Reasoning about Card Tears and Transactions in Java Card

Engelbert Hubbers and Erik Poll

SoS Group, NIII, Faculty of Science, University of Nijmegen
{hubbers,erikpoll}@cs.kun.nl

Abstract. The Java dialect Java Card for programming smartcards contains some features which do not exist in Java. Java Card distinguishes *persistent* and *transient* data (data stored in EEPROM and RAM, respectively). Because power to a smartcard can suddenly be interrupted by a so-called *card tear*, by someone removing the smartcard from the reader, Java Card provides a notion of *transaction* to ensure that updates of multiple fields in persistent memory can be performed atomically. This paper describes a way to reason about these Java Card specific language features.

1 Introduction

The Java Card language for programming smartcards has attracted a lot of attention in the formal methods community, especially people working on formal methods for Java. In many respects, it provides an ideal target for formal methods: the language and its API are simple, programs are very small, and their correctness is critical.

However, although the Java Card programming language for smartcards is usually presented as a subset of Java, Java Card has several features in addition to standard Java, which are specific to smartcards. First, Java Card distinguishes the two kinds of memory that are available on smartcards, persistent (EEPROM) and transient (RAM). Second, because a smartcard can be subject to a sudden loss of power due to a so-called *card tear* – namely when the card is removed from the reader – Java Card offers a *transaction mechanism* similar to that found in databases; this enables a programmer to ensure that several updates to memory are performed atomically, i.e. either all the updates are performed or none is. There are more Java Card specific features, but we do not take them into account in this paper.

To accurately reason about the behavior of Java Card programs – and to do program verification – these additional features should be taken into account. Most work on the verification of Java Card programs, with the exception of [1], ignores these special features. Given the complexities of program verification, this can certainly be justified for pragmatic reasons: before we try to verify that a program is correct in the presence of potential card tears, it makes sense to first verify its correctness under the simplifying assumption that no card tears occur and no transactions are ever aborted. However, ultimately we would like to be able to reason about such Java Card features, and this is what we set out to do in this paper.

The context of this work is the verification of Java programs that have been specified with JML [2], using the LOOP tool in combination with the theorem prover PVS. The verification of programs with the LOOP tool ultimately relies on a denotational semantics of Java and JML, for which a Hoare logic and weakest precondition calculus have been developed. In short, the LOOP tool compiles JML annotated Java source code into PVS theories. Proving these theories in PVS implies that it is formally verified that the Java program behaves the way it is specified in JML. For a more detailed overview of this LOOP project, see [3]. One of the achievements of this work has been that a commercial Java Card application has been completely verified, showing that such verifications of real Java Card programs are feasible. Still, our verifications ignore the possibility of card tears, so our next challenge is to take this into account.

To reason about card tears and transactions we need a formal semantics of these features (or a programming logic which takes them into account). Rather than defining a semantics of Java Card including these features from scratch, we will try to desugar Java Card programs with their special features into conventional Java programs, effectively modeling card tears and transactions inside Java. The central trick we use here is that we model card tears as special exceptions, a trick also used in [4,5]. Such a modular approach has several benefits (provided it is successful of course...): it is less work, it is easier to understand, and because it is independent of a particular semantics or programming logic for Java, it will be applicable in many other settings, not just the particular semantics and programming logic of Java that is used in the LOOP project. Modularity is not just a desirable quality for programs, but also for theories about programming languages!

The organization of the rest of this paper is as follows: Sect. 2 explains the peculiarities of Java Card that we want to reason about. Sect. 3, 4 and 5 describe our approach in detail. Sect. 6 says something about the implementation of our idea.

2 Card Tears and Transactions in Java Card

In this section we briefly explain the peculiarities of Java Card as opposed to Java when it comes to card tears and transactions. For a more complete explanation, see [6] or the Java Card Runtime Environment (JCRE) specification [7].

