ダウンロード可能なソフトウェア部品の仕様化法

海谷 治彦　　海尻 賢二

信州大学 工学部 情報工学科
〒 380-8553 長野市 若里 4-17-1

電話: 026-269-5469　　ファクシミリ: 026-269-5495
kaiya@cs.shinshu-u.ac.jp kaijiri@cs.shinshu-u.ac.jp

あらまし　本稿では，ネットワーク越しの他のマシンからダウンロードされ連結・実行が可能なソフトウェア部品に関する仕様化技法を提案する．システム外部からロードした部品は，十分に信頼おけるわけではないため，システム内での振る舞いや機能をある程度制限する必要がある．このような考え方はサンドボックス・セキュリティモデルに基づいている．Java における RMI やアプレットを利用する際は，このようなセキュリティ機構を非明示的に利用する場合が多いため，利用者は RMI やアプレットなどの部品の能力や制限を見誤る場合があり，それによって部品の誤用をする恐れもある．本稿ではダウンロード可能な部品のセキュリティに関する性質に注目した仕様を追加記述することを提案する．それによって，部品利用者が正確に部品の能力や制限を理解すること支援することが可能となる．

キーワード　仕様化技術，Java1，サンドボックス・セキュリティモデル，再利用

Specifying Runtime Functionality of Downloadable Components under the Sandbox Model

Haruhiko Kaiya　　Kenji Kaijiri

Department of Information Engineering, Faculty of Engineering
Shinshu University

4-17-1 Wakasato, Nagano City, 380-8553, JAPAN
Phone: +81-26-269-5469　　Fax: +81-26-269-5495
Email: kaiya@acm.org　　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. Such components enable a system evolve itself in runtime. 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. Such assumption is based on the sandbox security model. 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.*

Key words　*Specification techniques, Java1, Sandbox Security Model, Reuse*

1 Introduction

近年，ネットワーク上に偏在する他のマシンからソフトウェア部品をダウンロードし，その部品をシステムの実行中に連結し利用することが可能となった．このような種類のソフトウェア部品はモバイルコードと呼ばれる [15].

通常，ソフトウェア部品の仕様は自然言語の文書として提供される．しかし，そのような文書は長い割に，曖昧かつ冗長な場合があり，部品利用者は部品の不適切な利用をする恐れがある．例えば，RMI[13] の仕様書はおよそ 90 ページの文書からなり，それを完全に読むことは困難である．しかし，小さな予備資料や参考書などでは RMI 部品を十分に理解することは難しい．

一方，仕様化のための技術の 1 つとして形式的手法が古くから提案されている．ソフトウェア部品の形式的仕様を記述することは，かなりの労力を必要とするが，記述された仕様の繰り返しの利用によって部品の適切な再利用が可能となるなら，十分に費用に見合った効果があったと考えることができる [9].

ソフトウェア部品の再利用のための形式的もしくは半形式的な仕様記述技法は既に数多く提案されている．ほとんどのソフトウェア部品は，その手続き，関数に関するシグニチャの仕様を持っている．そして一部のシステムは意味的な部分に関する仕様さえも記述されている場合がある．例えば，プログラミング言語 eiffel [8] では，伝統的な事前，事後条件と普遍命題がコードとともに記述されている．同様の機構を Java に付加しようとする試みもある [6]．シグニチャ，事前，事後条件および普遍命題などはオブジェクト指向ソフトウェアシステムの仕様化にも適している．

しかし，これらの仕様化技法のみではモバイルコードの性質を仕様化することができない．なぜならば，モバイルコードはシステム内において常に信頼のおける要素ではないからである．また，モバイルコードを含むシステムを動作させる場合，システム外の様々な要因を整備する必要がある．例えば，RMI や Jini などの分散システムを動作させるには，いくつかの外部サービスをセットアップし，ネットワーク接続を確保し，適切な位置に適切なコードを配置するなどの準備が必要である．つまり，モバイルコードは外部環境やセキュリティモデルの仕様化無しにはその振る舞いや機能を完全には仕様化できないのである．

最も有名なセキュリティモデルの 1 つに，Java1 で採用されたサンドボックス・セキュリティモデルがある．このモデル下では，ローカルシステム内に配置されたコードはシステムの重要な資源 (ファイルシステム等) に自由にアクセスすることができるのに対して，ネットワーク越しにダウンロードされた部品はそのアクセスが一部に制限されている [14]．他にもいくつかのモデルが提案されており [11]，Java2(JDK1.2 以降) ではサンドボックスモデルを核とした複合モデルが採用されている．

