Verification of Safety Properties in the Presence of Transactions

Reiner Hähnle and Wojciech Mostowski

Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
`{reiner,woj}@cs.chalmers.se`

Abstract. The Java Card transaction mechanism can ensure that a sequence of statements either is executed to completion or is not executed at all. Transactions make verification of Java Card programs considerably more difficult, because they cannot be formalised in a logic based on pre- and postconditions. The KeY system includes an interactive theorem prover for Java Card source code that models the full Java Card standard including transactions. Based on a case study of realistic size we show the practical difficulties encountered during verification of safety properties. We provide an assessment of current Java Card source code verification, and we make concrete suggestions towards overcoming the difficulties by *design for verification*. The main conclusion is that largely automatic verification of realistic Java Card software is possible provided that it is designed with verification in mind from the start.

1 Introduction

As Java Card technology is picking up speed it becomes more and more interesting to employ formal analysis techniques in order to ensure that Java Card applications work as intended. Formal approaches to Java Card application development encompass a wide spectrum from byte code to source code, from fully automated to highly interactive, and from abstract to fully concrete semantics (see Section 5 for a brief overview).

Our work is aimed at Java Card source code verification with full modelling of all semantic aspects. This includes the Java Card transaction mechanism that ensures a sequence of statements either being executed to completion or not being executed at all. The underlying technology, described in Section 2.2, is theorem proving in an expressive logic, in which programs and their requirements are formalised. Fully automatic inference in this context is in general unachievable, but one goal of the presented work was to find out just how far automation reaches.

The experiments described in this paper were made with the KeY theorem prover, which is an interactive verification system for Java Card featuring a complete formalisation of atomic transactions [4]. It is part of the KeY system [1], an integrated tool for informal and formal development of object-oriented software described in Section 2.1. This paper makes the following contributions:

- An experience report about the verification of parts of a Java Card electronic purse application (*Demoney*) of realistic complexity [23]. The code includes atomic transactions. To our best knowledge, this is the first report on verification of Java Card source programs with transactions without any simplification or abstraction. The case study and the experiments are described in Section 3.
- An assessment of current source code verification technology: what can be automatically proven in terms of LoC, complexity, etc.? Which desirable requirements can be expressed and which not? This is discussed in Section 4.1.
- An analysis of the limitations of current technology and how they can be overcome. We explain why the *Demoney* case study had to be partially refactored to make verification feasible. In particular, we make concrete suggestions towards overcoming the difficulties by *design for verification* in Section 4.2.

The main conclusion we draw in this paper is that largely automatic verification of realistic Java Card software is in the realm of the possible, but it is essential to move from *post hoc* verification to a more aggressive approach, where software is designed with verification in mind from the start.

2 Background

2.1 The KeY Project

The work presented in this paper is part of the KeY project[1] [1]. The main goals of KeY are to (1) provide deductive verification for a real world programming language and to (2) integrate formal methods into industrial software development processes. For the first goal a deductive verification tool, the KeY Prover, has been developed. The verification is based on a specifically tailored version of Dynamic Logic—Java Card Dynamic Logic (Java Card DL), which supports most of sequential Java including the full Java Card language specification. For the second goal we enhance a commercial CASE tool with functionality for formal specification and deductive verification. The design and specification languages of our choice are respectively UML (Unified Modelling Language) and OCL (Object Constraint Language), which is part of the UML standard. The KeY system translates OCL specifications into Java Card DL formulae, whose validity can then be proved with the KeY Prover. All this is tightly integrated into a CASE tool, which makes formal verification as transparent as possible to the untrained user.

Of course, the use of OCL is not mandatory: logically savvy users of the KeY system can write their proof obligations directly in Java Card DL and use its full expressive power. As we see later, this is even relatively straightforward.

[1] `http://www.key-project.org`

2.2 Java Card Dynamic Logic

We give a very brief introduction to Java Card DL. We are not going to present or explain any of its sequent calculus rules. Dynamic Logic [28, 17] can be seen as an extension of Hoare logic. It is a first-order modal logic with parametric modalities $[p]$ and $\langle p \rangle$ for every program p (we allow p to be any sequence of legal Java Card statements). In the Kripke semantics of Dynamic Logic the worlds are identified with execution states of programs. A state s' is *accessible* from state s *via* p, if p terminates with final state s' when started in state s.

The formula $[p]\phi$ expresses that ϕ holds in *all* final states of p, and $\langle p \rangle \phi$ expresses that ϕ holds in *some* final state of p. In versions of DL with a non-deterministic programming language there can be several final states, but Java Card programs are deterministic, so there is exactly one final state (when p terminates) or no final state (when p does not terminate). The formula $\phi \rightarrow \langle p \rangle \psi$ is valid if, for every state s satisfying precondition ϕ, a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds. The formula $\phi \rightarrow [p]\psi$ expresses the same, except that termination of p is not required, that is ψ needs only to hold *if* p terminates.

Java Card DL is axiomatised in a sequent calculus to be used in deductive verification of Java Card programs. The detailed description of the calculus can be found in [2]. The calculus covers all features of Java Card, such as exceptions, complex method calls, atomic transactions (see below), Java arithmetic. The full Java Card DL sequent calculus is implemented in the KeY Prover. The prover itself is implemented in Java. The calculus is implemented by means of so-called taclets [3], that avoid rules being hard coded into the prover. Instead, rules can be dynamically added to the prover. As a consequence, one can, for example, use different versions of arithmetic during a proof: idealised arithmetic, where all integer types are infinite and do not overflow, or Java arithmetic, where integer types are bounded and exhibit overflow behaviour [6].

To sum up the description of Java Card DL and to give the reader an impression of concrete Java Card DL formulae, we present a simple Java Card DL proof obligation:

$$\texttt{card.balance} \doteq b \ \vdash \ \langle \texttt{card.charge(amount);} \rangle \ \texttt{card.balance} \doteq b + \texttt{amount}$$

It says that if the `card` object's `balance` attribute is equal to b in the initial state, then the execution of method `charge` with argument `amount` terminates normally (no exception thrown) and afterwards the `card` object's initial `balance` is increased by `amount`. The validity of this proof obligation under Java integer semantics depends on whether `charge()` accounts for overflow, the type of the $+$ operator, etc.

