# Java虚拟机动态类加载的形式化模型*)

左天军[1] 朱智林[1] 韩俊刚[2] 陈 平[1]
(西安电子科技大学软件工程研究所 西安710071)[1] (西安邮电学院计算机系 西安710061)[2]

**摘 要** Java虚拟机支持一种功能很强的动态加载类的机制,它具有惰性加载、类型安全连接、用户自定义加载策略、以及动态名字空间等特性。但是,在Java的早期实现(JDK 1.0和1.1)中,这种机制包含了一种称为类型欺骗的严重设计错误。尽管JDK1.2通过引入一种类加载约束策略修正了这个错误,但是由动态加载引起的其它形式的类型欺骗仍然存在于JDK1.2和1.3中。本文详细讨论了与动态类加载相关的类型欺骗问题,提出了一个严格定义Java虚拟机操作语义和静态语义的形式化模型。其中,操作语义描述了类加载约束策略、字段及方法解析算法等类加载的主要特性;静态语义采用类型规修正了JDK1.2和1.3中的类型欺骗。

**关键词** 动态类加载,类型安全,类型系统,操作语义,多加载器,Java虚拟机

## A Formal Model for Dynamic Class Loading in the Java Virtual Machine

ZUO Tian-Jun[1] ZHU Zhi-Lin[1] HAN Jun-Gang[2] CHEN Pin[1]
(Institute of Software Engineering, Xidian University, Xi'an 710071)[1]
(Department of Computer, Xi'an Institute of Post and Telecoms, Xi'an 710061)[2]

**Abstract** The Java Virtual Machine(JVM) supports a novel and powerful class loading mechanism which incorporates all of the following features: lazy loading, type-safety linkage, user-definable class loading policy and multiple namespaces. However, that class loading methanism contained a seerious type-spoofing bug in earlier implementations(JDK 1.0 and 1.1), which leads to type safety violations. Although JDK 1.2 introduces a class loading constraint scheme to fix the bug, subtle type spoofing related to class loaders still exists in JDK 1.2 and 1.3. We develop a formal model to specity the operational semantics and static semantics of the Java virtual machine, rigorously. In the model, the operational semantics describes the main features of class loading such as class loading constraints scheme, field and method resolutions etc. The static semantics uses typing rules to fix the type-spoofing bug in JDK 1.2 and 1.3.

**Keywords** Dynamic class loading, Type safety, Type system, Operational semantics, Multiple loaders, Java virtual machine

## 1 引言

Java支持一种动态加载程序组件的技术,即在运行时安装程序的类文件。Java的动态类加载具有惰性加载、类型安全连接、用户自定义加载策略以及动态名字空间等特性。Java最早在JDK1.0中实现动态类加载功能,当时的用途是为了在HotJava浏览中动态安装小程序(applet)。随后,动态加载被广泛应用于其它程序组件的设计当中,如服务方组件(servlet)[1],JavaBeans[2]。但是,一个称为类型欺骗的严重设计错误存在于Java开发平台的早期版本(JDK1.0和1.1)中,这种错误能够完全突破Java的类型检查。尽管Java从JDK1.2起引入了一种类加载约束机制来修正这个设计错误,但是在JDK1.2和1.3中仍然存在着由动态加载引起的其它形式的类型欺骗。

虽然动态加载在Java平台中具有非常重要的地位,但是在相关文献中对于它的描述却很不充分。为此,本文提出一种描述Java虚拟机的形式化模型,它不仅能够修正JDK1.2和1.3中的类型欺骗问题,而且可以严格定义Java虚拟机的语义。本文第2节简要介绍Java动态加载机制的基本概念;第3节给出Java类型欺骗的实例并分析其产生的原因;第4节详细讨论提出的形式化模型;第5节介绍相关研究工作;最后总结全文并指出进一步研究的方向。

## 2 基本概念

Java动态加载机制的核心概念是类加载器,它是Java系统类java.lang.ClassLoader或其子类的实例。类加载器操作的对象是类文件,这是Java编译器为程序产生的一种与平台无关的中间表示。指定一个类名后,类加载器会寻找相应的类文件并将其安装到JVM环境中,同时为该类创建一个相应的Class对象。通常,JVM从本地文件系统中加载类文件。例如,在UNIX系统中,JVM从CLASSPATH环境变量定义的目录中加载类。同时,JVM也允许应用程序通过实现ClassLoader的子类定义其加载策略。

java.lang.ClassLoader是一个抽象类,其中与加载直接相关的方法主要有:*loadClass*(),*findClass*()以及*defineClass*()。其中,*defineClass*()是*classLoader*类中的一个final方法,不能在应用程序的子类中重置。当加载一个类时,类加载器通过*loadclass*()的参数发出加载某个类文件的请求,然后*load-*

