Analysing the Java Package/Access Concepts in Isabelle/HOL [*]

Norbert Schirmer

Fakultät für Informatik, Technische Universität München
http://www.in.tum.de/~schirmer

Abstract. Java access modifiers and packages provide a mechanism to restrict the access to members and types as additional means of information hiding beyond the pure object oriented concept of classes. In this paper we clarify the semantics of access modifiers and packages by adding them to our formal model of Java in the theorem prover Isabelle/HOL. We analyse which properties we can rely on at runtime, provided that the static accessibility tests have been passed.

1 Introduction

The work presented in this paper is part of a comprehensive research effort aiming at formalising and verifying key aspects of the Java programming language. In particular we have a type system and an operational semantics (with a proof of type soundness) and an axiomatic semantics (with a proof of its equivalence to the operational semantics) for a large subset of Java [5]. All these formalisations and proofs have been carried out in the Isabelle/HOL system [4].

Access modifiers determine access restrictions and visibility of class and interface types and their members. Since safety and security properties of Java are based on the bare language itself, the access modifiers are the main means to protect data. During the effort to formally model the package/access concept some intrinsic problems became apparent. The Java Language Specification (JLS) [1] is very imprecise and ambiguous concerning the package/access concepts and the Java implementations do not exactly follow the specification. Although some of the problems have already been known for years (cf. BugParade, Bug IDs 1240831, 4094611 on [7]), we have discovered and reported to Sun two further inconsistencies (Bug IDs 4485402, 4493343). Information hiding (with packages) and reuse of implementations (with inheritance and overriding) are conflicting goals. In this paper we clarify the semantics and discuss the runtime properties of access modifiers. Since it is unclear from the JLS what the exact meaning of various relevant notions concerning the package/access concepts are, or even worse, what exactly the relevant notions are, we will introduce and clarify the following notions in this paper:

accessible-in: When is a class or interface accessible?

[*] This research is funded by the EU project *VerifiCard*

inheritable-in: When is a member inherited?
member-of: Which are the members of a class, including inheritance?
member-in: Which are the members contained in the class and its superclasses?
permits-acc-to: Which classes are permitted to access a member?
accessible-from: Which member accesses are statically valid?
dyn-accessible-from: Describes the properties of runtime member access.
$overrides_S$: Compile-time variant of overriding.
overrides: Runtime variant of overriding.

The package/access model presented in this paper is based on the JLS. In case of ambiguities we refer to the Java release of Sun (SDK 1.3.1). As far as we know, the present formalisation offers the most comprehensive and detailed model of the Java package/access concepts (for other approaches, see for example [6] or [2]). Inner classes are not yet part of our model.

2 Basic Definitions

In Java a program is a collection of interfaces and classes arranged in different packages. The aim of the package concept is to combine closely related classes (and interfaces) inside a single package and to offer privileged access between those classes. We model packages by qualifying all typenames for interfaces and classes with a packagename.

$$\textbf{record}\ \textit{qtname} = \textit{pid}::\textit{pname}\ \textit{tid}::\textit{tname}$$

With *pid* the package name is selected out of such a record, with *tid* the typename is selected. In Java the package names are hierarchically organised, but this internal structure only plays a role in the lookup process for a package. For accessibility concerns this hierarchy is irrelevant, so we do not model it here.

In our formalisation a Java program is a mapping from qualified typenames to the structures describing the corresponding classes and interfaces.

The access modifiers are described as enumeration:

$$\textbf{datatype}\ \textit{acc-modi} = \textit{Private} \mid \textit{Package} \mid \textit{Protected} \mid \textit{Public}$$

The nomenclature resembles the original keywords of Java, except for *Package*, which models the nameless default access modifier of Java (that is applied when no other modifier is given explicitly). We define an ordering on the access modifiers, from most restrictive to most liberal: $\textit{Private} < \textit{Package} < \textit{Protected} < \textit{Public}$. An access modifier is attached to every member (field, method) and to every class or interface. So there is an accessibility concept on the level of types and on the level of members.

