Fully Verified JAVA CARD API Reference Implementation

Wojciech Mostowski
woj@cs.ru.nl

Radboud University Nijmegen

Background & Motivation

- Full specification and implementation for the verification of JAVA CARD applets
  - reasoning on the level of interfaces very efficient (in 99.9% of the cases). . .
  - . . . but not always possible/feasible: strong invariants and transaction mechanism
  - full functional verification of contrived applets that prevent fault injections on the source code level
- Builds on top of earlier work: Sun Reference Implementation, Daniel's OCL work, Nijmegen gang's JML work
- Impl. + Spec. $\rightarrow$ Verification $\rightarrow$ extra confidence
- Covers the whole of the latest API used in practice (2.2.1)
- Exercise for KeY – performance & JAVA CARD compliance
- Study of the specs to identify hot spots

In This Talk

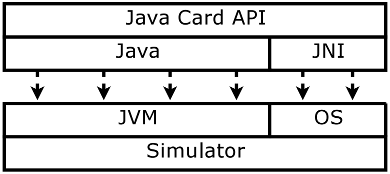
- Implementation structure
- Implementation coverage
- The choice of the specification language
- Java Card native interface in KeY
- Examples
- Experience & Discussion

Implementation Structure

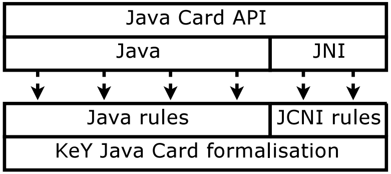
JAVA CARD API implementation: JAVA + Native Code

Native code:

- Smart card operating system
- JAVA CARD simulator interface, e.g. JCWDE
- JAVA CARD formal model, in KeY set of axiomatic rules $\longrightarrow$ full symbolic execution of the API code



Java Card Simulator



KeY Verification System

Java Card Features

- Applets
- APDUs
- AID registry
- PIN objects
- transactions
- Object sharing across firewall
- Remote Method Invocation (RMI)
- Cryptographic keys and ciphers
- Utilities

What Is Not Covered?

- Low-level APDU communication – not necessary
- Cipher logic – tedious to implement and specify, but possible (possible future work)
- RMI dispatching – on the edge of dynamic class loading, hence not possible to verify, yet not necessary
- Smart card memory consumption – verification possible for transient memory, for persistent memory need extra support from KeY

What Is Covered?

- Cryptographic routines – (almost) everything except the actual ciphering
- Lightweight AID registry
- Inter applet object sharing across the firewall
- Transaction mechanism (obviously)
- PIN objects, applets, utilities, etc.
- Applet firewall:
  - firewall checks on two levels: JVM and API
  - the API checks included in the implementation, but transparent during verification
  - JVM checks require an extension of the JAVA CARD execution model
- Official documentation followed closely

The Specification Language

- OCL. . .
- JML – suitable, but currently lack of sufficient control over the generated POs, wait for the JML front-end rewrite to complete
- Java Card DL – more tedious than JML, but:
  - full control
  - possible to take specification shortcuts, improve on verification performance
  - surprisingly almost no applicable technical limits (but few small bugs that needed fixing)
  - an almost exact JML "would-be"

Java Card Native Interface

Dedicated KeY specific class, `KeYJCSystem`:

```
public static native byte jvmIsTransient(Object theObj);
public static native byte[] jvmMakeTransientByteArray(
  short length, byte event);
public static native void jvmBeginTransaction();
```

`KeYJCSystem`: native methods and the card “operating system”

Dedicated logic rules:

```
\< transType = KeYJCSystem.jvmIsTransient(obj); ...\>...
→ {transType := obj.<transient>}\< ... \> ...
```

The actual API implementation:

```
public class JCSystem {
  public static byte isTransient(Object theObj){
    if(theObj == null) return NOT_A_TRANSIENT_OBJECT;
    return KeYJCSystem.jvmIsTransient(theObj); }}
```

Example 1 – AID.partialEquals, Implementation

```
public final boolean partialEquals(byte[] bArray,
  short offset, byte length) throws SecurityException,
    ArrayIndexOutOfBoundsException {
  if (bArray == null) return false; // resp. documentation
  if(length > _theAID.length) return false; // resp. documentation
  // Firewall check:
  if (KeYJCSystem.jvmGetContext(KeYJCSystem.jvmGetOwner(bArray))
      != KeYJCSystem.jvmGetContext(
        KeYJCSystem.jvmGetOwner(KeYJCSystem.previousActiveObject))
      && KeYJCSystem.jvmGetPrivs(bArray) != KeYJCSystem.P_GLOBAL_ARRAY)
    throw KeYJCSystem.se; // System owned singleton instance
  // Actual comparison:
  return Util.arrayCompare(bArray, offset,
    _theAID, (short)0, length)==0;
}
```