*)Supported by thd National Natural Science Foundation of China under Grant No. 90207015(国家自然科学基金)。左天军 博士,主要研究领域为网络安全、形式化验证、软件工程技术;朱智林 博士;韩俊刚 教授,博士生导师;陈 平 教授,博士生导师。

*Class*()既可以调用 *defineClass*()完成类加载并定义一个 *Class* 对象,也可以调用其它加载器的 *loadClass*()委托加载类文件。这样对于被加载的类而言,发出加载请求的类加载器与完成类加载的加载器可能并不相同。

**定义1**(初始化加载器) 如果采用 $L.loadClass()$ 加载一个类 $C$ 并返回一个 *Class* 对象,则称 $L$ 为类 $C$ 的初始化加载器,或称 $L$ 初始化类 $C$ 的加载。

**定义2**(定义加载器) 如果采用 $L.defineClass()$ 加载一个类 $C$ 并返回一个 *Class* 对象,则称 $L$ 为类 $C$ 的定义加载器,或称 $L$ 定义类 $C$。

在 JDK1.2的以前版本中,用户可以通过实现 *loadClass*()定义其加载策略。但是,从 JDK1.2起,Java2建议重置 *findClass*()定义加载策略。

类加载器的另一重要功能是为程序定义动态名字空间。一个类在编译时,其类型是由类名确定的;一个类在运行时,其类型是由类名和类的定义加载器共同确定的。本文采用 $\langle N,L_1\rangle^{L2}$ 表示一个运行时类,其类名为 $N$,定义加载器为 $L_1$,初始化加载器为 $L_2$。当不需要考虑初始化加载器时记为 $\langle N,L_1\rangle$;当不需要考虑定义加载器时记为 $N^{L2}$。

更详细的 Java 动态类加载的介绍可以参考 Java 虚拟机规范[4]。

## 3 类型欺骗

本节介绍两种类型欺骗,一种是由 Saraswat 最早提出的类型欺骗,这个问题在 JDK1.2以后的版本中得到了更正;另一种是 JDK1.2和1.3中存在的类型欺骗。在一个类型欺骗问题中,通常包含一个欺骗类、一个被欺骗类以及一个中间类。

·类型欺骗实例1

图1给出了一个 Saraswat 类型欺骗实例。在这个实例中,被欺骗类是 $class\langle R,L_2\rangle$,欺骗类是 $class\langle R,L_1\rangle$,中间类是 $class\langle RT,L_1\rangle$ 和 $class\langle RR,L_2\rangle$。

```
class ⟨RT,L1⟩{
    private R r;
    void test(){
        RR rr=new RR();
        r=rr.getR();
        r.k=r.k+1;//type spoofing
    }
}
class⟨R,L1⟩{
    public intk;
}
```

```
class⟨RR,L2⟩{
    R getR(){
return new R();
    }
}
class⟨R,L2⟩{
Private object k;
}
```

图1 Saraswat 类型欺骗

现在分析类型欺骗的过程。当程序运行到 $\langle RT,L_1\rangle$ 的 $RR\ rr$=new $RR()$ 语句时,由于在类 $RT$ 中引用了类 $RR$,根据规范 Java 会选用当前类($RT$)的定义加载器($L_1$)发出加载 $RR$ 的请求。然后,$L_1$ 将 $RR$ 的加载委托给 $L_2$ 并由 $L_2$ 定义 $RR$。当程序运行到 $\langle RT,L_1\rangle$ 的 $r=rr.getR()$ 语句时,由于引用 *getR*(),Java 通过方法解析算法在 $\langle RR,L_2\rangle$ 中查找并调用方法 *getR*()。当程序运行到 *getR*()的 return *new* $R()$ 语句时,由于引用了类 $R$,Java 采用类 $RR$ 的定义加载器 $L_2$ 加载并创建一个类 $\langle R,L_2\rangle$ 的对象。于是,程序通过 $rr.getR()$ 返回一个 $\langle R,L_2\rangle$ 对象。当运行到 $r.k=r.k+1$ 时,Java 通过 $L_1$ 加载类 $R$ 并检查 $R$ 是否具有属性 $k$ 以及 $k$ 是否能被公共访问。因为 $L_1$ 加载并定义的是欺骗类 $\langle R,L_1\rangle$,所以 Java 认为 $r$ 可以访问 $k$。但是,实际上 $r$ 指向的是被欺骗类 $\langle R,L_2\rangle$ 的对象。这样,程序就可以通过 $r$ 访问 $\langle R,L_2\rangle$ 中的私有属性 $k$,从根本上突破了 Java 的类型系统。