本稿では，サンドボックス・セキュリティモデル下におけるモバイルコードの仕様化技法を提案する．特に，自立的に移動を行わなわず，システムによって受動的にダウンロードされる種類のモバイルコードを扱うことにする．本稿の仕様を用いることで，ダウンロード可能な部品の利用者は，そのような部品の利用環境を適切に設定し，適切な再利用を行うことが可能となる．本稿では Java 言語の部品であるクラスを対象に議論を進める．

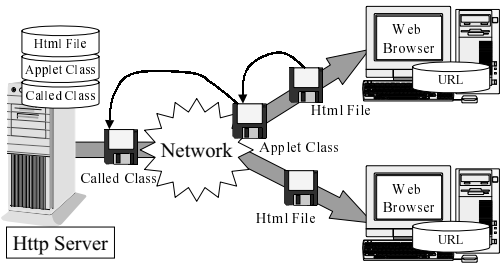

図 1: アプレットの概要

2 節では，ダウンロード可能な部品の性質や問題点を紹介する．そして，3 節において，前述の問題点を克服するためにどのような事項を仕様として追加記述すべきかを述べる．4 節では，ここでの仕様記述が，ダウンロード可能な部品の適切な利用を促進することを例示する．そして，最後にまとめと今後の課題を述べる．

2 ダウンロード可能な部品の性質と問題点

本節では，ダウンロード可能な部品の例を示しながら，その機構と問題点を整理する．本節での例題は Java 言語のクラスを部品の例として用いている．よって，部品と ‘クラス’ は同じ意味で用いる．

2.1 ダウンロード可能な部品とは何か?

アプレットは最も有名はダウンロード可能なソフトウェア部品の 1 つである．図 1 にアプレットの典型的な振る舞いを示す．本稿の図内では，ディスケットの図でダウンロード可能な部品を表し，円筒の図でローカルなシステム内に配置された部品を表すことにする．アプレットが動作するウエブ・ブラウザは最初にアプレットのブートストラップとなる html ファイルの URL のみを知っている．その html ファイルがロードされた後，そこに指定されているクラスがブラウザ内の JVM にロードされ，さらにそのクラスが利用している他のクラスも順次ロードされる．それゆえ，ブラウザの利用者は AWT グラフィックキットや Java Beans などの部品を汎用ブラウザ上で容易に利用することができる．一般にロードされたクラスは十分に信頼できるとは限らないため，ブラウザが動作するクライアントシステムの計算機資源に対してのアクセスに制限が設けられている．例えば，ファイルシステムの読み書き，ネットワーク接続の確立などは一般に禁止されている．

RMI (Remote Method Invocation)[13] は Java 言語に

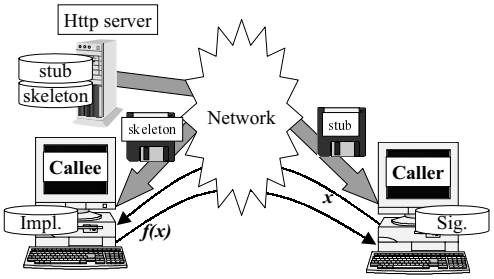

図 2: RMI の概要

おけるオブジェクト指向 RPC[2] とみなすことができる．一般に RPC や RMI を確立する場合には，あるマシン内の *caller* (RPC クライアント) プログラムと別のマシン内の *callee* (RPC サーバー) プログラムの間で，リモート手続きもしくはメソッドの引数や返り値を転送するためのソフトウェア部品が必要がある．RPC では，通常このような部品は caller と callee に静的に連結される場合が多い．それに対して RMI では，このような部品を静的に連結する必要がないだけではなく，クライアント，サーバーそれれぞれのローカルシステムに事前に配置する必要さえないのである．RMI では caller 側のこのような部品を *stub* と呼び，callee 側のを *skeleton* と呼ぶ．図 2 に RMI の概要を示す．caller は，Java 言語の interface として記述されたリモートメソッド (図中の *f(x)*) のシグニチャ情報と，stub が配置されているネットワーク上の位置を知っている．callee は，skeleton の位置を知っており，リモードメソッド *f(x)* が具体的に実装されている．そして，caller がリモードメソッドを実行する際に，http などのサービスを介して stub と skeleton はそれぞれに動的にダウンロードされ，caller と callee それぞれに動的に連結され，プログラム間の通信が可能となる．stub と skeleton も前述のアプレット同様に十分に信頼でき安全な部品とは限らないため，計算機システムの資源へのアクセスが制限されている．アプレットと異なり，RMI では，この計算機資源へのアクセス制限を `SecurityManager` クラスを利用して，アプリケーションプログラマが明示的に指定する必要がある．よって，場合によっては stub と skeleton の振る舞いを制限しないようなプログラムも記述することが可能である．