3 Accessibility of Types

Accessibility of types is captured in a predicate $G \vdash T\ \textit{accessible-in}\ P$ stating that in the context of a program G the type T is accessible in package P.

T	$G \vdash T\ \mathit{accessible\text{-}in}\ P$
PrimT	True
$\mathit{Iface}\ I$	$\mathit{pid}\ I = P \lor \mathit{is\text{-}public}\ G\ I$
$\mathit{Class}\ C$	$\mathit{pid}\ C = P \lor \mathit{is\text{-}public}\ G\ C$
$\mathit{Array}\ \mathit{elemT}$	$G \vdash \mathit{elemT}\ \mathit{accessible\text{-}in}\ P$

Primitive types like `int` or `string` are accessible in all packages. If an interface or class has the modifier *Public* then it is accessible in any package. Otherwise it is only accessible inside the same package. An array type is accessible in a package if the element type is accessible in this package.

4 Accessibility of Members

Accessibility of members is more involved than accessibility of types. First we have to clarify what the members of a class are.

$$\frac{\begin{array}{c} G \vdash \mathit{mbr}\ m\ \mathit{declared\text{-}in}\ C \\ \mathit{declclass}\ m = C \end{array}}{G \vdash m\ \mathit{member\text{-}of}\ C}\ (\textsc{Immediate}) \qquad \frac{\begin{array}{c} G \vdash m\ \mathit{member\text{-}of}\ S \qquad G \vdash C \prec_{C1} S \\ G \vdash \mathit{Class}\ S\ \mathit{accessible\text{-}in}\ \mathit{pid}\ C \\ G \vdash \mathit{memberid}\ m\ \mathit{undeclared\text{-}in}\ C \\ G \vdash m\ \mathit{inheritable\text{-}in}\ \mathit{pid}\ C \end{array}}{G \vdash m\ \mathit{member\text{-}of}\ C}\ (\textsc{Inherited})$$

Here m is a pair $\mathit{qtname} \times \mathit{memberdecl}$ consisting of the declaration class of the member and the member declaration itself. With *mbr* and *declclass* we can select the parts. Every freshly declared member is immediately member of the class. A class can also inherit members from its direct superclass[1]: $G \vdash C \prec_{C1} S$[2]. If m is a inheritable member of the direct superclass S and S is accessible in the current package and the class C does not declare a new member with the same *memberid*, then it is inherited by C. The *memberid* of a field is its name, and the *memberid* of a method its complete signature (name plus parameter types). A *Private* member is not inheritable, *Protected* and *Public* members are always inheritable and *Package* members are only inheritable inside the package of the members declaration class.

$\mathit{accmodi}\ m$	$G \vdash m\ \mathit{inheritable\text{-}in}\ P$
$\mathit{Private}$	False
$\mathit{Package}$	$\mathit{pid}\ (\mathit{declclass}\ m) = P$
$\mathit{Protected}$	True
Public	True

[1] In the JLS a class also inherits members from the direct superinterfaces it implements. This is not needed in our model for the following reasons. A wellformedness condition ensures that all interface methods are implemented by the class hierarchy (abstract classes are not supported). So for method inheritance it is sufficient to focus on the class hierarchy. Interface fields are not supported in our current model.

[2] In the following $\prec_C$ is the transitive closure and $\preceq_C$ is the reflexive transitive closure of the direct subclass relation $\prec_{C1}$

If we declare a new member, a member of the superclass with the same *memberid* is coalesced and is not *member-of* the class, according to this definition. Also if a member is not inherited by the subclass it is not *member-of* that subclass. This is important to notice, since at runtime the dynamic type of a reference may be a subclass of the static type, and by that the member we want to access may not be *member-of* the dynamic class anymore. Of course, there must be a superclass providing this member, so we define:

$$G \vdash m\ \textit{member-in}\ C \equiv \exists\, \textit{provC}.\ G \vdash C \preceq_C \textit{provC} \wedge G \vdash m\ \textit{member-of}\ \textit{provC}$$

The following example illustrates the difference of *member-of* and *member-in*:

```
public class A {
  private int n;
}

public class B extends A {
}
```

Since `n` is `private` in class `A` it is not inherited by `B`. So `n` is not *member-of* class `B`. But an object of class `B` will of course also contain the field `n`, since `B` extends `A`. Therefor `n` is *member-in* class `B`.