Persistent vs. Transient Memory. Java Card distinguishes two kinds of memory that are available on smartcards, persistent (EEPROM) and transient (RAM)[1]. The main difference is that persistent memory will keep its value when

[1] Smartcards will also have ROM, which is used for pre-installed program code, but this is of no concern to the Java Card programmer.

power is switched off. Java Card objects and their fields are allocated in EEPROM, so the fields of objects will keep their value during a power loss. However, the Java Card API offers methods to allocate arrays in RAM, so-called *transient arrays*. If a field is a transient array, then the contents of this array are lost as soon as power is lost, but the field itself, which is a reference to the piece of RAM allocated for the array, keeps its value as this is stored in EEPROM. Reasons for using RAM rather than EEPROM for (array) field are efficiency – reading and writing RAM is quicker than EEPROM –, the limited lifetime of EEPROM – EEPROM can only support a limited number of writes before the chips stops functioning –, and security – data kept in RAM is harder to spy out and moreover it is lost as soon as power is lost[2]. The stack is also stored in RAM, so the parameters and result of method calls and local variables are all lost as soon as power is lost.

Card Tears. In many card readers it is possible to tear the smartcard out of the reader while it is in operation. Such a so-called *card tear* results in a sudden loss of power. All data stored in RAM is lost when such a card tear occurs. The Java Card platform incorporates a special clean-up when power supply is restored, before any normal action applet operation takes place.

Transactions. To cope with card tears, the Java Card API offers a so-called *transaction mechanism*. This can be used to ensure that several updates to persistent memory are executed as a single atomic operation, i.e. either all updates are performed or none at all. The Java Card API offers three methods for this: `beginTransaction`, `commitTransaction` and `abortTransaction`. After a `beginTransaction` all changes to persistent data are executed conditionally. Note that changes to transient data, including local variables, are executed unconditionally. The transaction is ended by `commitTransaction` or `abortTransaction`; in the former case the updates are committed, in the latter case the updates are discarded. If a card tear occurs during a transaction, any updates to persistent data done during that transaction are discarded. This in fact happens the next time the smartcard powers up during the special clean-up mentioned before.

Example 1 (Java Card sample). Fig. 1 illustrates the use of the transaction mechanism, the use of the API method for allocating a transient array, and the use of JML to specify invariants and postconditions.

Every object of class `A` has a persistent field `p` and a field `t` that is a transient array of length 1. This means that whenever the smartcard loses power, the contents of `t[0]` is lost, but `p` and `t` itself –i.e. the pointer to the position in the RAM memory where `t[0]` is stored– keep their value.

[2] Indeed, for security reasons, the contents of transient arrays can also be cleared automatically at certain events other than card tears, e.g. the de-selection of an applet.

```
class A {
  // persistent field p, allocated in EEPROM
  byte p;
  //@ invariant p % 2 == 0 && 0<= p && p < 10;

  // transient array t, so t[0] is allocated in RAM
  byte[] t = makeTransientByteArray(1,CLEAR_ON_RESET);
  //@ invariant t != null && t.length == 1;
  //@ invariant t[0] % 2 == 0;

  //@ ensures p == \old(p)+2 && t[0] == \old(t[0])+2;
  void m() {
    beginTransaction();
    p++; t[0]++; p++;
    if (p < 10) commitTransaction();
           else abortTransaction();
    t[0]++;
    }
}
```

Fig. 1. Example program using transactions and transient data, with JML specification

There are four JML annotations in the example, written as comments starting with `//@`. This includes three invariants stating that `p` is even, that `t` is not null and has always length 1 and that `t[0]` is also even. This also includes one postcondition (`ensures` clause) for method `m`, stating that the method will increase the values of `p` and `t[0]` by 2. The use of the transaction guarantees that the invariant for `p` will not be broken if the method `m` is interrupted by a card tear. The treatment of transient memory makes sure that the invariants for `t` are not broken by a card tear. Note that the postcondition only relates to normal termination of the method, and does not say anything about what happens if the method 'aborts' because of a card tear.

Incorrect use of this mechanism can result in a `TransactionException` being thrown:

– The transactions cannot be nested. So if a new `beginTransaction` is called within another transaction a `TransactionException` is thrown. Likewise such an exception is thrown if a `commitTransaction` or `abortTransaction` is called while there is no transaction in progress.
  Reasoning about this requires no special machinery, as specifications for enforcing the correct use of the methods for beginning or ending transactions can easily be expressed in JML.
– A `TransactionException` is also thrown if certain hardware limitations are exceeded. Only a finite amount of storage, called the *commit buffer*, is available to keep track of the conditional updates done during a transaction. The size of this commit buffer depends on the specific smartcard hardware. If there are too many updates inside a transaction, and the available space in the commit buffer is exhausted, again a `TransactionException` is thrown.
  We will ignore the possibility of exhausting the commit buffer, and the resulting `TransactionException`. Proving that this never happens is best done in

an ad-hoc manner, i.e. by counting the maximum number of bytes needed in the commit buffer for every transaction in a program and checking that this does not exceed the space of the commit buffer. Including this in a general program logic would be overly complicated. Also, how much space needed in the commit buffer for the bookkeeping associated with an individual update will be specific to the particular implementation of the platform and/or the underlying hardware.

