Source Code Verification of a Secure Payment Applet

Bart Jacobs, Martijn Oostdijk, Martijn Warnier

Nijmegen Institute for Information and Computing Sciences,
University of Nijmegen, P.O. Box 9010, 6500GL, Nijmegen, The Netherlands

Abstract

This paper discusses a case study in formal verification and development of secure smart card applications. An elementary Java Card electronic purse applet is presented whose specification can be simply formulated as "in normal operation, the applet's balance field can only be decreased, never increased". The applet features a challenge-response mechanism which allows legitimate terminals to increase the balance by putting the applet into a special operation mode. A systematic approach is used to guarantee a secure flow of control within the applet: appropriate transition properties are first formalized as a finite state machine, then incorporated in the specification, and finally formally verified using the Loop translation tool and the PVS theorem prover.

1 Introduction

Reasoning about the security of software systems can, in general, be done on two distinct levels. The first level is the high abstract level of security protocols. On this level different principals are distinguished that communicate with each other by sending abstract messages over some untrusted network. Cryptographic primitives such as random nonce generation, one-way hashes, encryption, signatures, etc. are used to construct these messages. Security protocols are designed to establish high level properties such as authenticity, confidentiality and data integrity. Formal methods such as BAN logic [6] (applied to payment protocols in [1]), or state space exploration based methods such as Casper/FDR [19] have been developed to help make the analysis on this level more rigorous and efficient.

Email addresses: `bart@cs.kun.nl` (Bart Jacobs), `martijno@cs.kun.nl` (Martijn Oostdijk), `warnier@cs.kun.nl` (Martijn Warnier).

Preprint submitted to Elsevier Science *7 April 2003*

The second level is the low concrete level of program code (in the case of this paper Java Card [7] source code). Specifications on this level usually consist of pre- and postconditions for methods, and invariants or history constraints for a class as a whole. It is generally acknowledged that there is a substantial gap between these high and low level approaches, and that bridging this gap is very important. In this paper we shall try to narrow this gap from below. We choose to take this lowest level as starting point because:

- Certification of specific, security sensitive products is expected to focus primarily on the actual implementation, and not so much on a high level abstract description of the system;
- Refinement is a problematic notion within the area of computer security: it may well happen that by adding implementation details certain key security properties that can be established at a high abstraction level no longer hold at a low level (because an attacker has more material to exploit).

Within the setting of this paper we study a simple payment system that can be realized with (Java) smart cards, running a specific applet. The applet involves a balance, which represents an amount of electronic money. In general with such payment systems, there are separate protocols for paying (decreasing the balance), and for charging (increasing the balance). Our payment protocol is not very interesting, because it is extremely simple, and does not involve any form of authentication. Charging however is less trivial, because it involves a challenge-response mechanism in which the terminal needs to authenticate itself, and in which at most five tries are permitted. The smart card applet that we have developed involves an implementation of this protocol for charging.

Our aim in this paper is to push our current program specification and verification technology as far as possible for proving the correctness of the payment applet in general, and the charging protocol in particular. The restriction to current verification technology is very relevant, and excludes for instance specification language extensions such as in [5]. As a result the cryptographic aspects in specification and verification are not dealt with at all. But our approach works well for the control-flow aspects of the charging protocol. What we achieve here may be seen as first steps towards an applet development methodology, consisting of:

- Describing the control flow (of the applet-part of the protocol) as a finite state machine;
- Translating the transitions of this finite state machine into a class constraint, involving a special "`mode`" variable capturing the states.

Elaborating the details of this methodology forms the main contribution of this paper.

Throughout this paper the emerging standard specification language for Java, the Java Modeling Language (JML) [17,18], is employed for the specification of program properties on this level. JML assertions are written in special comment tags in order not to confuse the Java compiler. They can serve as precise formal documentation that can actually be verified (in contrast to Javadoc). The syntax of the JML assertions themselves is quite similar to that of Java's Boolean expressions. Both normal and abnormal postconditions can be specified for methods. A growing collection of more or less compatible tools is becoming available for JML, for instance the JML runtime checker [17], ESC/Java [9], Daikon [8], Krakatoa [25], and our own tool: the Loop tool [16].