図 2 では，stub と skeleton のダウンロードをサービスするマシン (http server) と，RMI の Callee は別のマシンとして記述されている．しかし，実際の RMI では，この 2 つのマシンは同一でなければならない．RMI と同様の機能を提供可能な Jini[1] システムではこの制限は無くなっている．

2.2 サンドボックス・セキュリティモデル

2.1 節で述べたアップレットや RMI の振る舞いは，サンドボックス・セキュリティモデルに従っている．サンドボックスの本質は「ローカルシステム内のコードはそのシステム内の重要な資源 (ファイルシステム等) に対して完全にアクセスできるのに対して，ダウンロードされたコードは，ある制限 (砂箱) 内にある一部の資源のみへのアクセスが許されている」である [14]．このモデルは Java1 言語システムで広く使われている．

サンドボックス・モデルはたった 2 種類の空間 (完全なアクセス可能なローカルコードのための空間とサンドボックス内にあるダウンロードされたコードのための空間) を提供しているのみなので，セキュリティ制限に関する表現の自由度が充分であるとは言えない．Java2 システムでは，多様なセキュリティ空間を記述することが可能となり，サンドボックス・モデルでの制限が取り除かれている．

しかしながら，サンドボックス・モデルは Java2 でもダウンロードされたコードにとっての基本的なモデルとなっている．Java2 での拡張されたモデルも多様なセキュリティ制限をもつ多数のサンドボックスを許したモデルとみなすことができる．さらに，プログラマが明示的に指定しなければ，Java2 においてもサンドボックス・モデルがデフォルト値として利用されている [4]．それゆえ，本稿では，Java1 におけるサンドボックス・モデルにのみ焦点を絞る．

2.3 ユーザー定義可能なクラスローダー

Java におけるソフトウェア部品は Java 仮想マシン (JVM) 上で実行される [16]．本稿では，実行時におけるそれら部品の機能に注目する．JVM ではユーザーがクラスローダーを独自に定義することができるため，アプリケーションプログラマがダウンロード可能な部品の振る舞いを明示的に管理することが可能となる．クラスローダーとは，Java においてダイナミック・ロードを可能とするための機構である [7]．Java1 では 2 種類のクラスローダーが利用可能である．1 つはシステム・クラスローダーであり，もう 1 つはユーザー定義可能なクラスローダーである [10]．ユーザー定義可能なクラスローダーを利用すると，ロードされる部品の振る舞いを制限することが可能である．よって，同じ部品であっても異なるローダーによってロードされた場合，その実行時の振る舞いは異なる場合がある．また，1 つの Java アプリケーションは同時に複数のクラスローダーを利用することが可能である．

Java1 のアプリケーションでは，最初に実行されるクラスはシステム・クラスローダーによってロードされる．システム・クラスローダーは，事前に設定された環境変数をもとに，実行されるマシンのローカルファイルシステムからクラスをロードする．そして，最初のクラスから直接に利用されるクラスも全てシステム・クラスローダーによってロードされる [13]．ユーザー定義可能なクラスローダーは `ClassLoader` クラスのサブクラスとして実装され，通常のクラスと同様に利用される．図 3 にその例を示す．図中で定義・利用されているクラスローダー `DLoader` は，インタフェース `Runnable` を実装した `args[0].class` という名前のクラスをネットワークストリームからロー

