Replace this file with prentcsmacro.sty for your meeting,
or with entcsmacro.sty for your meeting.
Both can be found at the ENTCS Macro Home Page.

Techniques for the Verification of Java Card Programs on the Source Code Level

Nicole Rauch [1,3] and Jörg Meyer [2,3]

Praktische Informatik V
FernUniversität Hagen
Hagen, Germany

Abstract

In the VerifiCard project [4] we concentrate on the development of formal techniques which provide the specification and Hoare-logic based verification of Java Card programs. They directly work on the programming language level.

Within this paper we sketch some results of this ongoing research. Based on a Hoare-logic for a small object oriented programming language kernel, we present some techniques to bring the programming logic and the programming language Java Card together, and some techniques to simplify the verification process as much as possible to obtain a maximum of automatization for the verification process. The techniques are based on program analysis, generation, transformation, and reconstruction. Some examples of problems that arise in developing these techniques are illustrated. Finally, the variety of tools we utilize to verify Java Card source code is introduced.

1 Introduction

Although Smart Cards are small devices with relatively restricted resources and usually require programming on a machine level, the Java Card approach to Smart Card programming utilizes the fully flavored programming language Java Card as a programming language for the Java Card API which is provided by Sun ([13]) and for user applications. On the one hand, this allows a very comfortable development of Smart Card applications, almost

[4] www.verificard.org

[1] Email: Nicole.Rauch@Fernuni-Hagen.de

[2] Email: Joerg.Meyer@Fernuni-Hagen.de

[3] The fragments of the Hoare logic presented in this paper are part of an ongoing work of Arnd Poetzsch-Heffter.

like developing usual Java applications, on the other hand it generates problems very similar to those treated by formal methods for program checking, specification, and verification.

Our task is to verify the specifications of Java Card programs on the source code level. To accomplish this, we use a Hoare-style programming logic which is based on the programming language Java-KE that we introduce in Subsection 3.1. We develop a method for proving specifications of the classes and methods of the program we are examining.

Java Card programs depend on the Java Card API. This API in fact represents a framework for Java Card applets. Every Java Card applet consists of at least one class `C` which implements a subclass of the class `javacard.framework.Applet`. The class `Applet` contains an `install` method which must be overwritten by `C`. In this method an instance of `C` is created and registered with the runtime environment. The method `process` is also expected to be overwritten by `C` because it defines the actions of the applet. This is the expected behavior of these two methods and has to be used in the specifications of these methods.

We specify a given program and prove this specification by means of the program source code. To achieve this, we need to specify the API and prove its specification first because every Java Card program heavily relies on the API, as illustrated above. The specification and verification of the API need of course only be performed once for all further program verifications. Class invariants, pre- and postconditions of methods are our means to specify the program. From this specification we generate proof obligations. The proof obligations are a set of Hoare logic triples. While proving these proof obligations, more program independent proof obligations are generated. The latter can be passed to the theorem prover PVS for proving. We are planning to also support the theorem prover Isabelle (see [8]) in the future.

We are using source code level verification because it is easier to handle by the user. If a program is transformed into a logic or into a formula, it is altered heavily, which makes it hard for the average user, who is the programmer of an application and who is not educated in logic, to follow the proof. Although we are transforming the source code as well, we still stay close to the original program. This aids the user in the verification process because he can recognize the code he wrote when it is displayed to him by the verification tool.

In this paper we present some techniques for source code level program verification. The main aspect is to bring the programming logic and the language to be verified together. Restrictions regarding the formal background of the logic and the complexity of the language must be accepted. Some solutions, especially suggestions to slightly extend the logic and to perform some transformations of the source code to eliminate certain language constructs, are sketched later in this paper.

The tool we are building which is supposed to generate automatic proofs, to interact with the user and to communicate with the theorem prover is called Sjive, the Smart Java Interactive Verification Environment.

Related Work

There are several ways to provide specification and verification techniques for Java-like programming languages. Some of them focus on improvements of formal methods in very special areas, like thread support, and others. We concentrate on the object imperative kernel language with its object oriented extension. Von Oheimb (cf. [7]) translates Java programs into data types of the theorem prover Isabelle and provides reasoning techniques over Java programs within Isabelle. This especially allows meta theory reasoning. The LOOP tool (cf. [5]) uses a coalgebraic approach where statements and methods are translated to functions which act on a given state space. Properties of these functions can be shown within a theorem prover (PVS or Isabelle) which provides the formal platform for this approach.