The general approach taken in this paper is bottom-up. Rather than attempting to specify and verify a fully featured industrial payment application, we describe a Java Card applet of our own creation with minimal functionality, but whose correctness is formally proved. The applet features some cryptography, which is traditionally thought of as hard to specify on the program code level and which is therefore usually specified on the security protocol level. Therefore it plays only a minor role in our verification effort.

It should be emphasized that the Java code in this paper is our own. It is written for the purpose of verification. It has been compiled and even loaded onto a smart card, but it has—deliberately—not been tested [1]. We wanted to see what kind of errors could be discovered with our verification technology alone, without actually running the code. And we actually found several program bugs, see Section 3. Yet, most of the mistakes occurred in the specifications we wrote. Once again it became clear that writing good specifications is much harder than writing good programs. But verifying the specifications is still the most difficult part of all. However, it is this effort that leads to the greatest level of confidence: knowing that one's implementation satisfies the specification.

The remainder of this paper is organized as follows. Section 2 describes how the specification of the Java Card applet came about. It also introduces some of the JML notation. Section 3 sheds some light on different aspects of the formal proof process. Section 4 draws some conclusions and discusses possible future work.

2 Specifying the Applet

An electronic purse applet should, at the very least, contain a `balance` field, say of type `short`. After card issuance it should be possible to decrease `balance` and to consult its current value. Initially `balance` is set to zero,

[1] Indeed, there is no 'select' case in our `process` methods below.

but the card can be recharged by certain terminals. This means the applet needs a `key` field as well. The key is supposed to be loaded onto the card before the card is issued. A symmetric key is used which is shared by any legitimate terminal[2]. The card authenticates a terminal by sending a challenge. Only a terminal equipped with appropriate credentials can respond to this challenge by sending it back encrypted with the key. The applet keeps track of the number of consecutive unsuccessfully answered challenges in a `counter` field. As soon as `counter` reaches five, the card locks itself.

To make the verification of the applet a little bit more challenging, we restrict `balance` to 12 bit signed values, to be interpreted as the number of euro cents on the card. This means that the maximum amount on the card is 40 euros and 96 cents, which seems reasonable for a simple electronic purse. It also means that the applet's source code will contain several bitwise operations such as shifts and masking. The Loop tool has support for verification of such operations, though this is still in a developmental stage.

Smart cards are instructed to perform certain operations through so-called APDUs (Application Protocol Data Units). Within a Java Card applet the `process` method is responsible for managing the incoming command APDUs once the applet has been selected. Every command APDU contains an instruction byte which is examined by the `process` method. The applet will throw an `ISOException` whenever the terminal sends a command that is not understood or not appropriate in the applet's current state. Exceptions show up on the terminal side as special return codes in the response APDU.

Instruction bytes for the electronic purse applet described above are encoded inside the applet as follows:

- `INS_SETKEY` sets the 56 bit DES (Data Encryption Standard) key. The data field should contain the eight bytes of key data.
- `INS_GETVAL` returns the value of `balance`. The response APDU that is to be expected contains a data field of two bytes with the current balance.
- `INS_DECVAL` decreases the value of `balance` with 1 cent.
- `INS_GETCHAL` asks the applet to generate a challenge (a random nonce) and send it back to the terminal application. The response APDU that is to be expected contains the 32 byte nonce. The applet increases the `counter` field for each requested challenge.
- `INS_RESPOND` responds to the card challenge by sending a 24 byte encrypted hash of the nonce. The APDU also includes a new value for `balance` in the last two bytes of the decrypted cipher text array. A successful response to a challenge resets the `counter` field.

The challenge-response protocol implemented by the last two commands can

[2] This may very well be a diversified key, but that is irrelevant for the applet itself.

be formulated in standard security protocol notation as:

$$C \longrightarrow T : \mathsf{Nonce}$$
$$T \longrightarrow C : \{\mathsf{Hash}(\mathsf{Nonce}), \mathsf{NewValue}\}_K$$

where C stands for the card, T for the terminal, and K for the shared key. If the response from the terminal checks out, the current value of `balance` is replaced by NewValue. Note that since the new value is also encrypted, a replay or a man-in-the-middle attack is not possible as an attacker would have to know K to change NewValue. Even though this is a very basic authentication protocol, we checked the absence of such attacks by using the security protocol compiler Casper [19] in combination with the model checker FDR [10]. It presented no problems.