2.3 Strong Invariants

While working on one of the Java Card case studies [27] it became apparent that the specification semantics based on the initial and final states of a program is not enough to specify and verify some Java Card safety properties. It turned out that the Java Card applet in question was not "rip-out safe": it is possible to destroy the applet's functionality by removing (ripping out) the Java Card device from the card reader (terminal) while the applet on the card executes. As a result of this the applet's memory may become corrupted and left in an undefined state, causing malfunctioning of the applet.

To avoid such errors one has to be able to specify and verify the property that a certain invariant on the objects' data is maintained at any time during applet execution and, in particular, in case of abrupt termination. Usually, class invariants (in OCL and elsewhere) are interpreted with respect to pre/post state semantics, that is, if the invariant holds before a method is executed then it holds again after the execution of a method. This semantics does not suffice to ensure properties of data in intermediate states during method's execution. To solve this problem, we introduced *strong* invariants, which allow to specify properties about all intermediate states of a program.[2]

For example, the following strong invariant (expressed in pseudo OCL) says that we do not allow partially initialised `PersonalData` objects at any point in our program. In case the program is abruptly terminated we should end up with either a fully initialised object or an uninitialised (empty) one:

```
context PersonalData throughout:
  not self.empty implies
    self.firstName <> null and self.lastName <> null and self.age > 0
```

To introduce the notion of a strong invariant it was necessary to extend the Java Card DL with a new modal operator $[\![\cdot]\!]$ ("throughout"), which closely corresponds to Temporal Logic's $\Box$ operator. In the extended logic, the semantics of a program is a sequence of all states the execution passes through when started in the current state (its *trace*). Using $[\![\cdot]\!]$, it is possible to specify properties of intermediate states in traces of terminating and non-terminating programs. And such properties (typically strong invariants and safety constraints) can be verified using the Java Card DL calculus extended with additional sequent rules for the "throughout" modality [4].

2.4 Java Card Atomic Transactions

There is one particular aspect of Java Card that makes the "throughout" extension considerably more complicated than expected, namely, the Java Card transaction mechanism. The transaction mechanism allows a programmer to enforce atomicity of sequences of Java Card statements. It is typically used to ensure consistency of related data that have to be updated simultaneously.

The memory model of Java Card differs somewhat from Java's memory model [12, 31]. In smart cards there are two kinds of writable memory: persistent

[2] In extended static checking a closely related concept called *object invariants* is used [21]. The semantics of OCL invariants is interpreted in the strong sense in [34], where a temporal extension of OCL is introduced.

memory (EEPROM), which is preserved between card sessions, and transient memory (RAM), whose contents disappears when power loss occurs, for example, when the card is removed from the reader. Hence, every memory location in Java Card (variable or object field) is either persistent or transient. The Java Card language specification gives the following rules (slightly simplified for this presentation): all objects (including the reference to the currently running applet, `this`, and arrays) are created in persistent memory. Therefore, in Java Card assignments such as "`o.attr = 2;`", "`this.a = 3;`", and "`arr[i] = 4;`" all have a permanent character; that is, the assigned values will be kept after the card loses power. A programmer can create an array with transient elements, but currently there is no possibility to make objects (fields) other than array elements transient. All local variables are transient.

The distinction between persistent and transient objects is very important since these two types of objects are treated in a different way by Java Card's transaction mechanism. The following are the Java Card system calls for transactions with their description:

`JCSystem.beginTransaction()` begins an atomic transaction. From this point onwards, until the transaction finishes, all assignments to fields of objects are executed conditionally, while assignments to transient variables or array elements are executed unconditionally (immediately).

`JCSystem.commitTransaction()` commits the transaction. All conditional assignments are committed (in one atomic step).

`JCSystem.abortTransaction()` aborts the transaction. All the conditional assignments are rolled back to the state in which the transaction started. Assignments to transient variables and array elements remain unchanged (as if there were no transaction in progress).

A "throughout" property (formula) has to be checked after every single field or variable assignment which, according to the Java Card runtime environment specification [31], is atomic. Such checks have to be suspended, however, when a transaction is in progress, because the assignments inside a transaction are not atomic, only the whole transaction is atomic. Moreover, as already said, each transaction can either finish successfully, in which case it commits all the conditional assignments, or it can fail and in that case the transaction is aborted and all the conditional assignments have to be rolled back. The logic has to account for the possibility of an abort and for the difference between persistent and transient data.

Observe that the possibility of an aborted transaction affects even the semantics of the standard modal operators $\langle\cdot\rangle$ and $[\cdot]$, because an abort affects the final state of the program. Details of how the extension of Java Card DL that deals with transactions is handled in the calculus can be found in [4]. We do not repeat the technical solution in this paper, but we stress that the details are rather involved and surprisingly complex. The KeY Prover implements the whole extension of Java Card DL with "throughout" and transaction mechanism. To our knowledge the KeY Prover is the only prover for Java Card programs that fully handles Java Card transactions.

When a strong invariant has been specified for a Java Card program, say, for a class C, each of C's methods can be a subject to verification with respect to the strong invariant. A typical proof obligation for a method `m()` involving a strong invariant looks as follows:

$$(\mathit{Inv} \wedge \mathit{Pre} \wedge \mathit{StrongInv}) \rightarrow [\![C :: \texttt{m();}]\!]\ \mathit{StrongInv}$$

Inv stands for a standard (weak) invariant of class C and *Pre* stands for the method's precondition. Apart from those two premises one also has to assume that the strong invariant *StrongInv* holds before method `m()` is executed to establish that *StrongInv* holds throughout the execution of `m()`.

3 Case Study: Java Card Electronic Purse

The case study presented here is based on the Java Card electronic purse application *Demoney* [23]. While *Demoney* has not all the features of a purse application actually used in production, it is provided by *Trusted Logic S.A.* as a realistic demonstration application that includes all major complexities of a commercial program.

Our target program is a somewhat refactored fragment of *Demoney* and concentrates on the important aspects of the application to highlight our verification results. The *Demoney* source code is at present not publicly available, and we do not show it. The program we verified is, however, very close to *Demoney* and follows the *Demoney* specification [23]. We deviate from *Demoney* mainly in that our program is designed to make verification simpler. We discuss these issues in detail in Section 4.2.