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  public class DLoader extends ClassLoader{
 5      // several definitions are omitted .....
 6
 7      protected Class load-
Class(String name, boolean res)
 8      throws ClassNotFoundException{
 9          Class clazz = null;
10
11          try{ return findSystemClass(name); }
12          catch(ClassNotFoundException e1){}
13          catch(NoClassDefFoundError e2){}
14
15          clazz=findLoadedClass(name);
16          if(clazz != null){ return clazz; }
17
18          clazz=findClass(name);
19          if(clazz == null ){
20            throw new ClassNotFoundException(name);
21          }
22          return clazz;
23      }
24
25      private synchronized Class find-
Class(String name){
26          // finding byte code from the net-
work resource
27          // and define class.
28          // .....
29                  return de-
fineClass(name, data, 0, total);
30          // .....
31      }
32
33      // several definitions are omitted .....
34
35      public static void main(String args[]){
36          try{
37              DLoader loader=new DLoader(new URL(args[0]));
38              Class cc=loader.loadClass(args[1]);
39              Runnable cmd=(Runnable)cc.newInstance();
40              cmd.run();
41          }catch(Throwable e){
42              e.printStackTrace();
43          }
44      }
45  }
```

図 3: 簡単なユーザー定義可能なクラスの例

ドする．このローダーによってロードされたクラスは特別な制限がされていないため，そのインスタンスはローカルシステム内からロードされたクラスのインスタインスと同様に自由に振る舞うことが可能である．例えば，パスワードファイルを第三者のマシンに転送することも可能である．

我々がアプレットや RMI を利用する際には非明示的に特定のユーザー定義可能なクラスローダーを利用しているため，クラスローダーの動作等に気は払うことは少ない．

あるクラスローダーのセキュリティ制限は以下のステップで決定される．

1. クラスローダーが `SecurityManager` を考慮するか否かを判定する．例えば，図 3 のクラスローダーは考慮しないように設計されているが，RMI やアプレットのローダーは考慮するように設計されている．
2. `SecurityManager` 内に列挙されているセキュリティ制限すべき項目の内容を調べる．`SecurityManager` クラスは，およそ 30 個のセキュリティに関するチェック項目に該当するメソッドを持つ．例えば，ファイルシステムへのアクセスの可否やネットワークコネクションの作成の可否などである [13]．ユーザーは `SecurityManager` のサブクラスとして独自の `SecurityManager` を実装することができ，これによって，それぞれの項目の可否を独自に決定することができる．例えば，RMI で用いられる `RMISecurityManager` では，ほどんとの項目が否となっているため，ほとんどのシステム内の資源へアクセスすることができない．

そして，上記のようなローダーでロードされた部品の実行時の機能は以下のように決定される．

- クラスローダーが `SecurityManager` を考慮するように設計されているにもかかわらず，`SecurityManager` が設定されていない場合，クラスローダー自身を実行することができない．よって，そのローダーでクラスをロードすることはできない．
- ローダーが `SecurityManager` を考慮しないように設計されている場合，ローダーは実行され，それによって他のクラスもロードされ，実行される．
- Java の標準 API 内のシステム資源を操作するメソッドは，`SecurityManager` が定義されている場合，その中で関連する項目を参照するように設計されている．よって，ロードされたクラスがシステム資源を API を介して利用しようとする場合，そのクラスの機能は `SecurityManager` によって制限されることになる．

  よって，ユーザー定義可能なクラスローダーが `SecurityManager` を必要としないように設計されていた場合，たとえ `SecurityManager` が設定されていたとしても，そこでの制限は全く無効である．