Not every command is always appropriate. For instance, APDUs with the `INS_SETKEY` instruction byte should no longer be taken seriously by the applet once the card has been issued. The applet's current state is made explicit in the implementation by using a dedicated field called `mode` with values for `INIT`, `ISSUED`, `CHARGING`, and `LOCKED`. Initially `mode` equals `INIT`. Once the key has been set, `mode` is changed to `ISSUED`. As soon as the terminal requests a challenge, `mode` is switched to `CHARGING`. The subsequent command will switch `mode` back to `ISSUED`. When five or more challenges are left unanswered, `mode` becomes `LOCKED` and the applet will no longer respond to commands.

The `mode` field implements a rudimentary lifecycle model comparable to those described for example in [20] and [11]. Usually such a model identifies different standard stages of an applet's life, for example: *Installation*, *Personalization*, *Processing*, and *Locked*. Of course, for concrete applets the *Processing* life stage can usually be refined into more detailed stages. Sending specific commands to the applet may result in a change of its state. This gives rise to the finite state machine in Figure 1.

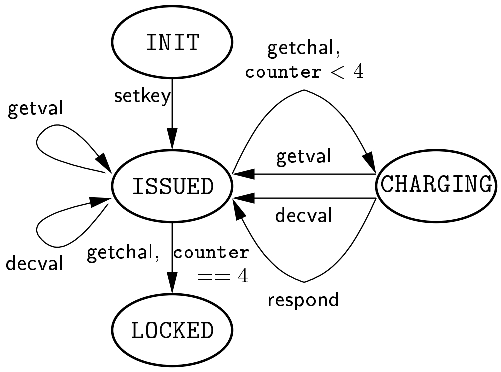

Fig. 1. A finite state machine describing the lifecycle model of the applet.

Such an explicit representation of the applet's state makes formulating global properties much easier when writing the specification. In fact, some of the applet's code and specification can be systematically derived from the finite state machine. In the implementation of the `process` method we created dedicated helper methods for each of the possible instruction bytes. These helper methods are only called from the `process` method, which consists of two levels of nested `switch` statements. The outermost `switch` performs a case analysis on the `mode` field, while the innermost `switch`es examine the instruction byte to call the corresponding helper method.

```
/*@ behavior
  @    requires apdu != null && random != null
  @              && cipher != null && digest != null;
  @  modifiable mode, balance, counter, apdu.buffer[*],
  @              key[*], cipher, nonce[*];
  @     ensures true;
  @     signals (ISOException) true;
  @*/
public void process(APDU apdu) throws ISOException {
  byte ins = apdu.getBuffer()[OFFSET_INS];
  switch(mode) {
    case INIT:
      switch(ins) {
        case INS_SETKEY: setKey(apdu); break;
        default: ISOException.throwIt(
                      SW_CONDITIONS_NOT_SATISFIED);
      }; break;
    case ISSUED:
      switch(ins) {
        case INS_GETVAL: getValue(apdu); break;
        case INS_DECVAL: decValue(apdu); break;
        case INS_GETCHAL: getChallenge(apdu); break;
        default: ISOException.throwIt(
                      SW_CONDITIONS_NOT_SATISFIED);
      }; break;
    case CHARGING:
      switch(ins) {
        case INS_GETVAL: getValue(apdu); break;
        case INS_DECVAL: decValue(apdu); break;
        case INS_RESPOND: respond(apdu); break;
        default: ISOException.throwIt(
                      SW_CONDITIONS_NOT_SATISFIED);
      }; break;
    case LOCKED: ISOException.throwIt(
                      SW_SECURITY_STATUS_NOT_SATISFIED);
    default: ISOException.throwIt(
                  SW_CONDITIONS_NOT_SATISFIED);
    }
}
```