The safety properties that we discuss here were directly motivated by the ones described in [24]. In fact the property we prove (that the current balance of the purse is always in sync with the balance recorded in the most recent log entry) for the `processSale` method presented in Section 3.4 is exactly the one described in [24, Section 3.5]. The example mentioned there is also based on the *Demoney* application.

3.1 The LogRecord Class

The UML class diagram of our program is shown in Figure 1. The basic class is `LogRecord` which is used to store data about a single purse transaction. The data consists of the new balance after the transaction (`balance:short`), transaction identifier (`transactionId:int`) and transaction date (`date:SaleDate`). Additionally, the attribute `empty` states if a particular instance of `LogRecord` is in use.

Such an attribute is characteristic for the Java Card platform, which is a memory constrained device and in general does not possess a garbage collector. To avoid memory overflow during execution all objects are allocated during the initialisation phase of Java Card applets and the programmer keeps track of

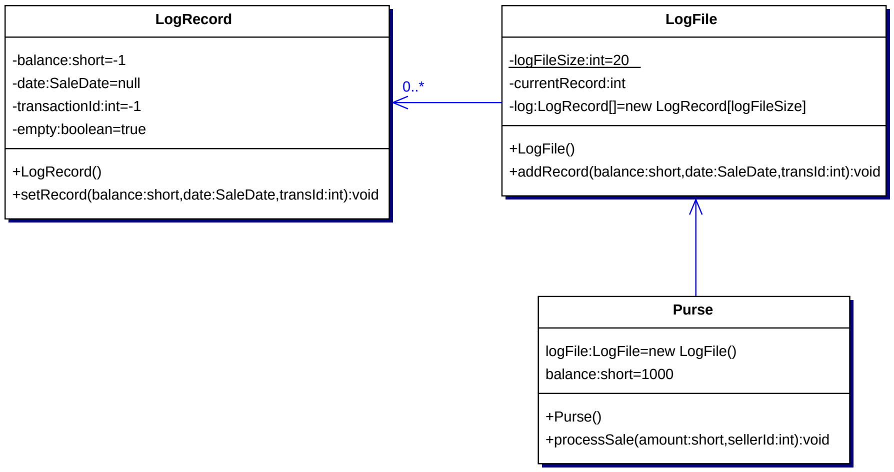

LogRecord
-balance:short=-1
-date:SaleDate=null
-transactionId:int=-1
-empty:boolean=true
+LogRecord()
+setRecord(balance:short,date:SaleDate,transId:int):void
0..*
LogFile
-logFileSize:int=20
-currentRecord:int
-log:LogRecord[]=new LogRecord[logFileSize]
+LogFile()
+addRecord(balance:short,date:SaleDate,transId:int):void
Purse
logFile:LogFile=new LogFile()
balance:short=1000
+Purse()
+processSale(amount:short,sellerId:int):void

Fig. 1. `Purse` application class diagram

which objects are already in use, for example by introducing attributes like `empty`.[3] The `LogRecord` class contains only one method, which is responsible for assigning values to its attributes:

```
public void setRecord(short balance, SaleDate date, int transId) {
  this.balance = balance;
  this.date = date;
  this.transactionId = transId;
  this.empty = false;
}
```

3.2 Specification and Verification of setRecord

Regarding data consistency, the main property one needs to establish about the class `LogRecord` is to assure that at any point all the instances of this class that are in use are properly initialised. Expressed in (pseudo) OCL this property reads:

```
context LogRecord throughout:
  not self.empty implies
    self.balance >= 0 and self.transactionId > 0 and self.date <> null
```

This states that all attributes of `LogRecord` objects that are in use have proper values at any point in time. We want to prove that the method `setRecord`

[3] Some design and implementation choices in our example may seem artificial (for example, the value of `empty` never changes from `false` to `true`), but the point was to illustrate certain critical issues.

preserves this strong invariant. In order to do this, one needs a precondition saying that the parameters that are passed to `setRecord` have proper values. The resulting Java Card DL proof obligation in the actual notation used by the KeY Prover is:

```
  !self = null
& balance >= 0 & !date = null & transId > 0
& (self.empty = FALSE ->
    (self.balance >= 0 & !self.date = null & self.transactionId > 0))
-> [[{ self.setRecord(balance, date, transId); }]]
  (self.empty = FALSE ->
    (self.balance >= 0 & !self.date = null & self.transactionId > 0))
```

This is proved automatically with 230 rule applications in 2 seconds.[4] If we change the strong invariant into a weak invariant, that is, replace the throughout modality in the formula above with a diamond modality, the resulting proof obligation is (as expected) also provable (125 rules, less than 2 seconds).

Observe that the order of attribute assignments in `setRecord`'s body is crucial for the strong invariant to hold. If we change `setRecord`'s implementation to

```
public void setRecord(short balance, SaleDate date, int transId) {
  this.empty = false;
  this.balance = balance;
  this.date = date;
  this.transactionId = transId;
}
```

then it does not preserve the strong invariant anymore, while it still preserves the weak invariant. When trying to prove the strong invariant for this implementation the prover stops after 248 rule applications with 6 open proof goals. The proof for the weak invariant proceeds in the same fashion as for the previous implementation.

3.3 The Purse Class

The `Purse` class is the top level class in our design. The `Purse` stores a cyclic file of log records (each new entry allocates an unused entry object or overwrites the oldest one), which is represented in a class `LogFile`. `LogFile` allocates an array of `LogRecord` objects, keeps track of the most recent entry to the log and provides a method to add new records—`addRecord`.

The `Purse` class provides only one method—`processSale`. It is responsible for processing a single sale performed with the purse—debiting the purchase amount from the balance of the purse and recording the sale in the log file. To ensure consistency of all modified data, Java Card transaction statements

[4] All the benchmarks presented in this paper were run on a Pentium IV 2.6GHz Linux system with 512MB of memory. The version of the KeY Prover used (0.1200) is available on request. The prover was run with Java 1.4.2.

are used in `processSale`'s body. Figure 2 shows the UML sequence diagram of `processSale`. The total amount of code invoked by `processSale` amounts to less than 30 lines, however, it consists of nested method calls to 5 different classes.