Example 1 – AID.partialEquals, Specification

```
\programVariables {AID aidInst; boolean result;
  byte[] bArray; short offset; byte length; }

(bArray != null -> length >= 0 & offset >= 0 &
  offset + length <= bArray.length)
& {\subst AID aid; aidInst}(\includeFile "AID_inv.key";)
-> \<{
      result = aidInst.partialEquals(bArray, offset, length)@AID;
   }\> (
     (bArray = null | length > aidInst._theAID.length -> result = FALSE)
  & (bArray != null & length <= aidInst._theAID.length ->
      (result = TRUE <-> \forall int i; ( i >= 0 & i < length ->
        aidInst._theAID[i] = bArray[offset+i])))
  & {\subst AID aid; aidInst}(\includeFile "AID_inv.key";))
\modifies {}
```

Example 2 – TransactionException.throwIt

Implementation:

```
public static void throwIt(short reason)
    throws TransactionException {
  _instance.setReason(reason);
  throw _instance;
}
```

Specification:

```
    (\includeFile "TransactionException_static_inv.key";)
  & {\subst TransactionException exc; TransactionException._instance}
       (\includeFile "TransactionException_inv.key";)
-> \<{#catchAll(TransactionException t) {
         TransactionException.throwIt(reason)@TransactionException;
       }}\>
  ( t = TransactionException._instance & t._reason[0] = reason
  & (\includeFile "TransactionException_static_inv.key";)
  & {\subst TransactionException exc; TransactionException._instance}
       (\includeFile "TransactionException_inv.key";))
\modifies {TransactionException._instance._reason[0] }
```

Verification

Figures:

- 60 classes, 205K Java code, 395K KeY specifications
- all methods with code verified

Interaction:

- 10 loops, 2 far from obvious (invariant depends on a logic variable created during the proof)
- otherwise practically automatic (but see next), no Simplify!
- verification purely contract based (no method in-lining) – the only way to go. . .

Total effort:

- over 2 man-months

A Loop

Removing an object from a table:

```
All non-null services[i] different
& s != null ->
\<{ int i = 0;
    while (i<maxServices) {
      if(services[i] == s) break; i++ }
    if(i != maxServices) services[i] = null;
}\> \forall int i; (i>=0 & i < maxServices
           -> services[i] != s)
```

What are:

- variant
- useful invariant

A Loop – Solution

Two cases

The object that we look for exists in the table or not:

```
\exists int i; (i>=0 & i<maxServices & services[i] = s)
```

1. Object s is not there

Variant: `maxServices - i`
Invariant:

```
i>=0 & i<=maxServices &
  (i < maxServices -> services[i] != s)
```

2. Object s is there

Variant: `i_0 - i`
Invariant: `i>=0 & i<=i_0`

A Loop – Solution

Two cases

The object that we look for exists in the table or not:

```
\exists int i; (i>=0 & i<maxServices & services[i] = s)
```

1. Object s is not there

Variant: `maxServices - i`
Invariant:

```
i>=0 & i<=maxServices &
  (i < maxServices -> services[i] != s)
```

2. Object s is there

Variant: `i_0 - i`
Invariant: `i>=0 & i<=i_0`

The Goods, The Bads, and the Surprises

Good – KeY can do it

Surprise – KeY has hidden functionality:

- it is possible to write JAVA CARD DL contracts for method constructors, despite a reported bug, just write a contract for the `<init>()` method
- it is possible to use an "at pre" array access operator (JML fails here!) – one non-rigid function solves the problem in JAVA CARD DL (Quote from Richard: "Nice trick")

Bad – one bug in the contract application mechanism that makes life difficult, now fixed for DL contracts. . .

The Goods, The Bads, and the Surprises

Bad – performance!!!

- propositional splitting rules (`orLeft`, `impLeft`) make the proofs grow out of reasonable limits – simply switching them off can reduce the proof size by a factor of 100 if not more, down sides:
  - occasional manual interaction to split the proof (when it is really needed) is required
  - with the splitting rules switched off the quantifier instantiation heuristics do not work (well)
- large specifications + large implementations choke KeY – large amount of references in the implementation cause lots of update splits, together with large terms KeY dies
- if-cascades for method binding not always a good idea
- the last method verified only last Monday

Good – quantifier instantiation heuristics work well, but. . .

The Goods,The Bads, and the Surprises

Good – non-rigid function symbols are the best

Surprise – an obvious bug in collecting implementing classes of an interface overlooked for a very long time

Surprise – “bad” programming can ease up verification!