为了解决这种类型欺骗,JDK1.2实现了一种类加载约束机制。Java 会在以下几种情况中引入类加载约束:(1)属性引用约束。如果 $\langle C,L_1\rangle$ 引用了 $\langle D,L_2\rangle$ 中的一个静态类型为 $T$ 的属性,那么就在集合中增加约束 $T^{L1}=T^{L2}$;(2)方法引用约束。如果 $\langle C,L_1\rangle$ 引用了 $\langle D,L_2\rangle$ 中的方法 $T_0$ method $(T_1,\cdots,T_n)$,那么增加约束 $T_0^{L1}=T_0^{L2},T_1^{L1}=T_1^{L2},\cdots,T_n^{L1}=T_n^{L2}$;(3)方法重置约束。如果 $\langle C,L_1\rangle$ 重置了 $\langle D,L_2\rangle$ 中的一个 $T_0$ method $(T_1,\cdots,T_n)$ 方法,那么增加约束 $T_0^{L1}=T_0^{L2},T_1^{L1}=T_1^{L2},\cdots,T_n^{L1}=T_n^{L2}$。

为了维护以上约束,每加载一个类时,Java 就检查这个类是否不违反所有约束条件,若违反其中任意一个约束则加载失败;每增加一个新的约束条件时,Java 就检查此约束是否能满足所有已加载的类,若不能满足则约束增加失败。

考虑如何利用类加载约束机制修正以上的类型欺骗。当程序引用 $rr.getR()$ 语句时,根据方法引用约束,Java 会增加约束 $R^{L1}=R^{L2}$。当程序通过 $r.k=r.k+1$ 加载 $\langle R,L_1\rangle$ 时,由于 $R^{L1}(=\langle R,L_1\rangle)\neq R^{L2}(\langle R,L_2\rangle)$,因此 Java 产生程序连接异常错误。

·类型欺骗实例2

图2的类型欺骗不能由 JDK1.2的类加载约束解决。在这个问题当中,被欺骗类是 $class\langle Sup,L_2\rangle$,$class\langle Sub1,L_2\rangle$ 和 $class\langle Sub2,L_2\rangle$;欺骗类是 $class\langle Sup,L_1\rangle$;中间类是 $class\langle Foo,L_1\rangle$。

```
class⟨Sup,L2⟩{private int f;}
class⟨Sub1,L2⟩extends Sup{}
class⟨Sub2,L2⟩extends Sup{}
class⟨Sup,L1⟩{public int f;}
class ⟨Foo,L1⟩{
    static public void foo(){
        Sub1 s1=new Sub1();
        Sub2 s2=new sub2();
        Sups;
        if(s1!=null)s=s1;
        else s=s2;
        System.out.println("this is private"+s.f);//type spoofing
    }
}
```

图2 JDK1.2和1.3中的类型欺骗

当程序运行到 System.out.println("Thisis private attribute"+s.f)语句时,尽管程序期望 $s$ 引用的是类型 $\langle Sup,L_1\rangle$ 中的属性 $f$,但 $s$ 实际引用的却是 $\langle Sup,L_2\rangle$ 中的属性 $f$。于是,程序就通过 $s.f$ 访问了 $\langle Sup,L_2\rangle$ 中的私有属性。

与 Saraswat 类型欺骗不同,这种形式的类型欺骗是由 Java 的字节码验证引起的。由于字节码经常通过网络传输,为了保证字节码的正确性,在解释字节码程序之前,Java 虚拟机采用一个字节码检验程序静态检查字节码能否满足规定的约束条件,这样程序在运行时就不需要进行动态类型检查。本质上说,Java 虚拟机规范[4]中的字节码检验程序是一种应用于字节码指令的抽象解释器。它操作的是字节码指令的类型而不是其具体数值。例如,对于一个整数加指令 iadd,运行时它将操作数栈最上面的两个整数相加并将结果放到操作数栈的顶部。字节码检验程序抽象解释这条指令时,它检查操作数栈的顶部是否存放着两个整数类型。如果是,它就将一个整数类型放到操作数栈的顶部。否则,它就报告程序异常。字节码检验算法的详细讨论见第4.9节。