The basic access restrictions associated with the modifiers are expressed in the predicate $G \vdash m\ \textit{in}\ C\ \textit{permits-acc-to}\ \textit{accC}$. A member m in class C permits the access from an accessing class *accC* according to the following table:

accmodi m	$G \vdash m\ \textit{in}\ C\ \textit{permits-acc-to}\ \textit{accC}$
Private	$\textit{declclass}\ m = \textit{accC}$
Package	$\textit{pid}\ (\textit{declclass}\ m) = \textit{pid}\ \textit{accC}$
Protected	$\textit{pid}\ (\textit{declclass}\ m) = \textit{pid}\ \textit{accC}\ \vee$ $G \vdash \textit{accC} \prec_C \textit{declclass}\ m \wedge (G \vdash C \preceq_C \textit{accC} \vee \textit{is-static}\ m)$
Public	*True*

Access to a *Private* member is only allowed from the declaration class itself. A *Public* member can be accessed from every class. Access to a *Package* member is only allowed in the same package. The restrictions of *Protected* access are twofold. First the member can be accessed from any class in the same package. Secondly the member can also be accessed from outside the package: all involved classes have to be in the same branch of the class hierarchy. Note that this may concern three different classes: The declaration class of the member, the class C the member belongs to (maybe a subclass of the declaration class), and the class *accC* that wants to access the member. With $G \vdash \textit{accC} \prec_C \textit{declclass}\ m$ we ensure that the accessing class *accC* already "knows" of the existence of the member, by being a subclass of the declaration class. For instance members the accessing class must also be a superclass of class C: $G \vdash C \preceq_C \textit{accC}$. This is circumscribed as class *accC* is "involved in the implementation" of class C in the JLS. For static members (class members) this is not necessary. Consider the following example:

```
package P;              package Q;               package R;
public class A {        import P.A;              import Q.B;
  protected int n;      public class B           public class C
}                              extends A {              extends B {
                        }                        }
```

The member n is inherited by both classes B and C. Class B is permitted to access C.n (since B is a superclass of C) but not to access A.n (since B is not a superclass of A). Or in the words of the JLS class B is involved in the implementation of class C but not of class A. Of course class B is permitted to access its own member B.n since B is both the accessing and the accessed class and therefor trivially lies in the same package. Note the differences between the *Protected* case of *inheritable-in* and of *permits-acc-to*. In the JLS inheritance is defined without an extra notion like *inheritable-in*, but with accessibility. That way *Protected* instance members would never be inherited across package boundaries (Bug ID: 4485402). This becomes obvious if we again refer to the example. Class B is not permitted to access A.n. So n would not be inherited by class B if inheritance would be based on this restriction.

Now we are ready to define static accessibility of a member.

$$\frac{\begin{array}{c} G \vdash m\ \mathit{member\text{-}of}\ C \\ G \vdash m\ \mathit{in}\ C\ \mathit{permits\text{-}acc\text{-}to}\ \mathit{accC} \\ G \vdash \mathit{Class}\ C\ \mathit{accessible\text{-}in}\ \mathit{pid}\ \mathit{accC} \end{array}}{G \vdash m\ \mathit{of}\ C\ \mathit{accessible\text{-}from}\ \mathit{accC}}\ (\textsc{Immediate})$$

$$\frac{\begin{array}{c} G \vdash (\mathit{declC}, \mathit{newM})\ \mathit{overrides}_S\ \mathit{old} \\ \mathit{new} = (\mathit{declC}, \mathit{mdecl}\ \mathit{newM}) \\ G \vdash \mathit{new}\ \mathit{member\text{-}of}\ C \\ G \vdash C \prec_C S \\ G \vdash \mathit{old}\ \mathit{of}\ S\ \mathit{accessible\text{-}from}\ \mathit{accC} \\ G \vdash \mathit{Class}\ C\ \mathit{accessible\text{-}in}\ \mathit{pid}\ \mathit{accC} \end{array}}{G \vdash \mathit{new}\ \mathit{of}\ C\ \mathit{accessible\text{-}from}\ \mathit{accC}}\ (\textsc{Overriding})$$

