Protocol Specification: specifying the JavaCard APDU protocol in JML

Erik Poll

ITT colloquium, 6 Oct 2000

The JavaCard platform

JavaCard platform (JCRE)	=	JavaCard language (JCVM)	+	JavaCard API

The JavaCard API provides

- some base classes and interfaces, eg. `Applet`
- some OS-like functionality, including the `APDU` class for communication with the smartcard terminal.

ISO 7816 and APDU's

Smartcard and terminal communicate by exchanging APDU's – sequences of bytes – as specified in ISO7816-4:

1. terminal sends a command APDU to smartcard

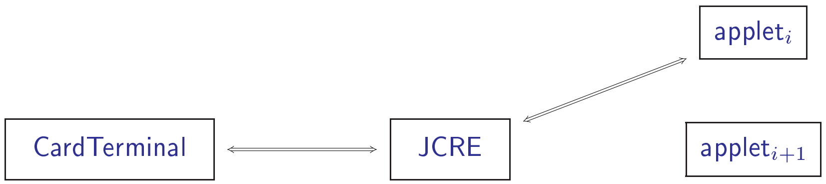
CLA	INS	P1	P2	LE	D_1	...	D_{LE}

2. smartcard sends a response APDU back to terminal
3. back to 1.

Two variants, T=0 and T=1: byte and block transmission

ISO 7816 and APDU's for applets

The JCRE mediates between applets and terminal:



JCRE passes an APDU-object to the selected applet, by invoking its `process(apdu)` method.

An APDU-object is essentially a buffer, with methods for reading/writing/etc. in it.

The APDU class

So an applet receives an APDU, on which it can invoke

```
public static byte[] getBytes()
public static short getInBlockSize()
public static short getOutBlockSize()
```

and

```
public short setIncomingAndReceive()
public short receiveBytes(short bOff)
public short setOutgoing()
 public void setOutgoingLength(short len)
 public void sendBytes(short bOff, short len)
  ⋮
```

in a certain order!

Informal (javadoc) spec

receiveBytes

`public short receiveBytes(short bOff) throws APDUException`

Gets as many data bytes as will fit without APDU buffer overflow, at the specified offset bOff. Gets all the remaining bytes if they fit.

Parameters: `bOff` - the offset into APDU buffer.
Returns: number of bytes read. Returns 0 if no bytes are available.
Throws: APDUException - with the following reason codes:

- `APDUException.ILLEGAL_USE` if `setIncomingAndReceive()` not called or if `setOutgoing()` or `setOutgoingNoChaining()` previously invoked.
- `APDUException.BUFFER_BOUNDS` if not enough buffer space for incoming block size.
- `APDUException.T1_IFD_ABORT` if T=1 protocol is in use and the CAD sends in an ABORT S-Block command to abort the data transfer.

Reference implementation

The reference implementation of `APDU` uses 7 flags

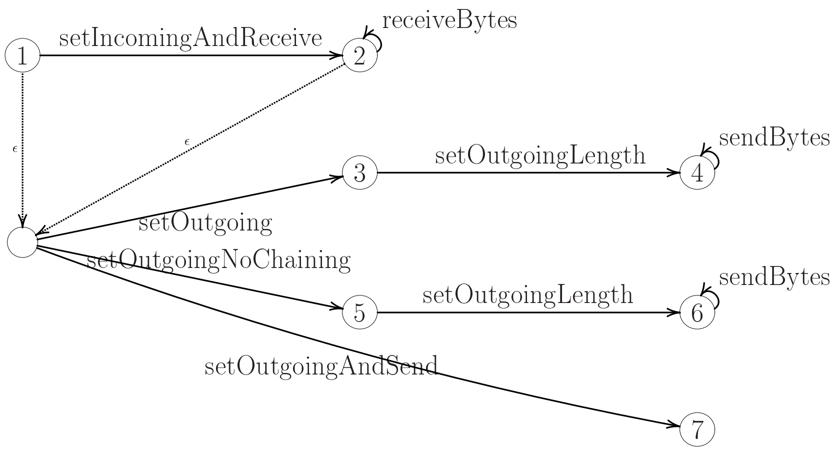
```
incomingFlag, outgoingFlag, outgoingLenSetFlag,
lrIs256Flag, sendInProgressFlag, noChainingFlag,
noGetResponseFlag
```

to enforce invocation order, eg.

```
public short receiveBytes(short bOff) throws APDUException
{ if (!getIncomingFlag() || getOutgoingFlag() )
      APDUException.throwIt( APDUException.ILLEGAL_USE );
   ...
```

but protocol has $<< 2^7$ states !