在字节码验证过程中,字节码检验程序会为访问字段和方法的字节码指令(如 getfield 和 invokevitual)创建类型约束

条件。对于 getfield 指令，会创建形如 $C_1^L \prec^* C_2^L$ 的约束。其中，$\prec^*$ 代表动态类之间的子类型关系，$L$ 代表当前被验证类的定义加载器，$C_1$ 代表抽象解释 getfield 指令前操作数栈顶部保存的类型，$C_2$ 为该字段的符号引用类型。例如，如果将 class $\langle Foo, L_1\rangle$ 中的 if…else…分支语句替换成 $s=s1$，那么在字节码验证过程中，由于 $s=s1$ 会将类型 $Sub1$ 传播到 System.out.println("This is private attribute"+s.f) 语句的 $s.f$，并且 $s.f$ 对应于编译后的 getfield 指令，因此字节码检验程序会为这个指令创建约束 $Sub1^{L1} \prec^* Sup^{L1}$。这样 Java 采用 $L_1$ 加载 $Sup$ 时会因违反这个约束出现一个异常错误。

现在考虑 class $\langle Foo, L_1\rangle$ 中的 if…else…分支语句。在这种情况下，不仅 $s=s1$ 会将类型 $Sub1$ 传播到 $s.f$，$s=s2$ 也会将类型 $Sub2$ 传播到 $s.f$。于是根据 Java 虚拟机规范，字节码检验程序在抽象解释 $s.f$（即 getfield）前需要先合并类型 $Sub1$ 和 $Sub2$。因为久类型合并就是计算所合并类的最小公共超类，所以在这个例子中合并的结果就得到类型 $Sup$。于是，Java 就为 class $\langle Foo, L_1\rangle$ 中的 $s.f$ 产生一个约束 $Sup^{L1} \prec^* Sup^{L1}$。由于这个约束是一个永真式，所以就出现了上述的类型欺骗。

## 4 形式化模型

我们提出一个严格定义 Java 虚拟机操作语义和静态语义的形式化模型。其中操作语义描述了 Java2的类加载约束机制、字段和方法解析算法；静态语义定义了 Java 虚拟机的字节码验证，其中的子类型约束解决了 JDK1.2和1.3中的类型欺骗问题。在详细介绍形式化模型之前，先给出模型中使用的一些形式化定义。

### 4.1 形式化定义

**定义3**（基本类型）

```
datatype CLASS={
  name:Classname;
  supername:Classname;
  cl:Loc;
  mthdlst:Mthod list
  fidlst:Field list
}
datatype Method={
  name:Methodname;
  mthdDesc:Chassname list;
  cls:CLASS;
  inst:Instruction list
}
datatype Field={
  name:Fieldname;
  fidDesc:Classnamet;
  cls:CLASS;
}
datatype Instruction={
  invokevirtual of Classname list×Methodname×Classname
  |getfield of Classname×Fieldname×Classname;
  |areturn
  }
```

定义3定义了模型中用到的一些基本类型。其中，CLASS、Method 和 Field 代表记录类型。当要提取一个记录对象的某个属性时，可以表示为"对象.属性名"。例如，一个 CLASS 类型的对象 $c$ 由类名 name、超类名 supername、定义加载器 cl、方法列表 mthdlst 以及字段列表 fldlst 所组成。$c.cl$ 表示对象 $c$ 的定义加载器。Instruction 类型定义了模型操作的字节码指令集。尽管这个集合只包含了 invokevirtual、getfield 和 areturn 这几个指令，但是这些指令对于研究动态加载的主要特性是充分的。

定义3～7定义了模型中的一些常用谓词。$\prec$ 定义了动态类型之间的直接子类型关系；非直接子类型关系 $\prec^*$ 定义为 $\prec$ 的传递闭包。$\subset_{cl}$ 定义了类名之间的偏序关系。如果两个类名同由加载器 $cl$ 初始化加载并且加载得到的类之间具有子类型关系，那么这两个类名之间就存在偏序关系 $\subset_{cl}$。$:_{cl}$ 定义了指定对象类型的规则。其中，抽象函数 $c$：Loc→CLASS 用于返回一个对象的动态类。$:_{cl}$ 还用于指定一个对象列表的类型。等介关系 $\approx_n$ 用于定义类加载约束。

**定义4** （子类型）

$c \prec c' \Leftrightarrow c=c' \vee c.\text{supername}^{c.cl}=c'$

$c \prec^* c' \Leftrightarrow \exists\ c_1, c_2, \cdots, c_n.\ c \prec c_1 \wedge c_1 \prec c_2 \wedge \cdots \wedge c_n \prec c'$

**定义5** （子类型约束）

$n \subset_{cl} n' \Leftrightarrow n=n' \vee n^{cl} \prec^* n'^{cl}$

**定义6** （对象造型）

$v :_{cl} n \Leftrightarrow C(v)=n'^{cl} \wedge n' \subset_{cl} n$

$vlst :_{cl} nlst \Leftrightarrow length(vlst)=length(nlst) \wedge \forall\ k.\ vlst[k] :_{cl} nlst[k]$