Overview

The second chapter presents some important aspects of Java Card with regard to source code based verification. In the third chapter we introduce our Hoare-style logic and the programming language Java-KE the logic is based upon. The fourth chapter shows how we can bring Java Card and Java-KE together.

2 Java Card

2.1 Differences between Java Card *and Java*

Java Card is a subset of Java. Several restrictions of Java Card simplify the verification process. They are thus investigated more closely in the following. For a complete overview of the differences between Java Card and Java see [14].

Threading is not supported in Java Card. As thread support is a problem class of its own, Java Card is well suited for us to demonstrate our techniques without introducing too many problem classes at the same time. The primitive types float and double are not allowed in Java Card. It is nontrivial to provide support for reasoning about floating point operations. The primitive types char and long are not supported in Java Card either. These limitations reduce our logical framework.

The smaller the number of types and syntactical constructs in our programming logic, the less complex the verification and the abstract data types can be. This simplifies proving because the strategies are simpler and the proofs generated by them are smaller and less complicated.

2.2 *The main focus in verifying* Java Card

When looking at Java Card code, be it the API or the samples delivered by Sun, one quickly gets the impression that in such a limited environment there is no space for fancy language constructs, abstraction, or overly much use of encapsulation. Everybody just wants to get his work done with the few resources the Smart Cards have to offer. This leads to small-scope programming. The most widely used language constructs are casts, which are mainly used to cast integer constants to byte or short variables in order to save memory, logical AND-ing and OR-ing, comparison operators, assignment, method invocation, addition and subtraction, and bit-operations. Integer values are widely used as status indicators or for error codes. Figure 1 illustrates the usage of many of these language constructs in a short example.

Most programs have a simple structure: statements usually consist of one assignment or one method call, the parameters that are passed to the methods are fields or local variables, and often static methods are called. Thus the extraction of expression parts that do not have side effects is not too difficult. Properties of arithmetic expressions without side effects can be proved within a PVS theory when the arithmetic is embedded into the PVS logic.

```
final static byte  PURSE_CLA               = (byte)0x80;
final static byte  VERIFY                  = (byte)0x20;
final static byte  READ                    = (byte)0xB2;
final static byte  INITIALIZE_TRANSACTION = (byte)0x20;
final static byte  COMPLETE_TRANSACTION   = (byte)0x22;
final static byte  INITIALIZE_UPDATE      = (byte)0x24;
final static byte  COMPLETE_UPDATE        = (byte)0x26;
// ...
//prepare transaction record in APDU buffer
short offset = Util.setShort(buffer, START, newTN);
buffer[offset] = (byte)transientShorts[TRANSACTION_TYPE_IX];
offset++;
offset = Util.setShort(buffer, offset, amount);
//CAD ID was left in this array from Initialize Transaction
offset = Util.arrayCopyNonAtomic(CAD_ID_array, START, buffer,
                                 offset, ID_LENGTH);
// ...
// Short file identifier
byte sfi = (byte) ( ( P2 >> 3 ) & 0x1F );
fid = Util.makeShort(FID_BYTE, sfi);
```

Fig. 1. An excerpt of class `com.sun.javacard.samples.JavaPurse.JavaPurse`

The proving of expression features must be supported efficiently by our verification tool Sjive as a large number of programs uses these expressions.

This also holds for operators. They are extensively used in JAVA CARD programs. This is illustrated in Figure 1, where an expression, either an operator or a method call, is used in each line of the code. There are two possibilities for proving expressions: Operator calls can be mapped to method calls with the same semantics, e.g. `a + b` can be mapped to `Operator.plus(a, b)` where `plus` is a static method of the class `Operator` which has to be provided. In this case, expressions are not supported by the logic as all expressions are transformed into statements. The other solution is to directly support expressions in the logic. Special care has to be taken of expressions with side effects. If a language construct does not have any side effects, it can be proved automatically, which is convenient for the user.

Therefore, we plan to develop verification strategies with a strong concern for expressions without side effects. The difficulty does not lie in proving parts of a program but in handling the interaction between the program and the API.