If a member of a class permits access and the class itself is accessible then the member is accessible. That is the Immediate rule. Note that the class has to be accessible, too. *Public* members of a non *Public* class are only visible inside the package. If a subclass is *Public* however, these members become accessible from outside the package if they are inherited. The Overriding rule needs more motivation, since it is not apparent in the JLS (Bug ID 4493343). It states that a method becomes accessible if it overrides another method that is already accessible[3]. This rule is only necessary to cover the special case of *Protected* methods, the other ones can be treated by the Immediate rule — *Public* methods always permit access, *Private* methods cannot be overridden at all and *Package* methods can neither be overridden nor accessed from outside of the package. Consider the following example:

[3] $\mathit{new} = (\mathit{declC}, \mathit{mdecl}\ \mathit{newM})$ means, that the member *new* is a method *newM*; *mdecl* constructs a member from a method.

```
package P;
public class A {
  protected void foo();
}

package P;
import Q.B;
public class C {
  ...
  B b = new B();
  b.bar();  // not accessible
  b.foo();  // accessible
}
```

```
package Q;
import P.A;
public class B extends A {
  protected void foo();
  protected void bar();
}
```

Equipped with the Immediate rule, `C` could only access `protected` members declared in package `P` or in subclasses of `C`. It could not access `B.foo`. But the Sun compiler (SDK 1.3.1) also implements the Overriding rule. It will allow `C` to access `B.foo`, because `C` can access `A.foo` (`A` and `C` are both in package `P`), but will reject access to method `B.bar`. The authors of [3] already reported this irritating behaviour. They consider this a flaw in the language definition. The intention of the language should be that a method overriding another should permit "at least as much access" as the overridden one. Due to the twofold nature of *Protected* access this would actually only be the case if we weaken the modifier to *Public* when we override it outside of the package. The JLS however, is satisfied with *Protected* or *Public*. So [3] suggest to add this restriction and to additionally introduce a new modifier `private protected` permitting access to all super and subclasses, but not to other classes in the same package. Sun considered the Overriding rule as a bug of their compiler and omitted it in the new release (SDK 1.4.0)[4]. So now `C` can access `A.foo` but isn't allowed to access `B.foo`. This is also irritating: If `B` would not redefine `foo` it would inherit `A.foo`. In this case class `C` would be allowed to access `B.foo`. So accessibility of `B.foo` depends on class `B` overriding `foo` or not. This does not fit well to the object oriented paradigm. Whether it is preferable to support the Overriding rule or not is not clear from the software-engineering perspective. As just explained, both solutions lead to some irritating behaviour, that illustrates the difficulty to integrate abstract data types and inheritance seamlessly.

Since overriding plays a major role for accessibility we will now investigate under which circumstances a new method overrides an old one:

$$\frac{\begin{array}{c} \mathit{msig\ new} = \mathit{msig\ old} \qquad \neg\ \mathit{is\text{-}static\ new} \\ G \vdash\!\mathit{Method\ old\ inheritable\text{-}in\ pid}\ (\mathit{declclass\ new}) \\ G \vdash \mathit{declclass\ new}\!\prec_{C1} S \\ G \vdash\!\mathit{Method\ old\ member\text{-}of}\ S \\ G \vdash\!\mathit{Method\ old\ declared\text{-}in\ declclass\ old} \\ G \vdash\!\mathit{Method\ new\ declared\text{-}in\ declclass\ new} \end{array}}{G \vdash \mathit{new\ overrides}_S\ \mathit{old}}\ (\textsc{Direct})$$

$$\frac{\begin{array}{c} G \vdash \mathit{new\ overrides}_S\ \mathit{inter} \\ G \vdash \mathit{inter\ overrides}_S\ \mathit{old} \end{array}}{G \vdash \mathit{new\ overrides}_S\ \mathit{old}}\ (\textsc{Indirect})$$

[4] The compilers of IBM (version 1.1.8 and 1.3.1) both implement the Overriding rule.

