その部品を利用することが適切であるか判断し，誤用があればそれを回避するための ‘マニュアル’ とならなければならない．そのためには，記述が曖昧でないことに加え，簡潔である必要がある.「契約による設計」の考え方は部品の適切性を判断する場合に適した記述スタイルだと考えられる．そして，Z 記法によって曖昧性のない記述を行うことができる．今後は記述が簡潔となるための方法を考案する必要がある．

記述対象は，RMI の利用方法の中でも，もっとも単純な Java アプリケーションによるプログラミングを対象とした．しかし，実際には RMI はアプレットなどと連携して利用される場合が多いため，アプレットなどを含めた形式化を行う必要がある．

Java の言語構成子は，通常のクラス，`System` や `Naming` などのクラスメソッドを利用するタイプのクラス，インタフェースなどがある．これらを仕様記述の上で，どのように記述し分けるかを再検討する必要がある．さらに現時点では，例外処理を事前条件違反として仕様化しているが，Java では故意の例外発生により機能を構成している場合があるため，この点も再検討が必要であると思われる．

仕様記述言語も，記述する対象がオブジェクト指向型部品であるため，その記述により適したもの，例えば Object-Z や Z++などの Z のオブジェクト指向拡張言語や，ツールサポートなども考慮し B 法の導入なども検討すべきであると思われる．

謝辞

本研究は，文部省重点領域研究「発展機構を備えたソフトウェア構成原理の研究」(課題番号 09245104) の援助の下に実施された．本研究のための調査にあたり適切な助言をいただいた堀 雅和氏および，例題の作成に協力していただいた高瀬 泰宏氏に記して謝意を表す．

参考文献

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

[2] Jean-Marc Jezequel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *COMPUTER*, Vol. 30, No. 1, pp. 129–130, Jan. 1997.

[3] Bertrand Meyer. *Object-oriented software construction, 2nd edition*. Prentice Hall, 1997.

[4] Reto Kramer. iContract - The Java™ Design by Contract™ Tool. In *TOOLS USA'98*, 199-8. `http://www.tools.com/usa_98/tools_usa98.html`.

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

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

[7] Martin Büchi and Wolfgang Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Science, Presented at the Workshop on Foundations of Object-Oriented Programming, Zuerich, September 1997, 1997. `http://www.abo.fi/~mbuechi/publications/GreyBoxes.html`.

[8] Martin Buchi and Emil Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In Jan Bosch and Stuart Mitchell (Eds.), editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, pp. 332–337, Finland, Jun. 1997. Springer.

[9] R. J. R. Back and Joackim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer Verlag, 1998.

[10] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *OOPSLA/ECOOP'90 Proceedings*, pp. 169–180, Oct. 1990.

付録 1: 部品のソース

```
 1 import java.rmi.*; // インタフェース MyInterface
 2 public interface MyInterface extends Remote{
 3   public String readfile(String file) throws RemoteException;
 4   public boolean writefile(String file, String str) throws RemoteException;
 5 }
```

```
 1 import java.rmi.*; // リモートオブジェクト定義のクラス: MyRObject
 2 import java.rmi.server.*;
 3 import java.net.*;
 4 public class MyRObject extends UnicastRemoteObject implements MyInterface{
 5   public MyRObject() throws RemoteException{ super(); }
 6   public String readfile(String s) throws RemoteException{
 7     return new String(SimpleIO.readfile(s));
 8   }
 9   public boolean writefile(String f, String s) throws RemoteException{
10     return SimpleIO.writefile(f, s);
11   }
12 }
```

クラス SimpleIO のソースは省略.

```
 1 import java.rmi.*; // クラス: MyManager
 2 import java.io.*;
 3 class MyManager extends RMISecurityManager{
 4   public synchronized void checkRead(String file){}
 5   public synchronized void checkWrite(String file){
 6     super.checkWrite(file); // disallow write to file.
 7   }
 8 }
```

付録 2: アプリケーションのソース

```
 1 import java.rmi.*; // サーバープログラムの本体
 2 public class MyServer{
 3   public static void main(String args[]){
 4     System.setSecurityManager(new MyManager());
 5     try{
 6       MyRObject h=new MyRObject();
 7       Naming.rebind(args[0], h);
 8       System.out.println(args[0]+" is ready");
 9     }catch(Exception e){
10       System.out.println(e+"");
11     }
12   }
13 }
```

```
 1 import java.rmi.*; // クライアントプログラムの本体
 2 public class MyClient {
 3   public static void main(String args[]){
 4     System.setSecurityManager(new MyManager());
 5     try{
 6       MyInterface h=(MyInterface) Naming.lookup(args[0]);
 7       if(args[1].equals("read") && args.length>2){
 8         System.out.println("("+h.readfile(args[2])+")");
 9       }else if(args[1].equals("write") && args.length>3){
10         h.writefile(args[2], args[3]);
11       }else{
12         System.out.println("do nothing.");
13       }
14     }catch(Exception e){
15       System.out.println("Exception in main: "+e);
16     }
17   }
18 }
```