3 A Hoare Logic for a Java Card Subset

Ideally, one wants to have a programming logic which supports the whole language JAVA CARD. This is hard to achieve because of the complexity of proving this huge logic sound and relative complete.

Additionally, some aspects of JAVA CARD can hardly be covered by a Hoare style logic, e.g. the grouping of classes into packages.

For verifying the JAVA CARD source code we use a Hoare-style programming logic (cf. [3]) which was developed from the logic presented in [10]. This logic formalizes the axiomatic semantics of the programming language JAVA-KE which is a (small) subset of JAVA CARD. Correctness of a given program can thus be shown by translating its specification into Hoare triples and verifying these triples using the logic. For a detailed presentation of our current logic's predecessor see [12].

3.1 The Programming Language JAVA-KE

This subsection briefly introduces the programming language JAVA-KE our logic is based upon. This language is derived from the language JAVA-K which is presented in [12]. The main enhancement is that it now contains exceptions. This is symbolized in the language name, as JAVA-KE means "JAVA-K with Exceptions". JAVA-KE is a Java subset with a reduced statement and expression syntax, but in which dynamic method invocation and all object-oriented features are fully supported.

The syntax definition of JAVA-KE consists of three parts. The first part treats packages and imports, the second part deals with type declarations and member declarations, and the last part specifies statements and expressions. The syntax is listed in a data type notation. Subtypes are declared in a constructor-like notation while alternative subtypes are separated by `|`.

data type

```
Package       = pack( PackId ImportList TypeDeclList )
ImportList    = list of PackId
TypeDeclList  = list of TypeDecl

TypeDecl      = ClassDecl( CTypeId CTypeId ITypeIdList ClassBody )
              | InterfaceDecl( ITypeId ITypeIdList InterfaceBody )
ClassBody     = list of MemberDecl
MemberDecl    = FieldDecl( Type FieldId )
              | MethodDecl( MethodSig Statement )
InterfaceBody = list of MethodSig
MethodSig     = Sig( Type MethodId Type )
Type          = booleanT() | intT() | nullT() | ct( CTypeId )
              | it( ITypeId )

Statement     = block( Type VarId Statement )
              | fread( VarId VarId FieldId )
              | fwrite( VarId FieldId VarId )
              | cassign( VarId Type Exp )
              | new( VarId CTypeId )
              | seq( Statement Statement )
              | if( Exp Statement Statement )
              | while( Exp Statement )
              | catch( Statement CTypeId VarId Statement )
                    // try-body  CTypeId= NullPExc or CastExc
              | invoc( VarId VarId MethodId Exp )
              | call( VarId CTypeId MethodId Exp )
Exp           = ic( Int ) | bc( Bool ) | null() | id( VarId )
              | unary( UnOp Exp ) | binary( Exp BinOp Exp )
```

`PackId`, `FieldId`, `MethodId`, `CTypeId`, `ITypeId`, and `VarId` are simple identifiers.

JAVA-KE lacks several important language constructs, e.g. arrays, the types `byte`, `short` and `void`, modifiers, constructors, static and nonstatic initializers, field declaration with assignment of an initial value, field declaration in interfaces, the keywords `throws` and `finally`, modifiers, `for`-statements, `switch`-statements, `do`-statements, `break`- and `continue`-statements, `return`-statements, conditional expressions, and qualified field and method access.

This gap between JAVA-KE and JAVA CARD can be closed by either disallowing part of JAVA CARD, by slightly extending the logic or by transforming the source code. As we will show in Section 4, a thorough mix of these three options is used to achieve our goal.

3.2 The Programming Logic

To verify that a given implementation of a method obeys its interface specification, we use a Hoare-style programming logic that is based on JAVA-KE.

Our logic contains 9 axioms and 22 rules. The rules treat subtyping, statements including try-catch-statements, and method invocation. Currently, operators are not supported by our logic yet. The axioms and rules support JAVA CARD well. Recursion is treated by assumptions; we assume the annotation of a given method to prove its body. Dynamic dispatch is handled by the following rules, where $\prec$ denotes the subtype-relation, T:m represents the virtual method m of class T (i.e. m can be called on an object of type T, but is not necessarily implemented in T) and T@m symbolizes the implementation of method m in class T (thus, with *impl* returning the implementation of a concrete method T:m we can state that *impl*(T,m) = R@m where T $\preceq$ R). $\tau()$ yields the type of its argument. We assume that the program to be verified is syntactically correct, so we need not take care of encapsulation aspects in our rules.