FSM for APDU



Using a model field and FSM to specify APDU

```
//@ public model int _APDU_state;

public short setIncomingAndReceive()
 /*@ public behavior
   @    requires: _APDU_state == 1 && ... ;
   @     ensures: _APDU_state == 2 && ... ;
   @*/


public short receiveBytes(short bOff)
 /*@ public behavior
   @    requires: _APDU_state == 2 && ... ;
   @     ensures: _APDU_state == 2 && ... ;
   @*/
```


Relating reference implementation to formal spec

Invariants relating the abstract state to its concrete representation, eg:

```
/*@ private invariant:
  @       _APDU_state == 2
  @  <==>
  @         getIncomingFlag() && !getOutgoingFlag();
  @*/
```


More detailed JML spec of receiveBytes(short bOff)

```
/*@ requires: _APDU_state == 2                              &&
  @           0 <= bOff                                     &&
  @           bOff + getInBlockSize() <= BUFFERSIZE;
  @
  @ modifiable: _APDU_state, _Lc, buffer[bOff..bOff+\result-1];
  @
  @  ensures: _APDU_state == 2                              &&
  @           0 <= \result && \result <= \old(_Lc)          &&
  @           _Lc == \old(_Lc) - \result                    &&
  @           bOff + \result <= BUFFERSIZE                  &&
  @           (* data received in buffer[bOff..bOff+\result-1] *);
  @
  @  signals: (APDUException e) e.getReason() == APDUException.IO_ERROR
  @                          || e.getReason() == APDUException.T1_IFD_ABORT;
  @*/
```

Here model field _Lc is the length of incoming command.

Relating reference implementation to formal spec

Length of the incoming command in JML spec:

```
//@ public model int _Lc;
//@ public invariant: 0 <= _Lc && _Lc < 256;
```

Representation in the reference implementation:

```
private byte getLc()
 { return ramVars[LC]; }
```

NB. byte-int conversion yields getLc() $\in [-128..127]$. So

```
//@ private invariant: _Lc == (getLc()&0xFF)
```

and **not** `_Lc == getLc()`.

Bug in reference impl. of `receiveBytes`

The reference implementation does NOT meet this spec, but requires a stronger precondition than

```
bOff + getInBlockSize() <= BUFFERSIZE,
```

namely

```
bOff + getInBlockSize() <  BUFFERSIZE.
```

This is probably a bug.

Conclusions

- (Still incomplete) formal specs for APDU protocol:
  400 lines code, 400 lines javadoc, 170 lines public JML spec
- State transitition diagram nice way to specify the APDU protocol.
  Why isn't it used anywhere in the existing documentation ??
  (Alternative approach in Jass: specifying traces of invocation sequences)
- Advantage of formal JML specs: clear and unambiguous
  eg. "`bOff + getInBlockSize() <= BUFFERSIZE`"
  instead of "enough buffer space for incoming block size"
- Whenever possible, our specs do not say "exception E_i is thrown, if P_i holds",
  but insist on "$\neg P1 \&\& \ldots \&\& \neg P_n$" as precondition.

Future work & questions

- To do: $\left.\begin{array}{l}\textbf{testing}\\ \textbf{checking}\\ \textbf{proving}\end{array}\right\}$ these specs

? More complete specs, eg incl. formalisation of

```
... && (* data received in buffer[bOff..bOff+\result-1] *)
```

? API does not assume any terminal-side behaviour

? asynchronous `sendBytes` hard to specify