3.4 Specification and Verification of processSale

As stipulated in [24], we need to ensure consistency of related data. In our case, this means to express that the state of the log file is always consistent with the current state of the purse. More precisely, we state that the current balance of the purse is always equal to the balance stored in the most recent entry in the log file. The corresponding strong invariant expressed in pseudo OCL is:

context Purse **throughout**:
  self.logFile.log.get(self.logFile.currentRecord).balance = **self**.balance

Since `processSale` is the method that modifies both the log file and the state of the purse, we have to show that it preserves this strong invariant. The most important part of the resulting proof obligation expressed in Java Card DL is the following:

```
  JCSystem.transactionDepth = 0
& !self = null
& !self.logFile = null
& !self.logFile.log = null
& self.logFile.currentRecord >= 0
& self.logFile.currentRecord < self.logFile.log.length
& self.logFile.log[self.logFile.currentRecord].balance = self.balance
-> [[{ self.processSale(amount, sellerId); }]]
  self.logFile.log[self.logFile.currentRecord].balance = self.balance
```

This proof obligation is proved automatically by the KeY Prover modelling the full Java Card standard (see Section 3.6) in less than 2 minutes (7264 proof steps).

3.5 *Post Hoc* Verification of Unaltered Code

We just reported on successful verification attempts of a refactored and partial version of the *Demoney* purse application. When it comes to capabilities and theoretical features of the KeY Prover there is nothing that prevents us in principle from proving properties about the real *Demoney* application. There are, however, some design features in *Demoney* that make the verification task difficult. We discuss them in detail in Section 4.2.

We also proved total correctness proof obligations for two simple, but *completely unaltered*, methods of *Demoney* called `keyNum2tag` and `keyNum2keySet`. This was possible, because the problems discussed in Section 4.2 below stayed manageable in these relatively small examples. It was crucial that the KeY Prover

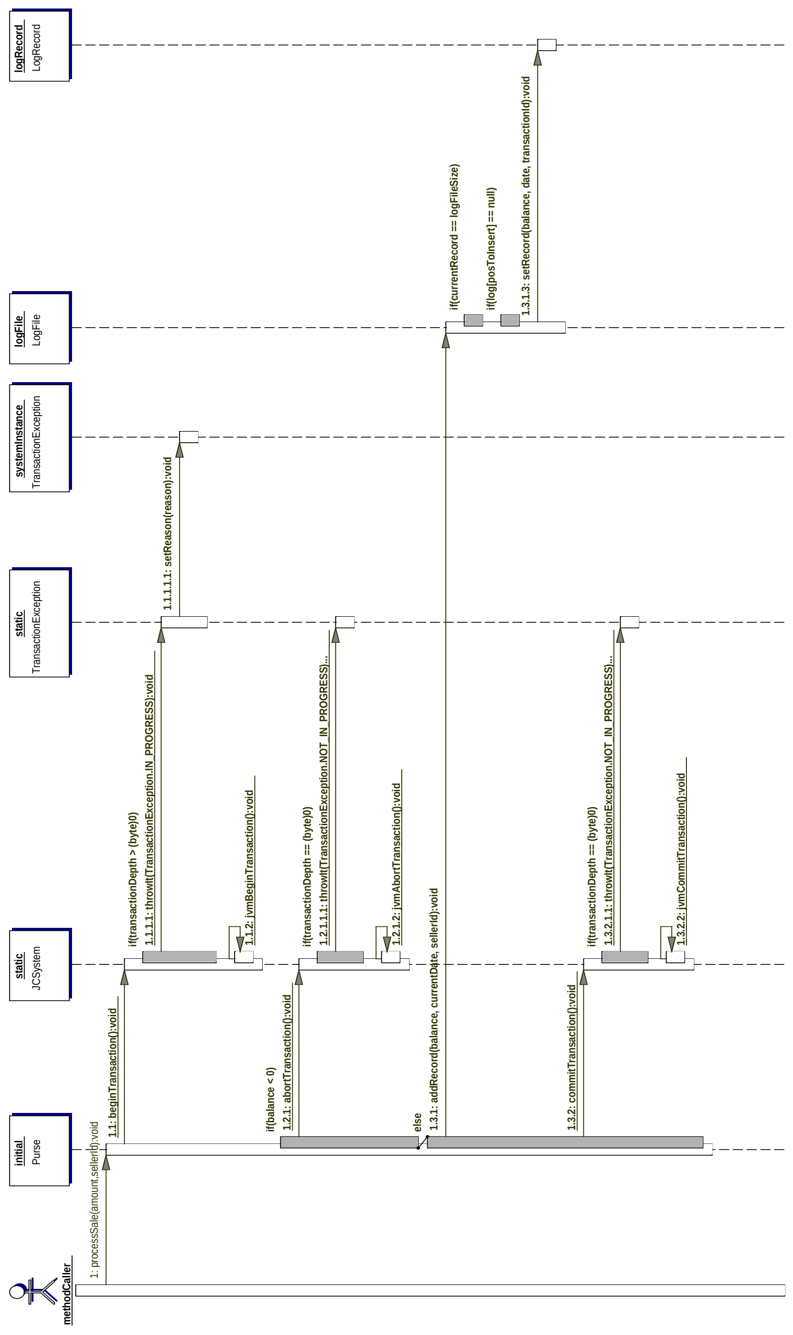

methodCaller
initial
Purse
static
JCSystem
static
TransactionException
systemInstance
TransactionException
logFile
LogFile
logRecord
LogRecord
1: processSale(amount,sellerId):void
1.1: beginTransaction():void
if(transactionDepth > (byte)0)
1.1.1.1: throwIt(TransactionException.IN_PROGRESS):void
1.1.1.1.1: setReason(reason):void
1.1.2: jvmBeginTransaction():void
if(balance < 0)
1.2.1: abortTransaction():void
if(transactionDepth == (byte)0)
1.2.1.1.1: throwIt(TransactionException.NOT_IN_PROGRESS)...
1.2.1.2: jvmAbortTransaction():void
else
1.3.1: addRecord(balance, currentDate, sellerId):void
if(currentRecord == logFileSize)
if(log[posToInsert] == null)
1.3.1.3: setRecord(balance, date, transactionId):void
1.3.2: commitTransaction():void
if(transactionDepth == (byte)0)
1.3.2.1.1: throwIt(TransactionException.NOT_IN_PROGRESS)...
1.3.2.2: jvmCommitTransaction():void

Fig. 2. Sequence diagram of the `processSale` method

allows to prove properties of *unaltered* JAVA code. This implies that, in principle, JAVA code does not have to be prepared, translated, or simplified in any way before it can be processed by the prover. Unaltered JAVA source programs are first-class citizens in Dynamic Logic. JAVA CARD DL formulae simply contain references to source code locations such as this:

```
fr.trustedlogic.demo.demoney.Demoney self;
byte keyNum;
byte result;
...
result = self.keyNum2tag(keyNum);
```

As the source code we proved properties about was given beforehand, what we did can be called *post hoc* verification.