Because the high level properties of the applet are captured in global invariants and constraints, the specification of the `process` method itself is not terribly interesting. The local specifications (in terms of pre- and postconditions) of the dedicated methods capture the more detailed functional behavior of those methods. An example of such a method specification is given in Section 3 for the methods dealing with APDUs whose instruction byte is `INS_GETCHAL` or `INS_RESPOND`. Those specifications exactly describe the behavior of the `counter` field as a result of the method invocations. The global properties are stated in the class invariant and class constraint. The class invariant reads as follows:

```
/*@ invariant
  @ (mode == INIT || mode == ISSUED ||
  @            mode == CHARGING || mode == LOCKED)
  @   && 0 <= counter && counter <= 5
  @   && (mode == LOCKED <==> counter == 5)
  @   && (mode == INIT ==> counter == 0)
  @   && 0 <= balance && balance <= 4096
  @   && key != null && key.length == DES_KEY_SIZE
  @   && nonce != null && nonce.length == NONCE_SIZE
  @   && sha_nonce != null && sha_nonce.length == SHA_SIZE
  @   && plaintxt != null && plaintxt.length == SHA_SIZE+2
  @   && ciphertxt != null && ciphertxt.length == CIPHERTEXT_SIZE;
  @*/
```

The first four conjuncts are the most interesting. They express that the applet can be only in one of four states (`INIT`, `ISSUED`, `CHARGING` or `LOCKED`); that the `counter` that keeps track of the number of unsuccessful challenge-responses is at most five; that the state `LOCKED` corresponds to the counter having the maximum value five; and that in the `INIT` mode the `counter` equals zero. Such invariant properties typically make explicit what the programmer has in the back of his/her mind.

The class constraint conveys more interesting information as it captures the intended flow of control. A constraint expresses a relation between the pre-state (indicated by `\old`) and the post-state, which should hold for all method invocations. Part of the constraint is generated automatically using the prototype *F2J* tool described in [13]. To emphasize that part of the constraint is generated automatically we list two constraints. In the actual verification of the applet these are simply conjuncted. The eight parts of the generated constraint describe the possible transitions from the different states. Note that sending a command that is not supported by the applet does not cause a change of state. This induces reflexive transitions from each state to itself, which are not explicitly specified in Figure 1.

```
/*@ constraint
```

```
  @    (mode == LOCKED ==> \old(mode) == ISSUED
  @                        || \old(mode) == LOCKED) &&
  @    (mode == INIT ==> \old(mode) == INIT) &&
  @    (mode == ISSUED ==> \old(mode) == INIT
  @                        || \old(mode) == ISSUED
  @                        || \old(mode) == CHARGING) &&
  @    (mode == CHARGING ==> \old(mode) == ISSUED
  @                        || \old(mode) == CHARGING) &&
  @    (\old(mode) == LOCKED ==> mode == LOCKED) &&
  @    (\old(mode) == INIT ==> mode == ISSUED
  @                        || mode == INIT) &&
  @    (\old(mode) == ISSUED ==> mode == ISSUED
  @                        || mode == CHARGING
  @                        || mode == LOCKED) &&
  @    (\old(mode) == CHARGING ==> mode == ISSUED
  @                        || mode == CHARGING);
  @*/
```

The second constraint was manually entered and captures the relation between balance, counter and mode.

```
/*@ constraint
  @    (\old(mode) == LOCKED ==>
  @       (balance == \old(balance) &&
  @        counter == \old(counter) &&
  @        mode == LOCKED))         &&
  @    (\old(mode) == ISSUED ==>
  @       ((mode == ISSUED && balance <= \old(balance) &&
  @          counter == \old(counter))
  @       || (mode == CHARGING && balance == \old(balance) &&
  @          counter == \old(counter + 1) && \old(counter) < 4)
  @       || (mode == LOCKED && balance == \old(balance) &&
  @          counter == \old(counter + 1) && \old(counter) == 4))) &&
  @    (\old(mode) == CHARGING ==>
  @       ((mode == ISSUED && counter == 0)
  @       || (mode == ISSUED && counter == \old(counter)
  @                         && balance <= \old(balance))));
  @*/
```