Let us first focus on the DIRECT rule. The new and the old method must have the same signature. Overriding (and dynamic binding) is only defined for instance methods and not for static methods ($\neg$ *is-static new*). The overridden method *old* must also be an instance method. This is not visible in this rule but is ensured by a more general wellformedness predicate not shown here (if $G \vdash$ *new overrides*$_S$ *old* holds then also $\neg$ *is-static old* has to hold). The old method has to be inheritable in the declaration class of the new method. The old method has to be member of the direct superclass of the new method's declaration class. Of course, all methods have to be declared properly. The INDIRECT rule is just a transitivity rule for overriding. Let us apply these rules to the following example.

```
package P;
public class A {
   void foo();
}
```

```
package Q;
import P.A;
public class B extends A {
  public void foo();
}
```

```
package P;
import Q.B;
public class C extends B {
  public void foo();
}
```

`B.foo` does not override `A.foo`, since `A.foo` has *Package* access and therefore is not inheritable in package `Q`. `C.foo` overrides `B.foo` since `B.foo` is *Public* and the DIRECT rule is applicable. But does `C.foo` override `A.foo` of the same package? According to our rules it does not, since `A` is not the direct superclass of `C` and the transitivity rule is not applicable either, since `B.foo` does not override `A.foo`. `A.foo` and `B.foo` are treated as uncorrelated methods and so it seems obvious that `C.foo` should not override both of them at the same time. The Java compiler of Sun also behaves compatible with these rules. Sun's Java virtual machine however, does not. In their JVM `C.foo` overrides both `A.foo` and `B.foo`[5]. The Sun JVM seems to implement the following rules for overriding:

$$\frac{\begin{array}{c} \textit{msig new} = \textit{msig old} \qquad \neg\ \textit{is-static new} \\ G \vdash \textit{Method old inheritable-in pid}\ (\textit{declclass new}) \\ G \vdash \textit{declclass new} \prec_C \textit{declclass old} \\ \neg\ \textit{is-static old} \qquad \textit{resTy new} = \textit{resTy old} \\ \textit{accmodi new} \neq \textit{Private} \\ G \vdash \textit{Method old declared-in declclass old} \\ G \vdash \textit{Method new declared-in declclass new} \end{array}}{G \vdash \textit{new overrides old}} \quad (\textsc{Direct})$$

$$\frac{\begin{array}{c} G \vdash \textit{new overrides inter} \\ G \vdash \textit{inter overrides old} \end{array}}{G \vdash \textit{new overrides old}} \quad (\textsc{Indirect})$$

I will refer to these rules as dynamic overriding and to the previous ones as static overriding (indicated by the subscript S in *overrides*$_S$). The DIRECT rule now allows to override not only methods of the direct superclass but also methods from any superclass if they are inheritable. The other novelties in the DIRECT rule can be viewed as wellformedness conditions that ensure typesafety

[5] The JVMs of IBM (version 1.1.8 and 1.3.1) implement another alternative: `C.foo` overrides `A.foo` but not `B.foo`.

at runtime. They are built into the notion of dynamic overriding in the JVM because they are not tested by the bytecode verifier or a runtime check. The JVM only regards a method to override another one, if it is safe to call this method instead of the overridden one. In particular this is only the case if the result types conform. The compiler on the other hand tests these typesafety constraints additionally to overriding: If $G \vdash \textit{new overrides}_S\ \textit{old}$ then the compiler enforces that the result types of the method conform ($\textit{resTy new} = \textit{resTy old}$)[6], the new access modifier is as least as liberal as the old one ($\textit{accmodi old} \leq \textit{accmodi new}$) and the overridden method also is an instance method ($\neg\ \textit{is-static old}$). This is also resembled in our model by a general wellformedness condition that we do not show in this paper. Note that dynamic overriding does not ensure that the access modifier is as least as liberal as the old one. It only has to be non *Private*.