The *impl-rule* states that the specification of a method must hold for the method body and describes the recursion treatment mentioned above:

impl-rule:

$$\frac{\mathcal{A}, \{\ \mathbf{P}\ \}\ \mathrm{T@m}\ \{\ \mathbf{Q}\ \} \vdash \{\ \mathbf{P} \wedge res = init(rtyp(\mathrm{T@m}))\ \}\ body(\mathrm{T@m})\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{T@m}\ \{\ \mathbf{Q}\ \}}$$

We can give a specification of a virtual method T:m. This is used in handling dynamically bound methods. This specification must hold (1) for the (possibly inherited) implementation of m in class T, and (2) for all implementations of m in all subclasses of T. To prove the second obligation, we assume that the program cannot be extended later on. Thus, all subtypes $\mathrm{S}_1, \ldots, \mathrm{S}_n$ of T are known. For a subtype S_0 of T the equivalence $\mathrm{S}_0 \prec \mathrm{T} \Leftrightarrow \mathrm{S}_0 \preceq \mathrm{S}_1 \vee \ldots \vee \mathrm{S}_0 \preceq \mathrm{S}_n$ holds. We can prove a method specification by proving it for all subtypes.

The *class-rule* is used in proving concrete methods. It states that both proof obligations mentioned above must be fulfilled.

The *subtype-rule* is applied to abstract methods. In this case only the second obligation remains to be proved because there is no implementation of an abstract method.

class-rule: Be m a method in class T, i.e. *impl*(T,m) is defined.

$$\frac{\begin{array}{l}\mathcal{A} \vdash \{\ \tau(this) = \mathrm{T} \wedge \mathbf{P}\ \}\ impl(\mathrm{T}, \mathrm{m})\ \{\ \mathbf{Q}\ \} \\ \mathcal{A} \vdash \{\ \tau(this) \prec \mathrm{T} \wedge \mathbf{P}\ \}\ \mathrm{T{:}m}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{T{:}m}\ \{\ \mathbf{Q}\ \}}$$

subtype-rule:

$$\frac{\begin{array}{l}\mathrm{S} \preceq \mathrm{T} \\ \mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \mathrm{S{:}m}\ \{\ \mathbf{Q}\ \}\end{array}}{\mathcal{A} \vdash \{\ \tau(this) \preceq \mathrm{S} \wedge \mathbf{P}\ \}\ \mathrm{T{:}m}\ \{\ \mathbf{Q}\ \}}$$

Exceptions can be handled by duplicate rules that have been modified so that one rule treats the occurrence of an exception and the other one treats exception-free execution, like in the rule for executing a sequence of statements. If an exception occurred, it is stored in exc, otherwise exc equals null. Statement sequences are defined in JAVA-KE as `seq(Statement Statement)`.

seq-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ s_1\ \{\ \mathbf{R} \wedge exc = \text{null}\ \} \qquad \mathcal{A} \vdash \{\ \mathbf{R}\ \}\ s_2\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ s_1\ s_2\ \{\ \mathbf{Q}\ \}}$$

seq-exc-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ s_1\ \{\ \mathbf{R} \wedge exc \neq \text{null}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ s_1\ s_2\ \{\ \mathbf{R}\ \}}$$

Other rules are not duplicated in order to treat exceptions. Here, the possible occurrence of an exception is integrated into the rule, e.g. as an additional termination criterion of a while-loop. While-loops are defined in JAVA-KE as `while( Exp Statement )`.

while-rule:

$$\frac{\mathcal{A} \vdash \{\ e \wedge \mathbf{P}\ \}\ s\ \{\ \mathbf{P}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{while}(e)\{s\}\ \{\ ((exc \neq \text{null}) \vee \neg\, e) \wedge \mathbf{P}\ \}}$$

Of course, some rules are necessary to prove try-catch-expressions (defined in JAVA-KE as `catch( Statement CTypeId Statement )`). We need the *catch-rule* if no exception occurred, the *catch-exc-rule* if an exception occurred which is not caught by the catch-clause, and finally the *catching-rule* if an exception occurred which is indeed caught by the catch-clause.