**定义7** （类加载约束）

$l \approx_n l' \Leftrightarrow n^l=n^{l'}$

算法1～2定义了字段解析和方法解析。fldResolv：CLASS×Classname×Fieldname×Classname→CLASS 定义了如何从一个类 $c$ 中解析一个符号引用 $objcn$ 中的某个字段。其中，字段由描述符 $cn$ 和字段名 $fnm$ 所指定。字段解析的过程就是在类层次结构中递归查找指定字段的过程。算法中的抽象谓词 $\mathscr{F}$ 用于确定一个类中是否包含了某个字段。

mthdResolv：CLASS×Classname list×Mthodname×Classname→CLASS 定义了方法解析算法。其中方法描述符 desc 是列表 $desc[0], desc[1], \cdots, desc[n]$ 的缩写。抽象谓词 $\mathscr{M}$ 用于确定一个类中是否包含了某个方法。

**算法1** （字段解析）

```
fldResolv(c,cn,fnm,objcn)=
  if 𝓕(objcn^{c,cl},cn,fnm)then objcn^{c,cl}
  else if objcn^{c,cl}.supername≠null
       then fldResolv(objcn^{c,cl},cn,fnm,objcn^{c,cl}.supername)
       else throw exception
```

**算法2** （方法解析）

```
mthdResolv(c,desc,mn,objcn)=
  if 𝓜(objcn^{c,cl},desc,mn)then objcn^{c,cl}
  eslf if object^{c,cl}.supernameo≠null
    then mthdResovle(objcn^{c,cl},desc,mn,objcn^{c,cl} supername)
      else throw exception
```

### 4.2 操作语义

在我们的模型中，Java 虚拟机的操作语义由状态转移关系 $\langle m, f, s, pc\rangle \rightarrow \langle m', f', s', pc'\rangle$ 所确定。其中，$m$ 表示当前状态；$f$ 清示局部变量列表；$s$ 表示操作数栈列表；$pc$ 表示程序计数器。

图3中的(op_invokevirtual)规则定义了指令 invokevirtual 的操作语义。invokevirtual 指令用于一个实例方法的调用。根据 Java 虚拟机规范[4]，这个调用包含三个过程：(1)方法解析。操作语义中的前提 mthdResolv(…)=c 定义了这个过程。(2)方法重置。这个过程由前提 mthdSel(…)=c′ 所确定。(3)调用此方法。

invokevirtual 语义中的前提 $\rho \wedge \rho'$ 定义了方法解析和方法重置过程中的类加载约束。

```
m[pc]=invokevirtual(desc[0],desc[1],…,desc[n],mn,objcn)
mthdResolv(m.cls,desc[0],desc[1],…,desc[n],mn,objcn)=c
mthdSel(c(o),desc[0],desc[1],…,desc[n],mn,(C(o).name)=c′
ρ∧ρ′
```

$c(v)=k^{c',cl}$
$c(o)=\prec^{*}c$
---
$\langle m,arg[n]\cdots::arg[1]::O::s,f,pc\rangle\rightarrow\langle m,v::s,f,pc+1\rangle$
where
$\rho=m.chs.cl\approx_{desc[0]}c.cl\wedge m.cls.cl\approx_{desc[1]}c.cl\cdots\wedge m.cls.cl\approx_{desc[n]}c.cl$
$\rho'=c.cl=_{desc[0]}c'.cl\wedge c.cl\approx_{desc[1]}c'.cl\cdots\wedge c.cl\approx_{desc[n]}c'.cl$
$m'[pc']=areturn$
$m[pc]=invokevirtual(desc[0],desc[1],\cdots,desc[n],mn,objcn)$
$mthdResolv(m.cls,desc[0],desc[1],\cdots,desc[n],mn,objcn)=c$
$mthdSel(C(o),desc[0],desc[1],\cdots desc[n],mn,(C(o).name)=c'$
$m'.cls=c'$
$\rho\wedge\rho'$
---
$\langle m',v::s',f',pc'\rangle::\langle m,s,f,pc+1\rangle\rightarrow\langle m,v::s,f,pc+1\rangle$
$m[pc]=getfield(cn,fnm,objcn)$
$fldResolv(m.cls,cn,fnm,objcn)=c$
$m.cls.cl\approx_{cn}c.cl$
$readHp(o,cn,fnm)=v$
$C(v)\prec^{*}cn^{c.cl}$
$C(o)\prec^{*}c$
---
$\langle m,o::s,f,pc\rangle\rightarrow\langle m,v::s,f,pc+1\rangle$

图3 指令的操作语义

指令 areturn 的操作语义表示从当前活动记录返回一个对象到前一个活动记录中。(op_areturn)结论中的$\langle m',\cdots\rangle::\langle m,\cdots\rangle$给出了调用堆栈的情况，其中$\langle m',\cdots\rangle$代表当前的活动记录。