The first conjunct expresses that once the applet becomes `LOCKED`, it will remain `LOCKED` and the balance and counter do not change anymore. The second one expresses that three things can happen in the normal state of operation `ISSUED`: the applet can stay in the same state, in which case the balance can only decrease—a key security property—and the counter will not change; the applet can go into the `CHARGING` mode, then the balance will not change, the counter will increase with one, and the value of the counter in the pre state was less then four; the applet will become `LOCKED`, in this case the balance will stay the same, the counter will again increase with one and in the pre state the counter field had value four. The latter constraint describes of course a crucial safety property for a payment applet [3] . The third conjunct expresses the two possible transitions from `CHARGING` to `ISSUED`. Either the response was valid, in which case the counter is reset to zero, or the response was invalid, in which case the counter remains unchanged and the balance does not increase.

We want to stress that in this paper we only look at the card-side of the whole payment protocol. For the terminal-side one can also consider an appropriate finite state machine, possibly also with code verification.

3 Correctness of the Applet specification

Once we have a (JML) specification of the intended behavior of our applet, we want to make sure that it actually holds for the given (Java) implementation. This is done by first translating the Java + JML code into the language of a theorem prover, via the Loop tool [2]. Next, the theorem prover is used for (interactive) verification. This approach has already proven itself in several case studies [23,3,24] mostly involving API classes. This paper presents the first application to a non-trivial smart card applet. The actual verification in the theorem prover PVS [22] proceeds via a special Hoare logic for JML [14], in combination with a recently developed weakest precondition calculus [15].

Discussing the verification of the whole applet would present too many details. Instead we concentrate on the `process` method, described in the previous section, and on two additional helper methods that are called by `process`, namely `getChallenge` and `respond`, see below.

The most important properties to be verified are the class-wide invariant and constraint properties. They express key security properties that should be established by all methods. In the semantics of JML—as incorporated in the Loop tool—invariants are implicitly added to all preconditions, and to all post-conditions, both for normal and abnormal termination. Similarly, constraints are added to all postconditions. At first sight it might seem easy to establish the trivial (normal and abnormal) postcondition `true` of the `process` method. But in reality, these postconditions are far from trivial because they involve the invariant and constraint.

The verification of the `process` method is essentially straightforward, and

[3] The observant reader might notice that some of the properties specified in the invariant and constraint are redundant. This is done deliberately, since we believe this makes the specification easier to read and thus understand.

involves making the various case distinctions in PVS, following the nested `switch` statements inside the `process` body [4]. In each case an appropriate helper method is called. During verification of the `process` method, we then *use* the specification of the method that is called. This means that the method's precondition must be established, so that its postcondition can be used in the remainder of the proof.

The two most prominent helper methods are `getChallenge` and `respond`. The `getChallenge` method is only called when `mode` is `ISSUED` and the incoming APDU contains the `INS_GETCHAL` instruction. It provides a new, randomly generated nonce, to be used in the challenge-response mechanism for authentication. For security reasons, we keep track of how many times the `getChallenge` method is called until a challenge is answered by an appropriate response. Technically, we use a private byte field `counter` which is incremented with every call to `getChallenge` as explained in Section 2. Thus, our `getChallenge` method must inspect the value of this `counter`. It does so in the following way.

```
/*@ behavior
  @    requires mode == ISSUED
  @               && counter < 5
  @               && apdu != null
  @               && apdu.buffer != null
  @               && apdu.buffer.length >=
  @                    OFFSET_CDATA + NONCE_SIZE
  @               && random != null;
  @  modifiable mode, counter, nonce[*], apdu.buffer[*];
  @     ensures mode == CHARGING
  @               && counter == \old(counter) + 1
  @               && counter < 5;
  @     signals (ISOException) mode == LOCKED && counter == 5;
  @*/
private void getChallenge(APDU apdu) throws ISOException {
  counter++;
  if (counter < 5) {
     mode = CHARGING;
     random.generateData(nonce,(short)0,NONCE_SIZE);
     apdu.setOutgoing();
     apdu.setOutgoingLength(NONCE_SIZE);
     byte[] buffer = apdu.getBuffer();
     Util.arrayCopy(
       nonce,(short)0,buffer,OFFSET_CDATA,NONCE_SIZE);
   } else {
```