catch-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ s_0\ \{\ \mathbf{Q} \wedge exc = \text{null}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{try}\{s_0\}\text{catch}(\text{T e})\{s_1\}\ \{\ \mathbf{Q}\ \}}$$

catch-exc-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ s_0\ \{\ \mathbf{Q} \wedge exc \neq null \wedge \tau(exc) \not\preceq T\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{try}\{s_0\}\text{catch}(\text{T e})\{s_1\}\ \{\ \mathbf{Q}\ \}}$$

catching-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ s_0\ \{\ \mathbf{Q} \wedge exc \neq \text{null} \wedge \tau(exc) \preceq T\ \} \qquad \mathcal{A} \vdash \{\ \mathbf{Q}[\text{e}/exc]\ \}\ s_1\ \{\ \mathbf{R}\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}\ \}\ \text{try}\{s_0\}\text{catch}(\text{T e})\{s_1\}\ \{\ \mathbf{R}\ \}}$$

The invocation of the correct catch-rule can be handled by a strategy.

4 Programming Language Based Program Verification

In this section we will take a detailed look at our approach to bridge the gap between JAVA CARD and JAVA-KE which we described in Subsection 3.1.

The solutions presented here should easily be transferable to other object-oriented languages, especially Java.

4.1 *Disallowed parts of* Java Card

Not all syntactical constructs of Java Card are currently supported by our verification tool Sjive. At the first glance, this seems to be a disadvantage because we cannot verify programs that use these language constructs. It seems especially disadvantageous if some of these language constructs were used by the API because then we would not even be able to verify any program that uses the affected API classes. In our decision over which language constructs to regard as unsupported, we examined the API classes and every Java Card program available to us, and so far we did not find any program that was rejected by Sjive because it implemented one of these unsupported language constructs. Thus the seeming disadvantage turns into an advantage because the programmers in fact do not use these language constructs, and by omitting them we can reduce the size of our verification tool.

These are the constructs that are omitted:

- Nested classes and interfaces. They are not supported because usually those language constructs are not used in Java Card programs. The programmers try to keep the number of classes small as the number of classes and interfaces allowed in a package is limited to 255 (cf. [14], Paragraph 2.2.4.1). Encapsulation is rarely used either, as we already pointed out in Subsection 2.2. Disallowing inner classes and other member types leads to some unsupported language constructs, like the creation of anonymous classes and the accessing of member class members via expressions like *ClassName.super.Identifier*.
- We also restrict field access and method invocation with qualified names (like $X.Y.Z.m()$) because the qualified name is highly ambiguous. It can start with a package name or not, it can contain a class name (if a static member is accessed) or not, followed by field names and method names in arbitrary combinations. Our solution is to always regard the prefix of qualified names as field access.
- Labels are not allowed because they prevent structured programming and a structured program flow. This makes the source code very difficult to verify, and even difficult to understand for humans. Considered the fact that labels are used rarely anyways, we decided to completely omit them.

4.2 *Extending the Logic*

The presented programming language and the programming logic lack important features, which make them applicable to Java Card. Currently support for methods with more than one parameter, for arrays, and for expression handling is missing. Supporting methods with more than one parameter is no problem because the logic can be canonically extended for this. Array support is more complex. To support arrays, rules for writing values into an array and reading values from an array have to be added. Both have to take care of the handling of `ArrayIndexOutOfBoundsExceptions`. Furthermore the data and state model the Hoare logic is based upon has to be extended to support arrays.

Support for expressions can be added in several ways. The whole program can be translated into a representation where only simple expressions like variables, this, and super are supported. Another way is to provide a Hoare logic which completely supports all kinds of expressions. Both approaches have their advantages and disadvantages. A complete translation makes the program code less readable and extends the program a lot. A Hoare logic for the complete expression syntax is very complicated to use because expressions can cause side effects with aliasing in any way. Thus the approach we propose is a mix of both techniques, which translates the source program into a simpler language and therefore prevents complex combinations of language features. In addition, usual arithmetic expressions which use the imperative language kernel of Java Card can be supported by an extension of the programming logic. In addition this technique on the one hand allows to isolate the hot spots for verification and on the other hand supports automated techniques for program parts that are straightforward to verify.

4.3 Source Code Transformations

There is a wide gap between the language Java Card we want to verify and the language Java-KE which our logic is based upon, as we already pointed out in Subsection 3.1.