5 Runtime Properties

In an object oriented setting it is usual, that if we statically expect a reference to an object of class A we can receive an object of class B at runtime. Class B then has to be a subclass of A. In Java, it is possible that A is declared *Public* but B is not. So we can receive an object of class B outside of its packages due to a reference of type A, although B is not statically accessible. As accessibility of the class is a precondition for accessibility of a member, we cannot expect that during runtime only the statically accessible members are the members valid to access. We need a more liberal predicate to capture the runtime properties we should expect:

$$\frac{\begin{array}{c} G \vdash m\ \textit{member-in}\ C \\ G \vdash m\ \textit{in}\ C\ \textit{permits-acc-to}\ accC \end{array}}{G \vdash m\ \textit{in}\ C\ \textit{dyn-accessible-from}\ accC}\ (\textsc{Immediate})$$

$$\frac{\begin{array}{c} G \vdash (declC, newM)\ \textit{overrides}\ old \\ new = (declC, mdecl\ newM) \\ G \vdash new\ \textit{member-in}\ C \qquad G \vdash C \prec_C S \\ G \vdash old\ \textit{in}\ S\ \textit{dyn-accessible-from}\ accC \end{array}}{G \vdash new\ \textit{in}\ C\ \textit{dyn-accessible-from}\ accC}\ (\textsc{Overriding})$$

These rules of dynamic accessibility resemble the rules of static accessibility, but leave out the precondition that the types must be accessible and switch from *member-of* to *member-in* and from static overriding ($\textit{overrides}_S$) to dynamic overriding (*overrides*). We want to ensure that for a wellformed program (only statically accessible members are accessed) only dynamically accessible members are accessed during runtime. Static accessibility is tested by the compiler to decide whether a given program is wellformed or not and therefor whether or not to accept the program. It can also be tested by the bytecode verifier to decide whether or not to run a program. Dynamic accessibility then captures the properties of the actual member accesses that can occur during execution of the wellformed program. The Overriding rule for the dynamic case is not as questionable as for the static case. If a method overrides another one it will be called anyway, due to dynamic binding, and therefor we have to accept such

[6] In our model we are more liberal and accept all new result types that widen to the old one (instead of being equal)

calls during runtime. So it is irrelevant if we support the OVERRIDING rule in the static case or not, in the dynamic case we have to deal with it. Only if we enforce that a *Protected* method can only be overridden outside of its package by a *Public* method, we can omit the OVERRIDING rule in both the dynamic and the static case, since then all legal accesses are captured by the IMMEDIATE rules.

We model the runtime behaviour of Java with a big step semantics. Whenever the dynamic accessibility is violated we throw a special exception that halts the program and signals the error. The following theorem states that this exception will never been thrown when executing wellformed programs.

Theorem 1. $[\![G\vdash \mathit{s0} -t\succ\!\to (v,\ \mathit{s1});\ (\!|\mathit{prg}{=}G,\mathit{cls}{=}\mathit{accC},\mathit{lcl}{=}L|\!)\vdash t{::}T;\ \mathit{wf\text{-}prog}\ G;\ \mathit{s0}{::}{\preceq}(G,L)]\!] \Longrightarrow \mathit{error\text{-}free}\ \mathit{s0} = \mathit{error\text{-}free}\ \mathit{s1}$

Evaluating the Java term t leads us from state $\mathit{s0}$ to state $\mathit{s1}$ and gives us v as result. Java statements and expressions are generalised to terms in this semantics. Statements evaluate to a dummy result. The term t is welltyped in the body of class *accC* ($(\!|\mathit{prg}{=}G,\mathit{cls}{=}\mathit{accC},\mathit{lcl}{=}L|\!)\vdash t{::}T$) and the whole program is wellformed. This guarantees that only statically accessible members are accessed. The starting state conforms to the environment ($\mathit{s0}{::}{\preceq}(G,L)$). This implies that all values within the state are compatible with their static types. L is the static typing environment for local variables. The exception components are encoded into the state. So this theorem guarantees that if we start in a error-free state we will end up in an error-free state (no access violation has occurred during evaluation). The proof is closely related to the type safety proof in [5].