[4] It was at this stage that we found a serious program bug: in our first version, the outer `break` statements after the three inner switches in `process` were not there. But the inner breaks only break out of the inner switches, not out of the outer one!

```
      mode = LOCKED;
      ISOException.throwIt(SW_SECURITY_STATUS_NOT_SATISFIED);
    }
  }
```

Notice that the precondition contains certain requirements about the length of the buffer of the incoming APDU. These requirements are actually redundant, because of the invariant that we use for the `APDU` class. The important point is that the `getChallenge` method only generates a challenge if the counter (in the pre-state) is less than 4. Recall that the invariant says $0 \leq$ `counter` ≤ 5. If `getChallenge` is called with `counter` equal to 4, the applet will become locked. And the `getChallenge` method cannot be called by `process` with `counter` equal to 5, because according to the constraint this means that the applet is already locked. We thus see the importance of the invariant and constraint in determining the appropriate flow of control.

We briefly discuss the verification of `getChallenge`. There are three points worth mentioning.

- Shortly after entering the method body, the counter is incremented and so the class invariant gets broken: the logical equivalence `mode == LOCKED <==> counter == 5` need not hold anymore. But at the end of the method the invariant is re-established. Our semantics and proof methods can handle such temporary violations of invariants.

  Re-establishing the class invariant at the end of the method body is sufficient for Java programs. For Java Card applets, however, a card-tear is always possible. Withdrawing the card from the terminal in the middle of the execution of this method could leave the applet in a state in which the invariant no longer holds. A solution to this is to use Java Card's transaction mechanism. Unfortunately, verification of transaction blocks is not supported by the Loop tool yet.
- The methods that occur in the `then` part are handled via their specifications. These specifications are fairly weak, and do not express any functional behavior. For instance, the specification for the `RandomData` method `generateData` looks as follows.

  ```
  /*@ normal_behavior
    @    requires buffer != null && offset >= 0
    @       && length >= 0 && offset + length <= buffer.length;
    @  modifiable buffer[*];
    @     ensures true;
    @*/
  public abstract void generateData(
               byte[] buffer, short offset, short length);
  ```

  This serves our purposes. We use similarly weak specifications for the `APDU` methods `setOutgoing` and `setOutgoingLength`. But the `Util` method `arrayCopy` has a functional specification—like in [3].

- The two methods `setOutgoing` and `setOutgoingLength` may generate a `APDUException`. But this is a runtime exception that we ignore in our specification of `getChallenge`. Of course, one can choose to include it and have a more detailed specification.