Java2 におけるセキュリティモデルは Java1 に比べ拡張・一般化されているが，上記のセキュリティ制限機構の多くは Java2 でも継承されている．

2.4 ダウンロード可能な部品における問題点

上述のように，Java では柔軟ではあるが複雑な部品のロード機構を備えている．結果として，クラスロード機構とセキュリティシステムの深い理解なしでは，部品の利用者であるアプリケーションプログラマは，部品の能力や制限を見誤る可能性がある．特に，ローカルシステム外からダウンロードされた部品を利用する場合は，このような危険性は高まる．ここで，ダウンロード可能な部品を利用する際の問題点をまとめる．

2.4.1 ローダー選択問題

前述のように，Java での部品は，その部品をロードするクラスローダーの違いによって，その振る舞いを変える場合がある．言い換えれば，クラスローダーはそのローダーによってロードされるクラスの振る舞いを制限しているといえる．それゆえに，部品利用者は，クラスローダーとロードされる部品の仕様を合わせて知っておく必要がある．しかし，それにはいくつかの

問題点がある．

- アプリケーションプログラマは通常，クラスローダーについて関心をはらっていない．
- 定義ローダーと呼ばれる実際にロードするクラスを定義するローダーは動的に決定される．定義ローダーの決定要因は主に以下の 2 点である．
  - 初期化ローダー内の選択ロジック
  - ネットワーク上の部品の元となるバイトコードの配置

部品 C が `L.loadClass()` の結果である場合，L を C の初期化ローダーと呼ぶ．そして，部品 C が `L.defineClass()` の結果である場合，L を C の定義ローダーと呼ぶ [7].

初期化ローダー内の選択ロジックは定義ローダーの選択順序を表現している．例えば，図 3 での `DLoader` では以下の順序で定義ローダーを選択する．

1. システムクラスローダー (11-13 行),
2. ローダー内のキャッシュ (15-16 行),
3. `findClass` を通して利用されるこのクラス自身の `defineClass` メソッド (18 行).

結果として，`DLoader` では，ローカルシステム内に配置されている部品が優先的に利用される．RMI やアプレットのクラスローダーも同様のロジックを持つ．しかし，リモートシステムに配置されている部品を優先的にロードするようなクラスローダーを作成することも可能である．

2.4.2 配置問題

クラスの元となるバイトコードの配置は定義ローダーの選択判断に大きく影響する．例えば，`RMIClassLoader` を用いてネットワーク越しの他のマシンに配置されている部品をロードしようとしたとしても，ローカルシステム内に同じ名前の部品が配置されていた場合，`RMIClassLoader` は定義ローダーとして利用されることは無く，システムクラスローダーが定義ローダーとして利用される．結果として，セキュリティマネージャーはロードされたクラスに対しては何の影響も与えない．それゆえに，セキュリティマネージャーが機能しているか否かを知るためには，部品の配置についても知っておかなければならない．

2.4.3 入れ子ロードの問題

図 5 のようにあるクラス *A* から利用されるクラス *B* では，*A* の定義ローダーが継承されるのではなく，初期化ローダーが継承される．よって，*B* は常に *A* と同じローダーで定義されるわけではないため，明示的にローダーを変更しない場合でも，セキュリティ制限が変わる可能性がある．部品利用者は，初期化ローダー内に記述される定義ローダーの選択ロジックにも注意を払いながら，ロードされる部品の振る舞いを理解する必要がある．

3 追加的な仕様記述

2 節で述べたダウンロード可能な部品の性質や問題点を仕様化するためには，以下に示すような項目を部品仕様とともに記述する必要がある．

1. **個々の部品におけるセキュリティポリシィ**: サンドボックス・モデルで述べるようにダウンロード可能な部品は自由な動作を許してよいほど信頼性があり安全なものではないため，どのような振る舞いが許され，また許されていないかを仕様化する必要がある．
2. **動作する部品の出生地**: 部品を実体化するためのバイトコードの所在によって，その部品の信頼性をある程度判断することができる．よって，部品の出生地にあたるバイトコードの所在についても仕様として記述すべきである．
3. **部品が実行されるマシンの位置**: 前述の出生地は，実際に部品が実体化し実行されるマシンとの相対位置によって意味を持つ場合がある．例えば，部品の出生地と実行マシンが一致している場合，サンドボックス・モデルでは，完全に信頼できる部品であるとみなす．よって，部品が実行される位置も仕様として記述すべきである．
4. **バイトコードの配置**: 前述の定義ローダー決定法で述べたように，アクセス可能なマシン，ネットワークストリーム等にバイトコードがどのように配置さているかによって，部品の実行時の振る舞いは変化する．よって，ネットワーク上のどのマシンにどのバイトコードが配置されているかも仕様化する必要がある．
5. **定義ローダーの選択ロジック**: Java では初期化ローダー内に実際にクラスを定義するローダーを選ぶロジックを記述している．また，定義ローダーではなくこのロジックが次に呼び出されるクラスに継承される．よって，定義ローダーの選択ロジックも部品仕様として記述する必要がある．

本稿では，ダウンロード可能な部品を，以下に示すような部品に関する状態と，その状態を変化させる関数として仕様化する．まず，以下のよう状態を記述する．

- 実行時の部品を実体化するのに必要なバイトコードと，それらが配置されているマシン等への写像．この写像はすべての JVM に共有される．