As we know now that dynamic accessibility is guaranteed for wellformed programs we can look at some derived properties.

Lemma 1. $[\![G\vdash m\ \mathit{in}\ C\ \mathit{dyn\text{-}accessible\text{-}from}\ \mathit{accC};\ \mathit{accmodi}\ m = \mathit{Private}]\!] \Longrightarrow \mathit{accC} = \mathit{declclass}\ m$

A *Private* member can only be accessed from its declaration class. This is a simple conclusion from the definition of *permits-acc-to*.

Lemma 2. $[\![G\vdash m\ \mathit{in}\ C\ \mathit{dyn\text{-}accessible\text{-}from}\ \mathit{accC};\ \mathit{accmodi}\ m = \mathit{Package};$
$\mathit{wf\text{-}prog}\ G]\!] \Longrightarrow \mathit{pid}\ \mathit{accC} = \mathit{pid}\ (\mathit{declclass}\ m)$

A *Package* member can only be accessed from inside the package. This obviously is a desirable property. For fields it is a simple conclusion from the definition of *permits-acc-to*. For methods it is rather involved since we switch from static to dynamic overriding. That is why we again need the wellformedness precondition of the program, to prove this lemma. As mentioned before, dynamic overriding does not ensure that the access modifier of the overriding method is as least as liberal as the modifier of the overridden method. This is only guaranteed for static overriding in wellformed programs. The JVM does not ensure this during bytecode verification or via a runtime test. So the executed programs are not necessarily wellformed in this sense. In hand written bytecode for example it is possible to call a *Package* method that overrides a *Public* method from outside of the package without an error.

Lemma 3. $[\![G \vdash f \ \mathit{in}\ C\ \mathit{dyn\text{-}accessible\text{-}from}\ \mathit{accC};\ \mathit{accmodi}\ f = \mathit{Protected};\ \mathit{is\text{-}field}\ f;\ \neg\ \mathit{is\text{-}static}\ f;\ \mathit{pid}\ (\mathit{declclass}\ f) \neq \mathit{pid}\ \mathit{accC}]\!] \Longrightarrow G \vdash C \preceq_C \mathit{accC}$

Outside of the package a *Protected* instance field can only be accessed from a superclass. This lemma does not hold for methods due to overriding.

6 Conclusion

In this paper we clarify the semantics of Java access modifiers and packages by formalising them in the theorem prover Isabelle/HOL and proving some key properties. Our model reflects the subtle interaction between inheritance, overriding and accessibility in Java. The Java formalisation of [5] was sufficiently mature to let us add and analyse the new concepts. This again shows, that it is feasible to investigate aspects of a realistic programming language completely formally in a theorem prover. Although the Java technology is now available for almost seven years and there already exists a second edition of the language specification, some aspects about the semantics remain unclear and ambiguous. This leads to different implementations of “overriding” in the compiler and the JVMs and to some disagreement of various Java compilers (of different versions and vendors) whether or not to support the Overriding rule to determine accessibility. Most of the sophisticated rules introduced in this paper became apparent when we failed to prove some expected properties of simpler versions. So to get a clear and unambiguous semantics of a programming language it appears to be very useful to carry out the language design formally with support of proof assistants.

Acknowledgements I am grateful to Gilad Bracha, Gerwin Klein, Tobias Nipkow, Martin Strecker and the anonymous referees for comments on the draft versions of this paper.

References

1. Guy L. Steele Jr. James Gosling, Bill Joy and Gilad Bracha. *The Java Language Specification, Second Edition.* Addison Wesley, 2000.
2. T. Jensen, D. Le Métayer, and T. Thorn. Security and dynamic class loading in Java: A formalisation. In *IEEE International Conference on Computer Languages*, pages 4–15, Chicago, Illinois, 1998.
3. P. Müller and A. Poetzsch-Heffter. Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur. In C. H. Cap, editor, *JIT '98 Java-Informations-Tage 1998*, Informatik Aktuell. Springer-Verlag, 1998.
4. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS 2283.
5. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.
6. Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.
7. Sun. Java developer connection. Available from `http://java.sun.com/jdc`.