The `respond` method is only called (within the `process` method from the previous section) when `mode` is `CHARGING` and the incoming APDU contains the `INS_RESPOND` instruction. The method checks whether the received response is appropriate to the challenge previously sent, and if so, it extracts the new value for `balance` from the APDU buffer through some bitwise shifting and masking. (The resulting balance is described in slightly more readable form in the `ensures` clause.) The method then resets the `counter` field to zero. Whether or not the appropriate response is given is expressed in the JML specification.

```
/*@ behavior
  @    requires
  @       mode == CHARGING
  @         && counter < 5
  @         && apdu != null
  @         && apdu.buffer != null
  @         && apdu.buffer.length >= CIPHERTEXT_SIZE
  @         && cipher != null
  @         && ciphertxt != null
  @         && digest != null;
  @    modifiable
  @       mode, counter, balance,
  @       ciphertxt[*], plaintxt[*], sha_nonce[*];
  @    ensures
  @       mode == ISSUED
  @         && balance == (short)(256 * (plaintxt[SHA_SIZE] & 0x0F)
  @                               + (plaintxt[SHA_SIZE+1] & 0xFF))
  @         && counter == 0
  @         && (\forall int i; 0 <= i && i < SHA_SIZE ==>
  @                             sha_nonce[i] == plaintxt[i]);
  @    signals (ISOException e)
  @       mode == ISSUED
  @         && balance == \old(balance)
  @         && counter == \old(counter)
  @         && (apdu.buffer[OFFSET_LC] != ciphertxt.length
  @            ||
  @            !(\forall int i; 0 <= i && i < SHA_SIZE ==>
  @                              sha_nonce[i] == plaintxt[i]));
  @
  @*/
 private void respond(APDU apdu) throws ISOException {
   mode = ISSUED;
```

```
    byte[] buffer = apdu.getBuffer();
    if (buffer[OFFSET_LC] != ciphertxt.length) {
       ISOException.throwIt(
          SW_SECURITY_STATUS_NOT_SATISFIED);
    };
    readBuffer(apdu,ciphertxt);
    cipher.doFinal(ciphertxt,(short)0,
                   CIPHERTEXT_SIZE,plaintxt,(short)0);
    digest.doFinal(nonce,(short)0,
                   NONCE_SIZE,sha_nonce,(short)0);
    if (Util.arrayCompare(sha_nonce,(short)0,
                          plaintxt,(short)0,SHA_SIZE) == 0) {
        balance = (short)(((plaintxt[SHA_SIZE] & 0x0F) << 8)
                          | (plaintxt[SHA_SIZE+1] & 0xFF));
        counter = 0;
    } else {
       ISOException.throwIt(
          SW_SECURITY_STATUS_NOT_SATISFIED);
    }
  }
```

The issues in the verification of the `respond` method are similar to the ones for `getChallenge`. The proof cannot be done automatically with weakest-precondition calculations because there are too many complicated method calls involved. Therefore we have used a proof in Hoare logic, with much user interaction. The resulting proof tree in PVS is included in Figure 2, and gives an impression of the complexity and number of interactions (each node corresponds to a user-command). The reader is not expected to analyze the information in each node. The bitwise operations involved present no problems in our recent refinement of the semantics of Java's integral types—which will be described in detail elsewhere.

4 Conclusions

As far as we know, this paper is the first to describe a formal verification of a Java smart card applet which uses cryptographic methods (in the form of a simple challenge-response mechanism). Other approaches, such as in [21], concentrate on the development process, and stop short of the actual verification. The closest work to this paper we could find is [12]. The applet they develop seems to have been verified in a dynamic logic which abstracts away from certain Java Card details, though.

In essence, what we have verified is a correspondence between the logical control flow and the flow of the actual Java implementation. Control flow

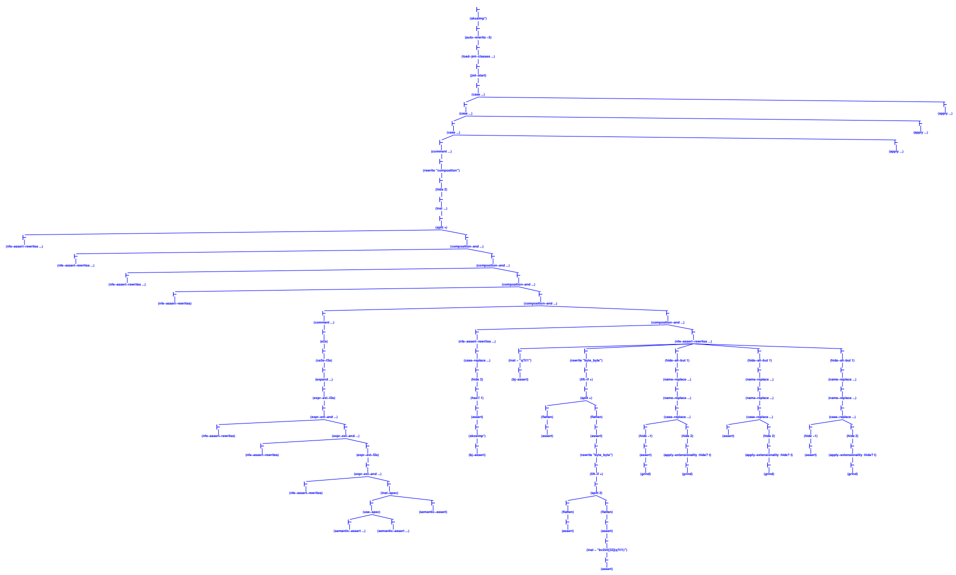

Fig. 2. Proof tree for the `respond` method.