指令 getfield 用于读对象字段的值。前提条件 fldResolv$(\cdots)=c$ 表示在获取字段值之前要先进行字段析。$m.cls.cl\approx_{cn}c.cl$ 表示字段解析过程中产生的加载约束。抽象函数 readHp：Loc→Classname→Fieldname 返回对象指定字段的值，并且此字段值满足类型约束 $C(v)\prec^{*}cn^{c.cl}$。

invokevirtual 和 getfield 语义中前提 $C(o)\prec^{*}c$ 是一个保证类型安全的条件。分析第3节中的类型欺骗可以发现，如果程序不能满足这个条件，就会出现类型欺骗。

### 4.3 静态语义

如果一个程序能够满足断言 $F,S\vdash m$，那么这个程序的类型是正确的。$F,S\vdash m$ 定义为

$$F,S\vdash m\Leftrightarrow F,S,0\vdash m,F,S,1\vdash m,\cdots,F,S,length(m.inist)\vdash m$$

其中，$F$ 和 $S$ 分别代表一个从程序计数器到局部变量数组和操作数栈列表的部分映射。由于抽象解释操作的是类型，因此 $F_k$ 表示程序计数器为 $k$ 时局部变量数组中的静态类型；$S_k$ 表示程序计数器为 $k$ 时操作数栈列表中的静态类型。$F,S,k\vdash m$ 代表程序点 $k$ 处的局部断言，它严格定义了每条指令的静态语义。图4给出定义局部断言的规则。

$m[pc]=invokevirtual(desc[0],desc[1],\cdots,desc[n],mn,objcn)$
$S_i\subset_{m.cls.cl}desc[n]\cdots::desc[1]::objcn::\beta$
$Setof(n,S_i)=\{\tau_1,\cdots,\tau_n\}\Rightarrow\tau_1\subset_{m.cls.cl}objcn,,\cdots\tau_n\subset_{m.cls.cl}objcn$
$desc[0]::\beta\subset_{m.cls.cl}S_{i+1}$
$F_i\subset_{m.cls.cl}F_{i+1}$
---
$F,S,i\vdash m$
$m[pc]=getfield(cn,fnm,objcn)$
$S_i\subset_{m.cls.cl}objcn::\beta$
$Setof(0,S_i)=\{\tau_1,\cdots,\tau_n\}\Rightarrow\tau_1\subset_{m.cls.cl}objcn,\cdots,\tau_n\subset_{m.cls.cl}objcn$
$cn::\beta\subset_{m.cls.cl}S_{i+1}$
$F_i\subset_{m.cls.cl}F_{i+1}$
---
$F,S,i\vdash m$
$m[pc]=areturn$
$S_i\subset_{m.cls.cl}m.desc[0]::\beta$
$F,S,i\vdash m$

图4 指令的静态语义

前面的分析指出，JDK1.2和1.3中的类型欺骗是由字节码验证中的类型合并引起的。因此，我们为局部变量列表和操作数栈列表中的每个元素引入一个集合，这个集合用于保存参与合并的静态类型。抽象函数 setOf：unm→Classname set 用于返回列表中某个元素对应的这个集合。根据这个集合，字节码验证程序就能产生正确的子类型约束。

举例来说，对于图2中的类型欺骗，尽管 System.out.println("This is private attribute"+s.f)语句的 $s.f$ 需要合并传播来的 $Sub1$ 和 $Sub2$，但是字节码检验程序为参与合并的类型保存了一个集合$\{Sub1,Sub2\}$。根据这个集合，字节码检验程序为 $s.f$(即 getfield)创建约束 $Sub1\subset_{L1}Sup$ 和 $Sub2\subset_{L1}Sup$。这样 Java 采用 L1加载 $Sup$ 时会因违反这个约束给出一个异常错误。

## 5 相关工作

Saraswat[3]最先提出类型欺骗问题并给出了两种解决方案，一种是动态检查对象类型；另一种与文[4]的类加载约束非常相似。Dean[5]在 PVS 系统中研究了动态连接和静态类型检查之间的关系。Dean 的模型非常接近于 Java 的动态连接模型。Drossopoulou[6]提出了一个 Java 连接的抽象模型，研究了类型检查在加载、连接、字节码检验中的关系。Drossopoulou[7]提出了一个描述动态连接的非确定模型，它能够用统一的方法研究 Java 和 C# 两种不同的动态连接模型。文[6,7]是基于 Java 语言层的，因此更加适合 Java 程序的推理。Higuchi[8]为 JVM 字节码构造了一个类型系统。Higuchi 的类型系统是一种类似于 λ 演算的带有类型的项演算。Jensen[9]给出了 JVM 动态加载和连接的操作语义。但是，与 JVM 规范相比 Jessen 的方法存在着不准确之处。例如，Jensen 假设在一个子类的层次结构中，所有类是由同一个加载器定义的。这种假设是不正确的，因为 Java 平台没有这样的约束。