3.6 Performance

We emphasise that all mentioned proofs were achieved fully automatically. What it means for the user is that there is no interaction required during the proof and, as a consequence, the user does not have to understand the workings of the JAVA CARD DL calculus.

Proof Obligation	Time (sec.)	Steps	Branches
$[\![$`setRecord`$]\!]$	2.0	230	20
$\langle$`setRecord`$\rangle$	1.5	125	6
$[\![$`setRecord`$]\!]^F$	2.1	248	6 open
$\langle$`keyNum2tag`$\rangle^D$	3.3	392	18
$\langle$`keyNum2keySet`$\rangle^D$	5.5	640	33
$[\![$`processSale`$]\!]^1$	41.4	3453	79
$[\![$`processSale`$]\!]^2$	51.3	4763	248
$[\![$`processSale`$]\!]^3$	111.1	7264	338

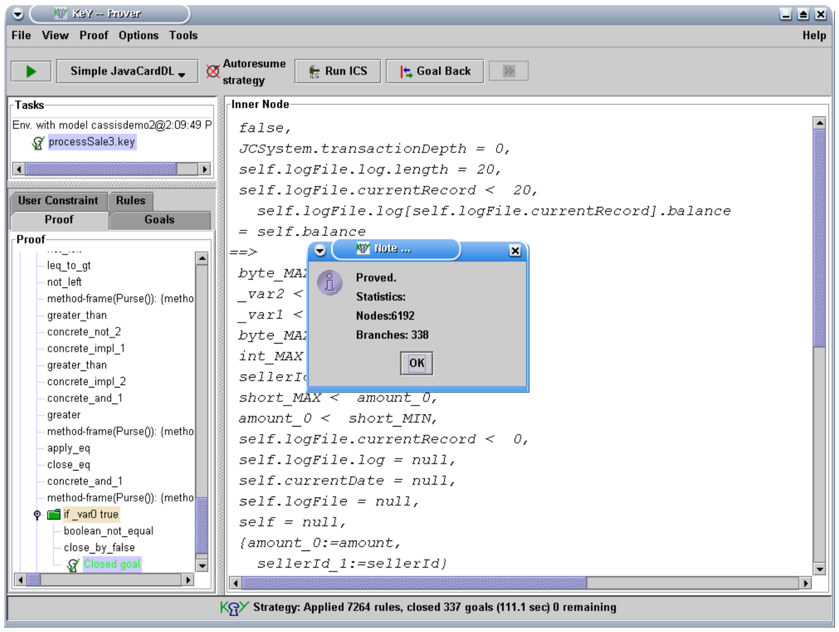
^{F} Failed proof attempt
^{D} Methods from *Demoney* (full pre/post behavioural specification)
^{1} Ideal arithmetic, no null pointer checks
^{2} Ideal arithmetic, with null pointer checks
^{3} JAVA arithmetic, with null pointer checks

Table 1. Performance of KeY Prover for examples discussed in the text

Table 1 summarises proof statistics relating to the examples discussed previously. Some explanations about the three different versions of the proof for `processSale` are due: the KeY Prover allows to use different settings for the rules used during a proof. One of those settings concerns the kind of arithmetics (see Section 2.2). When ideal arithmetic is used, then all integer types are considered to be infinite and, therefore, without overflow. When Java arithmetic is used, the peculiarities of integer types as implemented in Java are taken into account: different range (`byte`, `short`, etc.), finiteness, and cyclic overflow.

Another prover setting is the `null` value check. When switched off, many variables with object references are assumed to be non `null` without bothering to prove this fact. When switched on, the prover establishes the proper value of every object reference. Obviously, proofs involving `null` checks are more expensive. The checks for index out of bounds in arrays are *always* performed by the prover. The benchmark for the third version of `processSale` represents the prover's behaviour with support for the full Java Card standard.

Figure 3 shows a screenshot of the KeY Prover with a successful proof for the third version of `processSale`.



```
Env. with model cassisdemo2@2:09:49 P
  processSale3.key
```

```
leq_to_gt
not_left
method-frame(Purse()): {metho
greater_than
concrete_not_2
concrete_impl_1
greater_than
concrete_impl_2
concrete_and_1
greater
method-frame(Purse()): {metho
apply_eq
close_eq
concrete_and_1
method-frame(Purse()): {metho
if _var0 true
  boolean_not_equal
  close_by_false
  Closed goal
```

```
false,
JCSystem.transactionDepth = 0,
self.logFile.log.length = 20,
self.logFile.currentRecord <  20,
  self.logFile.log[self.logFile.currentRecord].balance
= self.balance
==>
byte_MA
_var2 <
_var1 <
byte_MA
int_MAX
sellerI
short_MAX <  amount_0,
amount_0 <  short_MIN,
self.logFile.currentRecord <  0,
self.logFile.log = null,
self.currentDate = null,
self.logFile = null,
self = null,
{amount_0:=amount,
  sellerId_1:=sellerId}
```

```
Proved.
Statistics:
Nodes:6192
Branches: 338
```

```
Strategy: Applied 7264 rules, closed 337 goals (111.1 sec) 0 remaining
```

Fig. 3. KeY Prover window with successful proof

4 Results

4.1 Verification Technology

Although we so far managed to verify only a small and partly refactored part of *Demoney*, we are encouraged by what we could achieve. The verified programs contain many complex features: nearly every statement can throw an exception, many Java arithmetic and array types occur, there are several nested method calls and, above all, Java Card transactions that may cause subtle errors.

The largest example involves about 30 lines of source code. This may not seem much, but it clearly indicates that methods and classes of non-trivial size can be handled. In addition, the next version of the KeY prover will support composition of proofs including a treatment of representation exposure by computation of modifier sets [7]. Consequently, we expect that formal verification of Java Card programs comparable to *Demoney* is achievable before long.