But this is not necessarily a disadvantage as we can make use of the semantical similarity of several of these statements to transform them so that only one of these statements remains. Other language constructs that mainly exist as a convenience for the programmer, like the conditional operator or the assignment operators `+=`, `-=` etc., can be replaced by syntactically longer but semantically equivalent code. This aids in verifying as it reduces the number of rules required in our logic. Additionally, small, simple language constructs are much easier to handle than large, complex constructs because the more complex a construct the "more semantics" is involved in it. Therefore we try to break down the Java Card language constructs as much as possible, replacing them with semantically equal code fragments, and we remove semantically similar "duplicate" constructs.

To give an example of such a transformation, it is striking that our programming language Java-KE does not support constructor calls and initializers.

In Java Card, when an object is created by a new-expression, the memory for the object is allocated, fields that are directly initialized are assigned their values, any initializers are executed, and then the appropriate constructor is called. This constructor must be defined in the class that defines the dynamic type of the newly-created object. If it is not defined by the programmer and no other constructors are defined, the compiler provides the default constructor without arguments which simply calls the superclass constructor.

Each constructor explicitly or implicitly calls a constructor of its superclass, except if the current class is `java.lang.Object`, in which case no super class exists and thus no super class constructor can be called.

Constructor calls and initializers can easily be transformed according to the code in Figure 2. Without constructors, object creation need not take care of code execution, which again simplifies our logic.

```
class S {                     class S {
   int a = 0, b, c;              int a, b, c;

   {                             void initializer(){
                                    this.a = 0;
      b = 1;                        this.b = 1;
   }                             }

   S(int i){                     S S_constructor(int i){
                                    Object_constructor();
                                    initializer();
      c = i;                        this.c = i;
   }                             }
}                             }

class C extends S {           class C extends S {
   int d;                        int d;

   C(int i){                     C C_constructor(int i){
      super(i);                     S_constructor(i);
      d = i;                        this.d = i;
   }                             }
}                             }

// ...                        // ...
C c = new C(1);               C c = new C().C_constructor(1);
S s = new C(1);               S s = new C().C_constructor(1);
```

Fig. 2. Constructor and initializer transformation

The call of `Object_constructor()` can be omitted because it is known to be empty in Java Card. The example calls at the end of the conversion look a bit overcomplicated because the constructor-replacement calls are executed directly after object creation. In the first example, these calls can be split up into two statements like `C c = new C(); c.C_constructor(1);`. But in the second example, this does not work because the static type of the variable `s` is `S` which does not contain a `C_constructor` method, so the split-up code would not even compile.

The conversion solution sketched above seems quite implausible; at first sight one could imagine more straightforward solutions. But in taking a closer look at the transformation attempts given in Figure 3, the reader will notice that these are incorrect.

```
class A {}
class B extends A {}

class S {
   S(){ }

   S(B b){ }

}

class C extends S {

   C(A a){

   }
}

// ...
S s = new C(new B());
```

```
class A {}
class B extends A {}

class S {
  void ctor(){ }

  void ctor(B b)
  { }
}

class C extends S {
  void
  ctor(A a){
    super.ctor();
  }
}

// ...
S s = new C();
s.ctor(new B());
```

```
class A {}
class B extends A {}

class S {
  static void
    ctor(S s){ }
  static void
    ctor(S s, B b){ }
}

class C extends S {
  static void
    ctor(C c, A a){
    S.ctor(c);
  }
}

// ...
S s = new C();
C.ctor((C)s, new B());
```

Fig. 3. Incorrect transformation attempts

The leftmost column contains the original code. The middle column turns the constructors into instance methods. This leads to the problem that in the example given below the code the method `S@ctor` is called because this is the “most specific” method, although one expects the method `C@ctor` to be called.

In the rightmost column, the constructors are transformed into static methods which will be called by specifying the containing class explicitly. But still, this conversion fails. The call of the example code below the transformation does not even compile because the method invocation is ambiguous: `C@ctor` is more specific in the first argument, whereas `S@ctor` is more specific in the second argument.