Qian[10]提出了一种描述 Java 动态加载的形式化模型。与我们工作的不同之处在于：(1)Qian 在字节码验证中没有考虑类型合并时的情况，因此不能修正 JDK1.2和1.3中的类型欺骗；(2)我们在类型规则中描述了局部变量和操作数栈在不同程序点应该满足的类型约束，Qian 没有在其模型中定义这种类似的约束；(3)我们定义了为对象指派类型的规则，因此我们就能按照传统的方式表达和证明模型的可靠性。Qian 通过定义可靠状态表达其模型的可靠性定理，显得不够直观。

Fong[11]认为动态加载、连接和字节码检验之间的耦合过于紧密，提出一种模块化结构研究此问题。

**结论** 本文介绍了 Java 动态类加载的概念，讨论了与动态加载相关的类型欺骗问题，提出了一个描述 Java 动态加载的形式化模型，它通过在字节码验证过程中增加类型约束来修正 JDK1.2和1.3中的类型欺骗。尽管本文的模型没有考虑 Java 的多线程、访问控制、数组、接口、包等特性，但这些特性对于分析 Java 动态加载的主要特性是没有多少影响的，在模型中增加这些特性是比较容易的。由于 Java 的安全性不仅局限于类型检查，因此，进一步的研究方向为在此模型的基础上研究 Java 的安全策略。

## 参考文献


1 JavaSoft, Sun Microsystems, Inc. Servlet, 1998. JDK1.2 documentation, available at http://java.sun.com/products/jdk/1.2/docs/ext/servlet

2 JavaSoft, Sun Microsystems, Inc. JavaBeans Components API for Java, 1997. JDK 1.1 documentation, available at http://java.sun.com/products/jdk/1.1/docs/guide/beans

3 Saraswat V. Java is not type-safe. Technical report, AT&T Rresearch, 1997. http://www.research.att.comp/vj/bug.html

4 Lindholm T, Yellin F. The Java™ Virtual Machine Specification-2nd edition. Addison-wesley, 1999

5 Dean D. The Security of Static Typing with Dynamic Linking. In

fourth ACM Conference on Computer and Communication Security, 1997
6 Drossopoulou S. Towards an abstract model of Java dynamic linking and verification. In: R. Harper, ed. TIC'00-Third Workshop on types in Compilation (Selected Papers), volume 2071 of Lecture Notes in Computer Science, Springer, 2001. 53～84
7 Drossopoulou S, Lagorio G, Eisenbach S. Flexible models for dynamic linking. In Pierpaolo Degano, editor, European Symposium on Programming 2003, volume 2618 of Lecture Notes in Computer Science, Springer, 2003. 38～53
8 Jensen T, Metayer D L, Thorn T. Security and Dynamic Loading in Java: A Formalisation. In: Proc. of the 1998 IEEE Intl. Conf. on Computer Languages, Chicago, inois, May 1998. 4～15
9 Higuchi T, Ohori A. Java bytecode as a typed term calculus. In: proc. of the conf. on Principles and practice of declarative programming, 2002
10 Qian Z, GoldBerg A, Coglio A. A formal specification of Java TM class loading. In: Proc. ACM Conf on Object-Oriented Programming, Systems, Languages, and Applications. ACM Press, 2000
11 Fong P W L, Cameron R D. Proof Linking: Modular Verification of Mobile Programs in the Presence of Lazy, Dynamic Linking. ACM Transactions on Software Engineering and Methodology, 2000, 9(4): 379～409

(上接第196页)

函数 *recursive-compute-alpha-set*($(\vec{l},\vec{s},D)$)来计算该符号化状态和其前驱的所有附加集合。

经过这样的优化后得到的可达性分析算法所维护的前驱-后继关系与由基本算法生成的关系是不同的。如果在两个状态 $S$ 和 $S'$ 之间有关于全局转换的前驱和后继关系，则这两者的关系是 $sp\delta(\vec{e})(S)\subseteq_v S'$，而不是 $sp\delta(\vec{e})(S)\subseteq S'$。由附加集合的计算方法可知，对于任何符号化状态 $(\vec{l},\vec{s},D)$，$\vec{s}$ 在元组 $(\vec{l},D)$ 上与附加集合中所有共享变量取值相兼容。

