Specification of a transacted memory for smart cards in Java and JML

Erik Poll

University of Nijmegen, NL

Joint work with

Pieter Hartel
Eduard de Jong

University of Twente
Sun Microsystems
Case study in the use of Java and the specification language JML to implement/model/specify part of a smartcard OS.

Reasoning about JavaCard code subject to card tears.
Background
A smart card is a miniature computer with
  ● with limited resources (ROM, RAM, EEPROM)
  ● very limited I/O (ISO7816)

Smartcard contains a miniature operating system (OS).

JavaCard smartcard contains VM that can execute applets.
Superset of a subset of Java for programming smart cards

- no threads, floats, . . . , very limited API

but

- persistent and transient objects (EEPROM and RAM)
- transaction mechanism

and increased security:

- standard sandbox + firewall between applets.

Interesting target for formal methods: small programs, simple language, correctness of crucial importance
Tricky issue for smartcards:

- Possible power loss at any moment due to card tear
- OS must support transactions:
  atomic writes consisting of several EEPROM writes
- On power-up: OS performs clean-up of any unfinished transaction
- NB this clean-up can be interrupted by a card tear . . .
Challenges

1. How to implement a transaction mechanism?
   Smartcard constraints
   - limited RAM (eg 512 bytes)
   - EEPROM behaviour:
     - atomic write of given word size
     - limited life

2. How to specify and verify – and model – this?

3. How to specify and verify code that uses transactions?
An implementation of Transacted Memory
Idea for transacted memory by de Jong & Bos. NB not as implemented in the JavaCard API.

Logging for free!
**Interface**

<table>
<thead>
<tr>
<th>Tag</th>
<th>DNewTag(Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfoSeq</td>
<td>Read(Tag)</td>
</tr>
<tr>
<td>void</td>
<td>Write(Tag, InfoSeq)</td>
</tr>
<tr>
<td>void</td>
<td>Commit(Tag)</td>
</tr>
<tr>
<td>InfoSeq</td>
<td>ReadGeneration(Tag, Gen)</td>
</tr>
<tr>
<td>void</td>
<td>Tidy()</td>
</tr>
</tbody>
</table>
Earlier work

Deficiencies:

- Z specs do not include card tears
- Big gap – and no real link – between Z specs and implementation
This work

Initial idea:

- Translate C code to Java
  By hand; easy but boring.
- Translate Z specs to JML
  By hand; easy but boring. Uses JML
- Tie the two together
Java Modeling Language JML

Specification language by Gary Leavens (Iowa Univ.) for annotating Java programs with

- pre- and postconditions
- invariants
- frame conditions (modifiability constraints)
- specification-only variables (model/ghost variables)
- ...

Pre-, postconditions, and invariants in JML are Java boolean expressions, extended with $\forall$, $\exists$, $\Rightarrow$, $old(\ )$, ...
Translating abstract Z spec
\[\Delta \text{MemSys} \]
\[t? : \text{tags} \]
\[t? \in \text{dom assoc} \]
\[\text{assoc } t? \neq \langle \rangle \]
\[\text{committed'} = \text{committed} \cup \{t?\}\]
Using JML library for sets, functions, relations, etc. Z specs can be translated to JML

```java
public void ACommit(Tag t)
/*@ requires assocs.domain().has(t) &&
  @     ! assocs.apply(t).isEmpty() ;
  @ ensures committed.equals(
  @     \old(committed).insert(t));
  @*/
```
Z vs JML:

- Z looks much prettier
- JML distinguishes pre- and postcondition
- JML can be easily made executable
Modelling card tears in Java

- *card tear* is like an *exception*
- *clean-up* is like the *exception handler*
- *card tear* is uncatchable exception, caught only in the main repetition of the OS
- Modelling card tear inside language, allows testing, specification, and verification
annotated Java implementation

```java
/*@ ensures Read(tag).equals(is);
@ signals (CardTearException)
@     TidyRead(tag).equals(is)
@     || TidyRead(tag).equals(\old(Read(tag)));
@*/

public void Write (Tag tag, InfoSeq is)
    throws CardTearException

Expresses atomicity of write operation
Specs still incomplete: nothing said about previous generations
JML assertion checker can cope with this.
```
Java impl with JML assertions

Bugs found:

- one typo - giving ‘version number’ instead of ‘generation number’
  Found during typechecking Java code

- clumsy interface - 4 write operations with disjoint preconditions
  Found writing JML assertions

- one serious error - card tear at certain point
  Found using runtime assertion testing
Java impl with JML specs

By translating

1. abstract Z spec to (executable) JML spec
2. C implementation to Java implementation

abstract spec & implementation in same language. We can tie them together, by

- running same test scenario on 1. and 2.
- including 1. in 2. using ghost/model variables

Conventional programming language (Java) useful to built formal model (of VHDL implementation).
Using Transacted Memory
Current JavaCard API offers transactions

```
beginTransaction();
beginTransaction();
abortTransaction();
```
The LOOP project

Verification of JML-annotated Java(Card) programs based on

- a denotational semantics for sequential Java,
- a compiler – the LOOP tool – which translates A.java to A.pvs describing its semantics.
- a Hoare logic for reasoning about JML,
- associated WP calculus,

All formalised in PVS: ie. a shallow embedding of Java and JML in PVS

How to allow for card tears?
Trick to model card tear as an exception also work when specifying and verifying Java Card code.

For example

```java
/*@ ensures \old(x+y) == x+y;
@ signals (CardTearException) \old(x+y) <= x+y;
@*/
void bla(){
  x++ ; y--; // This line could lead to a card tear
}
```
Java:
Invariant may temporarily be violated, but must hold at end of method - *also if an exception is thrown*

JavaCard:
Invariant may never be violated, except during transactions
beginTransaction();
x++ ; y--;
endTransaction();

**can be desugared into**

\[
x' = x; 
\]
\[
y' = y; 
\]

try {
x++ ; y--;
}
catch (CardException e) {
x=x'; y=y';
throw e;
}

Transacted Memory - Erik Poll – p.28/30
Limits of this approach

It can’t deal with

```c
void mn() {
    m(); n();
}
void m() {
    beginTransaction();
    x++
}
void n() {
    y--;
    endTransaction();
}
```
Alternative approach: including transactions in denotational semantics
Easy enough, but coming up with associated proof rules isn’t.
Conclusions

Conventional programming language (Java) maybe an interesting formal model

Future work:

- fixing bug
- verification of Transacted Memory using PVS & LOOP coping with model variables in LOOP
- verification of applets incl. card tears
- VHDL implementation

Worry: ensuring C code = Java code = VHDL code