Some (native) classes in the Java Card API provide persistent data which is not subjected to card tears: the counter associated with a `PIN` object, which keeps track of how many incorrect `PIN`s have been entered, is not restored in the event of a card tear. Otherwise the transaction mechanism might allow an unlimited number of guesses for the `PIN` code.

2.1 What Can Go Wrong, and How to Avoid It

Before we consider ways of describing the semantics of card tears and transactions, and how this might be used as a basis for reasoning about these language features, we first approach the issue from a different angle, by investigating what can go wrong if code is subjected to card tears or if it contains transactions, and what could we do to avoid these problems. Or, in other words, what are the properties that we fail to establish in our current verifications of Java Card code, but which we would like to be able to prove.

Invariants. Invariants usually play a crucial role in ensuring that a piece of code behaves correctly. When a card tear occurs, invariants may be left broken as a result. After all, invariants may temporarily be broken during the execution of a method.
Typically, the transaction mechanism is used to prevent card tears from disturbing invariants that involve persistent data, as in Fig. 1.

Postconditions. Just ensuring that invariants are not left broken as a result of a card tear may not be enough to ensure that a method is correct. We may want to establish additional properties. For example, in our example in Fig. 1, we might want to ensure that if method `m` is interrupted by a card tear, it will either leave `p` unchanged of increase `p` by 2, and not say reset `p` to 0, which is allowed by the invariant; in this case we would like to establish `p == \old(p) || p == \old(p)+2` as postcondition of `m` in the event of a card tear.

There are two mechanisms that we can use to ensure that an invariant is not left broken (c.q. an additional postcondition is met) after a card tear occurs:

– the transaction mechanism; e.g., in Fig. 1, the transaction mechanism ensures that the invariant for `p` is maintained in the event of a card tear.
– the clearing of transient memory; e.g., in Fig. 1, the clearing of transient memory ensures that the invariant for `t[0]` is maintained (or, rather, re-established) in the event of a card tear.

The former mechanism is only relevant if an invariant (postcondition) involves transient data, the second mechanism is only relevant if it involves persistent data.

Given the nature of transient data, and the fact that transient data typically serves as scratch-pad memory, it is unlikely that we will be interested in any invariants or postconditions involving transient data. (Indeed, the whole idea of an invariant seems at odds with the notion of transient memory.) So, for many Java Card applications, it will not be necessary to take the clearing of transient memory into account to establish their correctness.

Invariants which only depend on persistent memory can be dealt without trying to formalize the transaction mechanism, in two ways:

– *Ensure that an invariant is never broken.*
  It may seem an overly simplistic approach, but in practice, many invariants are never broken. For example, in Fig. 1, the invariant `t !=null && t.length == 1` will never be broken. This is the approach taken in [1].
  Still, one has to be careful about the notion level of atomicity here. E.g., an invariant `a == b` will be temporarily broken during the execution of the statement `int x = (a++) + (b++)`, even though the invariant will hold before and after execution of the statement if there are no card tears. When reasoning at the level of source code our notion of atomicity will be coarser than what it really is.
– *Ensure that the invariant is never broken outside a transaction.*
  Some invariants will have to be temporarily broken. (E.g. if we are updating two fields and there is an invariant expressing a relationship between these fields, the invariant will typically be broken after updating the first of these fields.) If these invariant involves persistent data, then this should be done inside a transaction.

3 Modeling Card Tears

To model card tears inside Java we use the same trick used in [4,5], i.e. card tears are modeled as a special kind of exception, which can arise at any moment during execution. Like an exception, a card tear is effectively an abrupt change of the flow of control. A difference is that whereas an exception can be caught, a card tear cannot be caught, as there is no VM executing that could execute an exception handler. However, conceptually we can consider the recovery to a card tear that happens the next time the card powers up (i.e. the undoing of any unfinished transaction and the clearing of all transient data) as the exception handler for a card tear exception. We introduce a special exception class `CardTearException` for modeling card tears[3].