### 5.2 用增强算法进一步优化可达性分析

在图1中所示的带有双下划线的程序代码描述的是使用增强算法来进行优化的部分。如果在状态空间遍历的某一时刻已生成的所有状态个数超过阈值 $\theta$ 时，就可以尝试进一步扩充这些状态的附加集合使其能兼容更多的共享变量取值。

## 6 案例研究

以上寻找兼容的共享变量取值的算法及增强算法已用 C++语言实现并且在带有512M 主存 IBM thinkpad 笔记本上检验了几个工业案例。这些案例包括 Fischer 互斥协议(Fischer's mutual exclusion protocol)，带界限重新传输协议(the Bounded Retransmission Protocol[6])和 Bang&Olufsion 音频协议(the Bang&Olufsion audio protocol[7])。本文中的优化技术在检验 Fischer 互斥协议时并不产生效率的提高。但当算法用来检验另外两个例子时会节省很多存储空间，在实验结果上表现为可达性图中结点和边的个数的减少。

Bang&Olufsion 音频协议用来在单一总线上的音频/视频组件之间传输数据。协议能在必要时检测数据冲突并重新传输数据。当检验 Bang&Olufsion 音频协议时，经过初步优化的可达性分析算法(不含增强算法)生成了包括 36555 个节点和 40769 条边的可达性图。未经优化的算法产生包括 116064 个节点和 126599 条边的可达性图。

表1 带界限重新传输协议的结果

| 参数 | 未经优化的节点个数 | 未经优化的边的个数 | 优化后的节点个数 | 优化后的边的个数 |
|---|---|---|---|---|
| MAX=3 N=3 | 709 | 917 | 539 | 732 |
| MAX=3 N=2 | 519 | 681 | 410 | 551 |
| MAX=2 N=3 | 570 | 738 | 444 | 601 |

带界限重新传输协议(the Bounded Retransmission Protocol)通过不稳定的信道传输文件。这个协议有两个参数：$MAX$ 和 $N$。发送者最多可以重复传送数据 $MAX$ 次；在一个传送序列中最多有 $N$ 个数据段。表1显示了用不同参数检验该协议得到的结果数据(检验时不含增强算法)。而同时包含寻找兼容的共享变量取值的算法及增强算法的可达性分析在 $MAX=3$、$N=3$ 且 $\theta=200$ 时产生了包括 325 个节点和 448 条边的可达性图。也就是说，增强算法可以进一步优化可达性分析算法的空间效率。

**结论** 时间自动机网络在模型检验中可用于并行实时系统的建模。网络中的时间自动机通过同步信道或共享变量交互。如果两个状态的共享变量取值不同，那么这两个状态也不同。因此共享变量也是状态空间爆炸的原因之一。本文定义了在共享变量取值之间的兼容性关系，并提出了检测符号化状态中共享变量取值所能兼容的取值的算法以及进一步进行这种兼容性关系检测的增强算法。算法找到的共享变量取值以一种紧凑的数据结构来记录。案例研究表明这两种优化技术用在可达性分析上时能够显著地降低可达性分析对于存储空间的需求，因而这两者是其它处理时间假设和约束的优化技术的一个有用的补充。

## 参考文献

1 Alur R, Dill D L. A theory of timed automata. In Theoretical Computer Science, The preliminary version appears in Proc. 17th ICALP, 1990, LNCS 443, 1994, 126: 183～235
2 Daws C, Olivero A, Tripakis S, Yovine S. The tool Kronos. In DIMACS Workshop on Verification and Control of Hybrid Systems, LNCS 1066. Springer-Verlag, Oct. 1995
3 Behrmann G, David A, Larsen K G, et al. Uppaal - Present and Future. In: Proc. of the 40th IEEE Conf. on Decision and Control (CDC'2001), Orlando, Florida, USA, IEEE Computer Society Press, Dec. 2001
4 Bengtsson J, Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In Lecture Notes on Concurrency and Petri Nets, Edited by W. Reisig and G. Rozenberg, LNCS 3098, Springer-Verlag, 2004
5 Burch J R, Clarke E M, McMillan K L. Symbolic model checking: 1020 states and beyond. In Information and Computation, 1992, 98: 142～170
6 D'Argenio P R, Katoen J-P, Ruys T, Tretmans J. Modeling and Verifying a Bounded Retransmission Protocol. In: Proc. of COST 247, Intl. Workshop on Applied Formal Methods in System Design, Maribor, Slovenia, June, 1996. Also appeared as Technical Report CTIT 96-22, University of Twente, July 1996
7 Havelund K, Skou A, Larsen K G, Lund K. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using Uppaal. In: Proc. of the 18th IEEE Real-time Systems Symposium, San Francisco, California, USA, Dec. 1997. 2～13