- サンドボックス内の部品からアクセス可能なシステム資源へのアクセス制限の状態を示すフラグ．このフラグは個々の JVM 毎に定義される．
- 部品のコードを保持しているバイトコードを探す場合のパス．このパスは個々のクラスローダー毎に定義される．
- 部品が実体化されるマシンの識別子．
- 実体化された部品とそのソースであるバイトコードの対応．
- 個々の部品固有の属性．

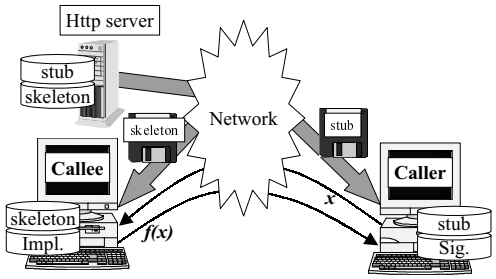

図 4: stub および skeleton がローカルおよびリモートシステムの双方に配置されている場合の RMI の実行

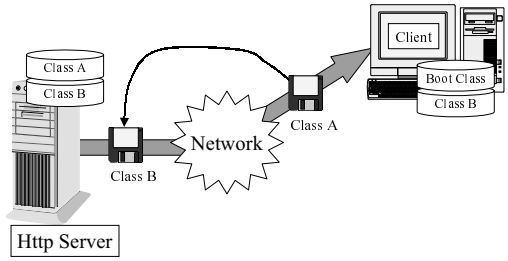

図 5: クラスの入れ子上の呼び出し

そして，以下のような関数を定義する．

- 前述のフラグ，写像，パスや対応を設定・変更するための関数．
- 部品内の個々のメソッドの意味を示す事前，事後条件．伝統的な仕様とは異なり，事前，事後条件は部品固有の属性だけでなく，前述のフラグ，写像，パスや対応も考慮して記述する．

4 例題

本節では，例題を通して 3 節で提案した仕様がダウンロード可能な部品の性質を理解するのに役立つことを示す．本稿では Z 記法 [12] を用いて同仕様を記述する．

4.1 クラッキング・コードを含むスタブ: 反例

4.1.1 背景

パスワードファイルを盗むようなクラッキングを行うコードが stub 内に混入していることがわかっている RMI を行わなければならない状況だとする．このようなクラッキングが実行されないために，ローカルファイルシステムに対するアクセスを禁止するような `SecurityManager` を作成し，それを実行時に使用することにした．さらに，stub 内のクラッキングコードがさらに増加することがないように，現段階での stub をローカルシステムに配置した．図 4 に上記の状況を図示する．

4.1.2 仕様

以下では，意図的に 3 節におけるフラグのみを考慮して仕様を記述することで，記述内容の不足のために誤った推論を行ってしまうことを示す．

最初に，システム資源とそれぞれの資源のセキュリティ制限をスキーマ *SysRes* として仕様化する．Java でのセキュリティ制限は `SetSecurityManager` メソッドによって 1 回のみ変更可能なである．これを *SetLimit* スキーマで仕様化する．

SysRes

$$res : R \pfun Bool;\ limit : \mathbb{P}\, R$$

$$limit \subseteq \mathrm{dom}\, res;\ \forall x : limit \bullet (x, \mathrm{false}) \in res$$

SetLimit

$$\Delta SysRes;\ l? : \mathbb{P}\, R$$

$$limit \neq \emptyset \Rightarrow l? = limit'$$

R はシステム内の資源の集合を示す型であり，*SysRes* スキーマはこのマシンでのセキュリティ設定の状態を表現している．

クラッキングするコードを含むメソッド自身 (*Func*) と，クラッキングするコード (*Crack*) はそれぞれ以下のように仕様化する．

Func

$$x?, y! : \mathbb{Z}$$

$$y! = f(x?)$$

Crack

$$pas! : R;\ \Xi SysRes$$

$$(pas!, \mathrm{true}) \in res$$

RMI で呼び出すメソッド全体の仕様は以下のようになる．

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

4.1.3 推論

前述の仕様下において，以下のスキーマの真偽を推論する．

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

このスキーマは「セキュリティ制限を行ったにもかかわらず，クラッキングされてしまう.」ということを述べている．

前述の仕様をもとに形式推論すると，*res* が関数であるにもかかわらず，$(pas!, \mathrm{true}) \in res$ と $(pas!, \mathrm{false}) \in res$ が同時に充足されるため，このスキーマは矛盾している．それゆえ，クラッキングは実際には行われないことが推論できる．

4.1.4 議論

前述の仕様では部品の出生地に違いによってセキュリティマネージャーによるセキュリティ制限が行われるか否かが変化することに言及していない．結果として，上記の推論は事実を反映していなものとなっている．すなわち，この状況下では実際にはクラッキングは行われてしまうのである．この問題点を次節で解決する．

4.2 クラッキング・コードを含むスタブ: 解決編

4.2.1 仕様

部品の元となるバイトコードの配置位置に相当する型，*Loc* を新たに導入する．そして，バイトコードを示す型 *ByteCode* も導入する．これらを利用してネットワーク上のバイトコードの配置を以下のように定義する．これは，3 節での配置写像に相当する．

$$| \mathit{deploy} : \mathit{Loc} \pfun \mathbb{P}\, \mathit{ByteCode}$$

次に *SysRes* スキーマに部品が実体化し動作する位置を示す属性，*here* を追加する．これは 3 節の識別子に相当する．さらに部品とその出生地，およびその定義ローダー選択ロジック (初期化ローダー) の対応を示す *Class* スキーマも導入する．

SysRes

$$\mathit{res} : \mathit{R} \pfun \mathit{Bool};\ \mathit{limit} : \mathbb{P}\, \mathit{R};\ \mathit{here} : \mathit{Loc}$$