On the other hand, there are also serious limitations. To start with, we observed that verification of the more complex methods of the unaltered *Demoney* program results in specifications and proof obligations that simply become too long and complex. In our opinion, this problem must be attacked by moving from *post hoc* verification to *design for verification*, see the following section.

It would be desirable to have a more formal statement here relating types of programs and proof complexity. The problem is that even loop-free Java Card programs contain control structures like exceptions and transactions that have a global effect on control flow. Taking away all critical features yields an uninteresting programming language, while leaving them in renders general statements on proof complexity (at least the ones we could think of) simply untrue.

A principal obstacle against automating program verification is the necessity to perform induction in order to handle loops (and recursion). In most cases, the induction hypothesis needs to be generalised, which requires considerable user skill. There is extensive work on automating induction proofs, however, mostly for simple functional programming languages as the target. Only recently, preliminary work for imperative target languages [29, 16] appeared. If, however, *Demoney* is a typical Java Card application, then loops might be much less of a problem than thought: of 10 loops in *Demoney* (9 `for`, 1 `while`) most are used to initialise or traverse arrays of known bounds. Such loops do not require induction at all. The next version of the KeY Prover contains a special automated rule for handling them. Our analysis showed that at most one loop in *Demoney perhaps* needs induction. There is no recursion.

Speed and automated theorem proving support, for example, for arithmetic properties, need to be improved in order to achieve an interactive working mode with the prover, which is not possible with proofs that in some cases take minutes. There is no principal obstacle here; for example, the speed increased by an order of magnitude since we began the case study.

An important question is whether we are able to express all relevant requirements. There is no agreement on standard requirements for Java Card, but the report [24] can serve as a guideline. Many of the security properties related there can be expressed in Java Card DL including strong invariants. In the present paper we concentrated on data consistency in connection with atomic transactions. The examples included also overflow control. In [14] it was shown that also information flow properties are expressible. We have strong evidence that also memory allocation control, error control and even the well-formedness of transactions can be formulated. For example, the following two properties, taken from [24] can be formulated in Java Card DL: (i) no `TransactionException` related to well-formedness is thrown, (ii) only `ISOExceptions` are thrown at the top level of an applet.

The main limitation of the currently used version of Java Card DL is the impossibility to express complex temporal relationships between the execution of different code fragments to establish advanced control flow properties such as a certain temporal order on method calls. This requires more complex temporal operators than "throughout" or some kind of event mechanism, and is a topic for future research. On the specification side, some work was done in [32], while [8] looked at abstracted byte code in a model checking framework.

4.2 Design for Specification and Verification

The way *Demoney* is designed and coded causes certain technical complications both when specifying and proving safety properties of programs with transactions. We demonstrate two issues and discuss their impact on the process of specification and verification of Java Card programs. Thereby, we give guidelines for the design of Java Card applications to avoid such problems.

Byte Arrays. Following the specification in [23, p. 17] *Demoney* implements a cyclic log file in a very similar fashion to our `Purse` class. *Demoney* stores more information than our program in a single log record, but that's not an issue when it comes to formal verification. The major difference is that each single log record is implemented as a `byte` array instead of an object (of class `LogRecord` in our case). We suspect that the main reason for implementing a log record as a `byte` array is to ease the transportation of log data to the card terminal. Another reason, explicitly mentioned in the specification, is to follow the schema of recording data in the form of TLVs (Tag-Length-Value). Finally, because of memory costs in smart cards, `byte` arrays are still much used to save some small memory overhead one has to pay for object instances and booleans.[5]

The use of a `byte` array instead of an object type has consequences for the verification process. To start with, Java Card allows only one dimensional arrays, which means that one cannot explicitly declare in a Java Card program that a log file is a two-dimensional array. So instead of saying

```
byte[][] logFile;
```

one has to say

```
Object[] logFile;
```

and then allocate this data structure by saying:

[5] The last point was confirmed by Renaud Marlet, Trusted Logic S.A., in personal communication.

```
logFile = new Object[logFileSize];
for(short i=0; i<logFile.length; i++)
  logFile[i] = new byte[LOG_RECORD_SIZE];
```

Since this is a dynamic allocation, there is no static information on the type of elements in the `logFile` array. Statically, one can only deduce that those elements are of type `Object`. In the verification process however, such information has to be made more precise. Since it cannot be deduced statically, it has to be included in the assumptions (that is, preconditions) of a proof obligation explicitly. In Java Card DL this requires use of existential quantifiers and lengthy Dynamic Logic expressions. In many cases, existential quantification makes it harder to find a proof automatically. If, instead, one declares a `logFile` as

```
LogRecord[] logFile;
```

the situation is much clearer from the prover's point of view. The only assumption needed in this case is that the elements of the `logFile` array are not `null`. In general it would also require a quantifier (universal), but in the special case of our program we are only interested in two elements of this array, so that the following assumption is sufficient:

```
!logFile[currentRecord] = null &
  !logFile[(currentRecord + 1) % logFileSize] = null
```

This, together with the declaration of `logFile`, is enough for the prover to establish type information about all relevant elements of `logFile`. Moreover, if the `logFile` is statically allocated right after it is declared,

```
LogRecord[] logFile = new LogRecord[20];
```

then no assumptions about the elements of `logFile` are necessary at all. The `logFile` example is not an isolated case, as one can find several occurrences of declarations of `Object` arrays in *Demoney*.

The second issue with the use of `byte` arrays for storing log records is related to arithmetics. The strong invariant for our `Purse` class states:

```
self.logFile.log[self.logFile.currentRecord].balance = self.balance
```

The type of attribute `balance` both in `LogRecord` and in `Purse` is `short`. When the `byte` array is used for storing log record data, then the value of balance is stored in two `byte` elements of this array. Comparing such a two `byte` value stored in an array to a `short` value becomes a bit complicated:

```
self.logFile.log[self.logFile.currentRecord][OFF_BALANCE] =
  castToByte((self.balance - castToByte(self.balance % 256)) / 256) &
self.logFile.log[self.logFile.currentRecord][OFF_BALANCE + 1] =
  castToByte(self.balance % 256)
```

This specification expression is based on an educated guess of how the Java Card API method `Util.setShort` [31] is implemented (`setShort` is a native method and its implementation is not disclosed). Expanding the Dynamic Logic function symbol castToByte results in another modulo operation. Also note that all arithmetic function symbols have Java types and must be checked against overflow. Proving with expressions such as the one shown above is difficult, if not practically unfeasible.