Bad – specifications and modifies clauses for object creating methods

Bad – other known issues and bugs, like treatment of queries (the proposed solution is to use non-rigid function symbols – Philipp)

Details – Bad Programming

```
public class RSAPrivateCrtKeyImpl {
  private byte[] p;
  private byte[] q;
  private byte[] dp;
  private byte[] dq;
  private byte[] pq;
}
```

against

```
public class RSAPrivateCrtKeyImpl {
  private byte[] keyMaterial;
  private short pOff, pLen;
  private short qOff, qLen;
  private short dpOff, dpLen;
  private short dqOff, dqLen;
  private short pqOff, pqLen;

}
```

Details – Bad Programming

```
public class RSAPrivateCrtKeyImpl {
  private byte[] p;
  private byte[] q;
  private byte[] dp;
  private byte[] dq;
  private byte[] pq;
}
```

against

```
public class RSAPrivateCrtKeyImpl {
  private byte[] keyMaterial;
  // private short pOff, pLen;
  // private short qOff, qLen;
  // private short dpOff, dpLen;
  // private short dqOff, dqLen;
  // private short pqOff, pqLen;
  private short size;
}
```

Details – Splitting Specifications

Precondition:

```
apduState = STATE1 | apduState = STATE2 | apduState = STATE3 ...
```

Possible to use a function symbol with suitable unfolding rule:

```
validAPDUState(apduState)
```

Why not e.g. wrap all invariants inside non-rigid functions: more readable, smaller formulas, . . .

Obstacle: when done manually, the taclet mechanism stands in the way a bit – not possible to use concrete attribute and class names in taclets

Experimented here a bit – this can potentially help, but there are downsides, e.g. no early closing of goals

Details – Type if-cascades

```
InterfaceType o= ...;
o.methodCall();
```

results in:

```
if(o instanceof Type1) {o.methodCall()@Type1;
}else if (o instanceof Type2) {o.methodCall()@Type2;
}else ...
```

All types are subtypes of some `SuperType` which is the only implementor of `methodCall`

The same method body will be expanded, or the same contract used, X times. This is a total waste! In the Java Card API there are 18 classes implementing the `Key` interface with one common base class.

Moreover, what if I want to use a method contract for the interface itself? Then the if-cascade is not necessary at all!

Details – Methods Creating Objects

```
public static byte[] makeTransientByteArray(...
```

In JML it is sufficient to say:

```
ensures \fresh(\result);
```

In DL however:

```
result != null & result.<created> = TRUE &
result.<transient> = ... &
\forall int i; (i>=0 & i<result.length -> result[i] = 0) &
result = jbyte[]::<get>(jbyte[]::<nextToCreate>@pre) &
jbyte[].<nextToCreate> = jbyte[]::<nextToCreate>@pre + 1

\modifies {
  jbyte[].<nextToCreate>,
  jbyte[]::<get>(jbyte[].<nextToCreate>).<created>,
  jbyte[]::<get>(jbyte[].<nextToCreate>).<transient>,
  jbyte[]::<get>(jbyte[].<nextToCreate>).length,
  jbyte[]::<get>(jbyte[].<nextToCreate>)[*] }
```

Details – Methods Creating Objects

If the method allocates more than one object this gets out of hand! Moreover, the user has to know in which order the objects are created if they are of the same type.

Is there an easy way to reflect JML `\fresh` in JAVA CARD DL?

Details – Propositional Splitting

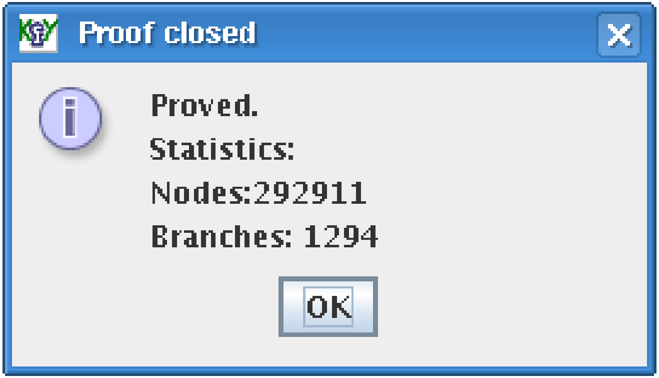
`orLeft` and `impLeft` (also `andRight`) can make things really bad:

This is with no splitting (?!) Multiply this by 100 (at least) and you get the idea. . .

Would simply postponing the splitting do the trick, so that the program is executed first on a single branch?

Details – Propositional Splitting

`orLeft` and `impLeft` (also `andRight`) can make things really bad:



Proved.

Statistics:

Nodes:292911

Branches: 1294

This is with no splitting (?!) Multiply this by 100 (at least) and you get the idea. . .

Would simply postponing the splitting do the trick, so that the program is executed first on a single branch?

Future Work

- Recreate this in KeYJML once the new front-end is available
- Clean up and optimise the code
- Put it up on the web
- Implement the firewall formalisation and verify again (any one interested?)
- Deal with JAVA CARD memory consumption
- Implement the cipher logic in an abstract way in terms of non-rigid function symbols