$$\mathit{limit} \subseteq \mathrm{dom}\, \mathit{res}$$
$$\forall x : \mathit{limit} \bullet (x, \mathrm{false}) \in \mathit{res}$$
$$\mathit{here} \in \mathrm{dom}\, \mathit{deploy}$$

Class

$$\mathit{birth} : \mathit{Loc};\ \mathit{byte} : \mathit{ByteCode}$$
$$\mathit{lslctr} : \mathrm{seq}\, \mathit{Loc}$$

$$\mathit{birth} \in \mathrm{ran}\, \mathit{lslctr}$$
$$\mathit{birth} \in \mathrm{dom}\, \mathit{deploy}$$

Class スキーマ内の属性 *birth* は以下のようなスキーマによって設定される．

SetLoader

$$\mathit{sl}?;\ \mathrm{seq}\, \mathit{Loc};\ \Delta \mathit{Class}$$

$$\mathit{lslctr}' = \mathit{sl}?$$
$$\forall x, y : \mathbb{N} \bullet \mathit{byte} \in \mathit{deploy}\, \mathit{lslctr}'\, x \land$$
$$x \in \mathrm{dom}\, \mathit{lslctr}' \land \mathit{lslctr}'\, y = \mathit{birth}' \Rightarrow y \leq x$$

述部の 2 行目以降は多少複雑な記述になっているが，列 *lslctr* 内で最初に *byte* が見つかった場所がその出生地 (*birth*) となることを述べている．

スキーマ *Crack* は出生地の情報を参照するように以下のように修正する．

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

4.2.2 推論

$$\mathit{deploy} = \{(\mathit{here}, \{\mathit{byte}, \cdots\}),$$
$$(\mathit{there}, \{\mathit{byte}, \cdots\}) \cdots\}$$

の環境化において，以下のスキーマは充足可能となる．

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

図 4 との対応をとると，*here* が ‘Caller’ に相当し，*there* が ‘Http server’，*byte* が ‘stub’ に相当する．結果として，このような状況下では，例えセキュリティマネージャーによるセキュリティ制限が行われていたとしても，クラッキングは行われてしまう危険があることを示すことができる．そこで，クラッキングを防止するためには，例えば，*sl*?(初期化ローダーのロジック) や *deploy*(バイトコードの配置) の値を変更する必要がある．

4.3 定義ローダーの変化

4.3.1 背景

クラス *A* はあるクラスローダーによってロードされ，*A* から別のクラス *B* が別途呼び出されたとする．クラス *A, B* ともにリモートシステムに配置されてており，ダウンロード可能な部品であったとする．このような状況に加えて，リモートシステム上のクラス *B* と同じインタフェースを実装した新しいクラス *B* をローカルシステムに新たに配置したとする．現状でのバイトコード等の配置は図 5 に示す状態になっている．我々は後からローカルシステムに配置した新しいクラス *B* を利用したいという要件を持っている．

しかしながら，初期化ローダーのロジックがリモートシステム上のバイトコードを優先的にロードするように設定されていたため，上記の要件，新しく配置した *B* を利用することが達成されないことを以下に示す．

4.3.2 仕様と推論

ここでも前節での仕様 *SysRes*, *Func* そして *SetLoader* を修正して利用する．以下のスキーマは上述の要件を表現している．

$$\mathit{SetLimt} \fatsemi ($$
$$(\mathit{SetLoaderA} \land \Xi \mathit{SysRes} \fatsemi$$
$$\mathit{FuncA} \land \Xi \mathit{SysRes} \land \Xi \mathit{ClassA} \fatsemi$$
$$\mathit{CallA} \land \Xi \mathit{SysRes}) \setminus \mathit{ClassA}$$
$$>>$$
$$(\mathit{SetLoaderB} \land \Xi \mathit{SysRes} \fatsemi$$
$$\mathit{FuncB} \land \Xi \mathit{SysRes} \land \Xi \mathit{ClassB}) \setminus \mathit{ClassB}$$
$$) \mid \mathit{sl}? = \langle \mathit{there}, \mathit{here} \rangle \land \mathit{birthb} = \mathit{here}$$

そして，図 5 に示す実行環境は以下の式で表現できる．

$$\mathit{deploy} = \{(\mathit{here}, \{\mathit{byteb}, \cdots\}),$$