[3] Strictly speaking, `CardTearException` should not be an `Exception`, but rather an `Error`, because we clearly do not want `CardTearExceptions` to be caught by any existing `try-catch` blocks in a program. However, using `Error` would introduce a problem in JML.

3.1 Throwing a `CardTearException`

There are several ways to account for the possibility of a `CardTearException` being thrown at any moment during execution, namely at syntactical level, at semantic level, or at logical level:

a). One possibility is to do this purely syntactically, by desugaring any sequence of statements, e.g.

```
S1; S2;
```

to include calls to a method `possibleCardTear` before and after each statement, e.g.

```
possibleCardTear(); S1; possibleCardTear(); S2; possibleCardTear();
```

where `possibleCardTear` is a method which either performs a `skip`, or throws a `CardTearException`. We can even give a possible implementation of this method `possibleCardTear` in Java, for instance

```
possibleCardTear() {
  if (cardtear_counter-- < 0) throw new CardTearException();
}
```

where `cardtear_counter` is a global (i.e. final static) variable, initialized to an unknown value.
Such a syntactic approach has its limitations, namely the level of atomicity of statements that we can distinguish at the level of source code syntax. This notion of atomicity is coarser than it is in reality. E.g. in the example above we treat the statements `Si` as atomic, whereas in reality only individual byte code operations are atomic. For example, a statement such as `int x = (a++) + (b++)` would have to be rewritten into `a++; b++; int x=a+b;` if we want to include possible card tears after incrementing `a` or `b`[4].

b). Instead of modeling the possibility of card tears syntactically, as sketched above, an alternative would be to redefine our semantics of Java to include card tears. For instance, in the LOOP project we use a denotational semantics, and we could redefine the semantics of composition ; and increment operation `++` to include the possibility of an exception being thrown. Effectively, this comes down to for instance changing the semantics of composition ; to the composition of $\hat{;}$, where $S_1 \hat{;} S_2$ is defined as

$$possibleCardTear(); S_1; possibleCardTear(); S_2; possibleCardTear();$$

c). Another possibility of modeling card tears is at the logical level, i.e. in the logic used to reason about programs. For instance, if our reasoning about Java programs uses some Hoare logic, we could adapt all Hoare rules to allow for the possibility of card tears. Effectively, this comes down to for instance replacing the Hoare rule for composition ; by the Hoare rule for $\hat{;}$.

[4] Still, Java Card does not support the data types `double` and `long`, for which assignments are by definition non-atomic; see [8], section 17.4.

For the remainder of this paper, we leave it open which of the mechanisms above is used to model the possibility of card tears. Clearly, introducing explicit calls to `possibleCardTear()` at all program points quickly makes programs unreadable, so we prefer to leave the possibility of card tears being thrown implicit.

3.2 Specification and Verification Using `CardTearException`

Modeling card tears as exceptions is useful both when it comes to verifying and specifying Java Card code.

An `invariant` in JML has to hold if a method throws an exception. So an immediate consequence of modeling card tears as exceptions is that to verify a method we must ensure that invariants hold at every program point, as discussed earlier in Sect. 2 (and as in the approach of [1]).

Another advantage of treating card tears as exceptions is that it becomes possible to specify the behavior in the event of a card tear in JML, as mentioned as a wish in Sect. 1. This is not possible in the approach of [1]. For example, we could specify the behavior of the method `m` from Fig. 1 as follows:

```
//@ ensures p == \old(p)+2 && t[0] == \old(t[0])+2;
//@ signals (CardTearException) (p == \old(p) || p == \old(p)+2)
//@                              && t[0] == 0;
void m() throws CardTearException{ ... }
```

Here the JML keyword `signals` is used to specify an *exceptional postcondition*, i.e. a condition that should hold after a certain exception occurs. Note that here we assume that the undoing of any unfinished transaction and the resetting of the transient memory occurs immediately after a card tear occurs, so that this has occurred before we exit the method.