The initialization values of static fields cannot be put into a similar method `static_initializer()` because one cannot ensure that this method is called before one of the static fields is accessed (this could only be guaranteed if the method call would be generated into the runtime environment). But as static fields may only be initialized to compile-time constants or arrays of constants, as stated in [14], Paragraph 2.2.4.5, we need not perform calculations or evaluate fields to get at the initial values. Thus these values can be added to the program specification. If the field is final, the value can be added to the class invariant specification.

By means of these simple transformations constructors, static and non-static initializers are removed from our code and replaced by methods which we already know how to prove. Thus the logic can be kept smaller, and the workload for building the verification system is reduced immensely.

Other language constructs can be transformed as well, e.g. for-loops and do-loops can be transformed into while-loops, conditional expressions can be replaced by if-then-else-statements, switch-statements can be expressed by if-then-else-cascades (where special attention must be given to code running from one switch-statement to another), array initializers can be expressed by a sequence of assignments and so on.

Another kind of transformation is the removal of language constructs that are irrelevant to the verification process. Examples are the separation of types into compilation units and packages, or import-statements. They are not necessary for the verification because once the name analysis is finished, we know which identifier belongs to which class. So these statements become superfluous and can be removed.

Our intention is to provide a kind of "reverse transformation" so that the program can be output in a shape that is not too different from its original shape. Of course, we will not be able to really bring the program back into its original state. If, for example, a class contains more than one initializer, we combine the code of all initializers into a single `initializer()` method. We are not able to divide the code back into several initializers afterwards. But we will try to find a "normal form" for outputting the code which will bear enough resemblance to the original code for the user of Sjive to still recognize her code.

4.4 Tool Support

For solving the problems that arise when verifying Java Card, we need a mix of technologies. First, we ensure that the code is syntactically correct by compiling it with a Java compiler. This is possible because Java Card is a subset of Java (cf. [14]). We use the Java compiler kopi because it can be called from within our verification tool Sjive, it does not terminate after compilation, and it can be told to suppress binary output which speeds up this step a little as we do not need binaries.

Next, we scan and parse the source code. The scanner is automatically created by JFlex (cf. [6]). The parser is automatically generated by the Java parser generator CUP (cf. [4]).

We work with the MAX system (cf. [9,11]) to represent the source code as an attributed abstract syntax tree. MAX produces C code which is called from Sjive via the Java Native Interface.

Finally, PVS (cf. [2]) is integrated into Sjive. We use it as general theorem prover for type checking and for formula verification. The integration is established in a similar way as is described in [1]. Our long-term goal is to put up a PVS server that can be connected to via TCP/IP so that it is no longer necessary to have PVS run on the same machine as Sjive.

The connection to PVS is not required to run Sjive. If the connection is undesired or impossible for some reason, Sjive can be told so by a command line switch. Without PVS, of course proofs cannot be carried out. Sjive will then only generate the proof obligations.

5 Conclusion

In this paper we sketched some ideas how to bring our programming logic and the language Java Card together. We suggested some extensions to the programming logic to support methods with more than one parameter, arrays, and expression handling. We also gave an example how to remove constructors and initializers by means of source code transformation.

The development of strategies will allow for a maximum of automatization in the verification process. Still it is possible to perform a proof interactively.

References

[1] B. Buth. An interface between Pamela and PVS. Technical report, Universität Bremen, 1997. Available from `http://www.informatik.uni-bremen.de/~bb/bb.html`.

[2] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[3] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[4] Scott E. Hudson. CUP user's manual, July 1999. Available from `http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`.

[5] M. Huisman. *Reasoning about Java programs in higher order logic Using PVS and Isabelle.* PhD thesis, University of Nijmegen, 2000.

[6] Gerwin Klein. JFlex user's manual, version 1.3.5, October 2001. Available from `http://www.jflex.de/docu.html`.

[7] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/ oheimb/diss/`.

[8] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1994.

[9] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX System. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 137–150. Springer-Verlag, 1993.

[10] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[11] A. Poetzsch-Heffter and T. Eisenbarth. *The MAX System. A Tutorial Introduction.* Institut für Informatik, Technische Universität München, November 1997.

[12] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, volume 1576 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[13] Sun Microsystems, Inc., Palo Alto, CA. *Java Card^{TM} 2.1.1 Application Programming Interface*, May 2000.

[14] Sun Microsystems, Inc., Palo Alto, CA. *Java Card^{TM} 2.1.1 Virtual Machine Specification*, May 2000.