$$(\mathit{there}, \{\mathit{bytea}, \mathit{byteb}, \cdots\}) \cdots\}$$

上述の *CallA*，*CalssA*, *ClassB* 等は以下のように定義される．

$$CallA \mathrel{\widehat=} [slb! : \operatorname{seq} Loc;\ \Xi ClassA \mid slb! = lslctra']$$

$$ClassA \mathrel{\widehat=} Class[birtha/birth, bytea/byte, lslctra/lslctr]$$

$$SetLoaderB \mathrel{\widehat=} SetLoader[slb?/sl?, ClassB/Class]$$

4.3.3 議論

上述の仮定下では，*birthb* = *there* および *birthb* = *here* が同時に成り立つため，要件を記述した最初の式は矛盾する．よって，新たに配置したクラス *B* を利用したいという前述の要件は現状では満たされることはない．仕様レベルにおいて，バイトコードの配置と，初期化ローダーのロジックを考慮してクラスを仕様化したためにこのような正しい推論を行うことができた．要件を満たすには，*deploy* もしくは *sl*? を変更する必要がある．

5 おわりに

本稿では，Java の例題として，ダウンロード可能なソフトウェア部品の機能をどのように仕様化すべきかを議論した．ここでの仕様化スタイルは従来のスタイルと同じではあるが，ダウンロード可能な部品に関してはセキュリティに関係する仕様も部品の仕様とともに仕様化すべきであること明らかにした．JVM の形式化はすでに提案されており [5]，Java におけるセキュリティ問題も数多く報告されている [3]．しかし，これらの研究はダウンロード可能なソフトウェア部品の再利用を促進するという点を目標とはしていない．

セキュリティポリシィを柔軟に記述できればできるほど，ダウンロード可能な部品に関するシステムの振る舞いを詳細に指定することが可能となる．しかし，同時にシステムの振る舞いを把握するのが困難になることもある．例えば，個々の部品が異なるセキュリティポリシィを持った上でシステムとして動作する場合，実行時におけるシステムの機能を認識することは容易ではない．Java2 では，`Permission` と `AccessController` クラスの導入により，より柔軟性のあるセキュリティポリシィを記述することが可能となった [14]．さらに，ダウンロード可能な部品のセキュリティに関して，サンドボックス・モデルだけではなく，コード著名や proof-carrying コードなどの多彩な技術が利用可能となりつつある [11]．しかし，これらが実行時にどのように働き，システムの機能や振る舞いに影響を与えるかを認識することはますます困難になると思われる．それゆえに，ダウンロード可能な部品に関する仕様の記述法も，それにあわせて拡張してゆく必要がある．

REFERENCES

[1] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The* Jini™ *Specification* . Addison-Wesley, first edition, Jun. 1999.

[2] J. Bloomer. *Power Programming with RPC*. O'Reilly & Associates, Inc, Sep. 1992.

[3] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[4] L. Gong. Secure Java Class Loading. *IEEE Internet Computing*, 2(6):56–61, Nov. and Dec. 1998.

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

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

[7] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, Oct. 1998.

[8] B. Meyer. *Object-oriented software construction*. Prentice Hall, second edition, 1997.

[9] B. Meyer. The Next Software Breakthrough. *COMPUTER*, 30(7):113–114, Jul. 1997. IEEE/CS.

[10] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, first edition, Mar. 1997.

[11] A. D. Rubin and J. Daniel E. Geer. Mobile Code Security. *IEEE Internet Computing*, 2(6):30–34, Nov. and Dec. 1998.

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

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

[14] Sun Microsystems, Inc. *Java Security Architecture (JDK1.2)*, Oct. 1998. Version 1.0.

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

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