We sum up the problems associated to byte arrays: (1) typing information is difficult to establish, causing very complicated preconditions, and (2) comparison of short values unwrapped into two byte values requires the use of complex expressions involving modulo arithmetics. Both problems have serious impact on the size of proofs and automation.

The use of byte arrays is partially steered by the TLV standard. We do not argue with the purpose or usability of this standard in smart card technology, and we accept its motivations, such as the performance and space optimisation of Java Card applets. It seems obvious, however, that some things have to be traded off to ease formal specification and verification of Java Card programs.

One general guideline would be to use object types to store any kind of non-primitive data, at least if they are persistent (for transient data there is no choice but an array in Java Card). Furthermore, serialise objects only if necessary (in case of Java Card for communication). As part of a bigger picture one should consider to decouple application functionality from the communication model. Such a decoupled design is likely to allow decomposable, and thus easier, verification. It is more robust, too. We point to the fact that the examples presented in [24] follow for the most part the guideline of using object types instead of byte arrays for storing data.

Cyclic Indexing of Arrays. Another problematic issue for specification and verification is the way information on the most recent record in the log file is kept and updated in *Demoney*. This is rather a problem of coding conventions and not a design issue. *Demoney*'s cyclic file class has an attribute that stores the index of the next record to be used—nextRecordIndex. In order to access the most recent entry in the log, one writes an expression like:

```
logFile[(nextRecordIndex - 1) % logFileSize] ...
```

Modulo arithmetics is used to calculate the actual index. If we add the way the nextRecordIndex is updated, that is

```
nextRecordIndex = (nextRecordIndex + 1) % logFileSize;
```

then the prover has to establish the validity of equations such as:

```
index = (((index - 1) % logFileSize) + 1) % logFileSize
```

where all arithmetic function symbols have Java types and must be checked against overflow. This is certainly not impossible, but it adds substantially to the complexity of the resulting first-order proof obligations and, in connection with other phenomena, can make the problems too difficult to prove automatically.

To avoid these complications, we suggest two simple guidelines. The first is to keep track of those indices that are relevant for specification and verification, instead of those for implementation (or simply keep both kinds of indices). The second is to avoid modulo operations, if possible. The update of `nextRecordIndex` can be easily rewritten as:

```
nextRecordIndex++;
if (nextRecordIndex == logFileSize)
   nextRecordIndex = 0;
```

This program fragment might not be as simple and fast as the one before, but it considerably eases verification.

We believe that if the problems mentioned in this section were not present we would be able to verify automatically that *Demoney*'s `performTransaction` method preserves the kind of strong invariant that we had in our `Purse` class.

Discussion. Asking a programmer to rewrite the code to ease verification may seem unrealistic. It may look as if we put the burden of making verification feasible on the programmer instead of enabling the prover handle arbitrarily complex programs. This is not the case. Our aim is to make the KeY prover powerful enough to deal with complex Java Card code, however, one cannot expect a prover to deal with baroque programs optimised for performance. A trade-off has to be found. The guidelines we proposed are simple to follow and, in addition, make sense from a software engineering point of view. In particular, we do not assume that the programmer has any knowledge of the theorem prover.

Another counter argument against rewriting the code is that abstraction and interface specification should be used to simplify the verification process and get around some of the problems we described above. We fully agree with this, where this possibility is applicable, but in the context of Java Card applet verification it is not so. For example, when one proves a rip-out related property, one cannot abstract away from the implementation of the API methods, because the actual implementation of an API method affects the intermediate states of the program being verified.

5 Related Work

A version of Dynamic Logic that extends pure Dynamic Logic with trace modalities "throughout" and "at least once" was first presented in [5]. The axiomatisation of transactions was provided in [4]. Paper [18] proposes another approach to reasoning about rip-out properties (called card tears there). It presents a theoretical framework for dealing with card tears and transactions based on global program (method) transformation (as opposed to the KeY approach of local transformations). This paper does not report on any practical verification attempts. In [32] temporal constructs are introduced to the Java Modelling Language (JML), but they refer to sequences of method invocations and not to sequences of intermediate program states.

Paper [19] is closely related to our work in the sense that it reports on successful verification attempts of a commercial Java Card applet with different verification tools (ESC/Java2, Jive, Krakatoa, LOOP). The security property under consideration, also mentioned in Section 4.1, is that only `ISOExceptions` are thrown at the top level. Transaction related properties are not investigated. Like in the present study, it is stressed that two-dimensional `byte` arrays and the use of `byte` arrays in general are problematic in Java Card verification, and have serious impact on the size and complexity of proofs. One of the main results is that subtle bugs were found in the applet.

GemPlus provides a Java Card case study similar to *Demoney* [10], also a purse application and publicly available.[6] We do not use it at the moment, because it contains a large number of features that detract from the basic issues and make it less suitable as a starting point for Java Card verification. In addition, it was not developed further in the last three years.

Related work in Java Card verification can be classified according to several criteria. Working on byte code avoids the problems of source code availability and compiler trustworthiness, but makes full verification more difficult due to information loss during compilation. An overview of work done on the byte code level is provided in [9]—we concentrate on efforts targeted at source code: here, one can distinguish between methods that attempt complete modelling of the Java Card semantics and those that do not. The latter include model checking and extended static checking.

Model checking is based on a suitable abstraction of the execution model, which in the Bandera project [13] is Java, and of the requirements. The advantages are full automation of the model checking phase, trace generation for counter models, and treatment of concurrent Java programs. The drawback is the need for abstraction which poses difficulties for programs containing Java arithmetic and other inductive data structures. Bandera handles Java, not Java Card, and hence no transactions. In design-by-contract [25] and extended static checking (ESC) [15] Java source code is decorated with annotations from a restricted language. Annotated programs (via an intermediate representation) undergo a dynamic analysis that produces first-order verification conditions for a theorem prover. The analysis does not attempt to be complete, but it is fully automatic and produces warnings, when annotations are potentially violated. ESC is related to our strong invariants, because arbitrary code locations can be annotated with object invariants [21]. An approximation of strong invariants within ESC can be obtained by annotating every program point with the desired invariant.[7] Again, atomic transactions are not supported, as the target language is Java.