analysis usually involves a lot of data abstraction. In contrast, our approach uses no data abstraction at all. This makes sure that methods are called in an appropriate order, obeying security sensitive restrictions—such as the upper limit on the number of outstanding challenge-responses. Furthermore it is possible to find actual implementation errors which can prove to be security critical. Low level implementation issues, such as overflow of data types and unwanted exceptions, simultaneously with control flow properties can thus be detected.

This work forms part of the European Research project VerifiCard [26], on tool-assisted verification of Java smart cards. It shows that actual verification of smart card applets is becoming feasible (see also [4]). However, it is still a non-trivial effort, not only the verification itself, but also the development of appropriate formal specifications. Yet, this investment forces the developers to think very carefully about their code. We think that our formalized approach with explicit states and state-based switching within the `process` method may serve as a blueprint for other applets.

Of course, this is not the final story. There are several aspects left for future research—such as transactions, authentication, links with abstract protocol descriptions, verification with a more complete API specification,more complex Java Card applications—which provide ample material for future work.

Acknowledgments

We thank Joachim van den Berg for his help with the proofs, especially in the construction of the dedicated PVS strategies.

References

[1] R. Anderson. The formal verification of a payment system. In M. G. Hinchey and J. P. Bowen, editors, *Industrial Strength Formal Methods*, pages 43–52. Springer-Verlag, September 1999.

[2] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer-Verlag, 2001.

[3] J. van den Berg, B. Jacobs, and E. Poll. Formal specification and verification of Java Card's application identifier class. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security (JavaCard Workshop 2000)*, volume 2041 of *LNCS*, pages 137–150. Springer-Verlag, 2001.

[4] C.-B. Breunesse, J. van den Berg, and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In *Algebraic Methodology and Software Technology*, LNCS. Springer, Berlin, 2002, to appear.

[5] R. Brinkman and J.-H. Hoepman. Secure method invocation in Jason. In *Proceedings of Cardis 2002*, pages 29–40. USENIX Assoc., 2002.

[6] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proc. Royal Soc.*, Series A, Volume 426:233–271, 1989.

[7] Z. Chen. *Java Card technology for smart cards: architecture and programmer's guide*. Addison-Wesley, June 2000.

[8] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEETSE*, 27(2):99–123, 2001.

[9] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, 2002.

[10] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement, FDR2 user manual*, May 2000.

[11] Global Platform. Open platform card specification version 2.1, June 2001. Available at `http://www.globalplatform.org/`.

[12] Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. A method for secure smartcard applications. In *Proceedings of AMAST 2002*, volume 2422 of *LNCS*, pages 319–333. Springer-Verlag, 2002.

[13] E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct java card applets. In *Proceedings of the 18th IFIP Information Security Conference*. Kluwer Academic Publishers, 2003. To appear.

[14] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *LNCS*, pages 284–303. Springer-Verlag, 2000.

[15] B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming.* (in this issue).

[16] B. Jacobs et al. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.

[17] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and William Harvey, editors, *Behavioral Specifications for Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[18] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06q, Dep. of Comp. Sci., Iowa State Univ. `http://www.cs.iastate.edu/~leavens/JML.html`, 2002.

[19] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6:53–84, 1998.

[20] R. Marlet and D. Le Métayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic, August 2001.

[21] W. Mostowski. Rigorous development of JavaCard applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002.

[22] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, USA, September 1999. Available at `http://pvs.csl.sri.com/`.

[23] E. Poll, J. van den Berg, and B. Jacobs. Specification of the Java Card API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer, 2000.

[24] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[25] The Krakatoa team. The Krakatoa proof tool, 2002. Available from http://www.lri.fr/~marche/krakatoa/.

[26] European IST-2000-26328 Project "VerifiCard". http://www.verificard.org.