Closest to our approach are source code verifiers for Java based on various program calculi. The LOOP tool [20] translates Java source code with JML specifications into theories for the PVS theorem prover. Java semantics is described with co-algebras and uses higher-order logic as an internal representation.

[6] `http://www.gemplus.com/smart/enews/st1/pacap.html`

[7] We thank Rustan Leino for pointing this out.

Higher-order logic is also used to formalise syntax and semantics of a Java fragment in Isabelle [33] and in the Krakatoa tool [22]. In the latter Java programs and their JML specifications are translated into an intermediate, mostly functional, language, then proof obligations are generated, which in turn are proved with the Coq proof assistant. The Jive system [26] is based on an extended Hoare style calculus, Jack [11] on weakest precondition calculus, and KIV [30] on Dynamic Logic. The last three systems are closely related to the KeY Prover in that they all axiomatise Java with logical rules that can be seen as a small step operational semantics and proofs can be interpreted as symbolic execution with induction. The differences lie in the details and scope of the axiomatisation as well as support for automation. As far as we know, KeY is the only system that supports strong (object) invariants and, in particular, the semantics of Java Card transactions.

6 Conclusions

In this paper, we presented and analysed a case study concerned with formal specification and verification of Java Card programs. Our results show that largely automated formal verification of realistic Java Card applications without abstraction is possible in the near future. It is possible already now provided that applications are designed with verification in mind from the start. We gave a number of simple design guidelines that drastically simplify proofs while creating only a moderate performance overhead. We believe this to be acceptable, because even in the smart card world, performance restrictions become less of an issue. Besides, a small memory overhead seems an acceptable price for provably correct programs.

We concentrated in this case study on safety (data consistency) properties in the presence of transactions and possible arithmetic overflow. Information flow, memory allocation, well-formedness of transactions, and error analysis would be possible to formulate, but we cannot say anything about feasibility at this time. Temporal relationships between the execution of different code fragments as needed to enforce an order on method calls are a topic for future research.

Acknowledgements

We would like to thank Renaud Marlet of Trusted Logic S.A. for providing the *Demoney* case study. We also thank the organisers of CASSIS'04 for the opportunity to present this work. We thank the following people for reading drafts of this paper and providing valuable feedback: Renaud Marlet, Steffen Schlager, and Martin Giese. The anonymous reviewers helped with their constructive criticism and pointers to relevant literature to improve the paper.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software*

and Systems Modeling, April 2004. Springer-Verlag, Online First issue, to appear in print.

2. B. Beckert. A dynamic logic for the formal verification of JAVA CARD programs. In I. Attali and T. Jensen, editors, *Revised Papers, JAVA on Smart Cards: Programming and Security, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
3. B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.
4. B. Beckert and W. Mostowski. A program logic for handling JAVA CARD's transaction mechanism. In M. Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference*, volume 2621 of *LNCS*, pages 246–260, Warsaw, Poland, April 2003. Springer-Verlag.
5. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, volume 2083 of *LNCS*, pages 626–641. Springer-Verlag, 2001.
6. B. Beckert and S. Schlager. Software verification with integrated data type refinement for integer arithmetic. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, volume 2999 of *LNCS*, pages 207–226. Springer, April 2004.
7. B. Beckert and P. H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*, pages 91–99. IEEE Press, 2003.
8. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking secure interactions of Smart Card applets. *Journal of Computer Security*, 10(4):369–398, 2002.
9. R. Boyer. Proving theorems about JAVA and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003.
10. E. Bretagne, A. E. Marouani, P. Girard, and J.-L. Lanet. PACAP purse and loyalty specification v0.4. Technical report, GemPlus, January 2001.
11. L. Burdy, A. Requet, and J.-L. Lanet. JAVA applet correctness: a developer-oriented approach. In *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.
12. Z. Chen. *JAVA CARD Technology for Smart Cards.* Addison Wesley, 2000.
13. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proc. SPIN Software Model Checking Workshop*, LNCS, pages 205–223. Springer-Verlag, 2000.
14. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. Technical Report 2004-01, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004.
15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for JAVA. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
16. R. Hähnle and A. Wallenburg. Using a software testing technique to improve theorem proving. In A. Petrenko and A. Ulrich, editors, *Post Conference Proceedings, 3rd International Workshop on Formal Approaches to Testing of Software*

(FATES), Montréal, Canada, volume 2931 of *LNCS*, pages 30–41. Springer-Verlag, 2003.
17. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
18. E. Hubbers and E. Poll. Reasoning about card tears and transactions in Java Card. In *Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.
19. B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology And Software Technology*, volume 3116 of *LNCS*, Stirling, UK, July 2004. Springer.
20. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: ISSS 2003. Revised Papers*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.
21. K. R. M. Leino and R. Stata. Checking object invariants. Technical Note #1997-007, Digital Systems Research Center, Palo Alto, USA, January 1997. Available from `ftp://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-007.ps.gz`.
22. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/Java Card programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.
23. R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.
24. R. Marlet and D. L. Métayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.
25. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.
26. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The Jive system—implementation description. Available from `http://softech.informatik.uni-kl.de/old/en/publications/jive.html`, 2000.
27. W. Mostowski. Rigorous development of Java Card applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002. Available from `http://www.cs.chalmers.se/~woj/papers/room2002.ps.gz`.
28. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.
29. E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants for imperative programs. Available from `http://www.lsi.upc.es/~erodri/ijcar04ex.ps`, November 2003.
30. K. Stenzel. Verification of Java Card Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001.
31. Sun Microsystems, Inc., Santa Clara/CA, USA. *Java Card 2.2.1 Platform Specification*, October 2003.
32. K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, volume 2422 of *LNCS*, pages 334–348. Springer-Verlag, 2002.
33. D. von Oheimb. *Analyzing Java in Isabelle/HOL.* PhD thesis, Institut für Informatik, Technische Universität München, January 2001.
34. P. Ziemann and M. Gogolla. An OCL extension for formulating temporal constraints. Technical Report 1/03, Universität Bremen, Fachbereich für Mathematik und Informatik, 2003.