Example 2 (specifications for arrayCopy(NonAtomic)). More example specifications that use the notion of `CardTearException` are given in Fig. 2. Here specifications are given for the Java Card API methods `arrayCopy` and `arrayCopyNonAtomic`. These two methods are interesting examples because the only difference between them is what happens when a card tear occurs during their execution. The former method is atomic, so either all array entries are copied, or none are. The latter method is not atomic, so some array entries may be copied whereas others are not. The JML specifications in Fig. 2, more in particular the `signals` clauses, make this difference precise. Note that the specification of `arrayCopyNonAtomic` makes no assumptions on the order in which the array elements are copied.

4 Modeling the Clearing of Transient Memory

We now consider how to model the clearing of transient memory in the event of a card tear. Because transient data is completely unaffected by transactions, we can consider this issue in isolation, without taking into account how we model the transaction mechanism.

```
/*@   requires src != null && dest != null &&
  @            srcOff >= 0 && destOff >= 0 && length >= 0 &&
  @            srcOff+length <= src.length && destOff+length <= dest.length;
  @
  @ assignable dest[destOff..destOff+length−1];
  @
  @    ensures (\forall short i; 0 <= i && i < length
  @                           ; dest[destOff+i] == \old(src[srcOff+i]));
  @    signals (CardTearException)
  @               (\forall short i; 0 <= i && i < length
  @                               ; dest[destOff+i] == \old(src[srcOff+i]))
  @            || (\ forall short i; 0 <= i && i < length
  @                               ; dest[destOff+i] == \old(dest[destOff+i]));
  @*/
 native public static final short arrayCopy(byte[] src,
                                            short srcOff,
                                            byte[] dest,
                                            short destOff,
                                            short length);
/*@   requires ...
  @ assignable ...
  @    ensures ...
  @
  @    signals (CardTearException)
  @             (\forall short i; 0 <= i && i < length
  @                          ;    dest[destOff+i] == \old(src[srcOff+i])
  @                            || dest[destOff+i] == \old(dest[destOff+i]) );
  @*/
 native public static final short arrayCopyNonAtomic(byte[] src,
                                                     short srcOff,
                                                     byte[] dest,
                                                     short destOff,
                                                     short length);
```

Fig. 2. JML specifications for the API methods `arrayCopy` and `arrayCopyNonAtomic`. In the latter only the differences with the former are shown.

We model the clearing of transient memory by enclosing every method in a `try-catch`, where in the `catch` the transient memory is cleared, i.e. reset to the initial default for that type. For the code from Fig. 1, this desugaring results in:

```
 class A {
   // persistent field p, allocated in EEPROM
   byte p;
   //@ invariant p % 2 == 0 && 0<= p && p < 10;

   // transient array t, so t[0] is allocated in RAM
   byte[] t = makeTransientByteArray(1,CLEAR_ON_RESET);
   //@ invariant t != null && t.length == 1;
   //@ invariant t[0] % 2 == 0;

   //@ ensures p == \old(p)+2 && t[0] == \old(t[0])+2;
   void m() {
     try {
       beginTransaction();
       p++; t[0]++; p++;
       if (p < 10) commitTransaction();
              else abortTransaction();
       t[0]++;
     }
     catch (CardTearException e) {
       for (int i = 0; i < t.length; i++) t[i] = 0; // clear transient array t
       throw e; // re−throw the exception
     }
   }
 }
```

Note that in this particular example the explicit clearing of the transient array `t` in the event of a `CardTearException` will re-establish the invariant `t[0] % 2 == 0`.

One subtlety in the desugaring above is that during the clearing of transient memory in the `catch` block we do not want to allow card tears.

The only question in this desugaring is deciding which transient arrays a method should clear. The easiest way to decide this is to look at the postconditions (i.e. the `ensures` and `signals` clauses in conjunction with any `invariant`s) that we want to prove for the method. Letting every method clear only the transient fields mentioned in its postconditions is sufficient. If a method calls another method, and a card tear occurs in this inner method call, this may lead to transient arrays being cleared several times, but as this clearing is clearly an idempotent operation, this is not a problem.

The only problem that can arise is when a specification refers to a transient field of another object to which the current object does not have access. In JML specifications the normal visibility constraints imposed by the Java modifiers (such as `private` or `protected`) can be loosened up, so it is possible for a JML specification of a method to mention a transient field `o.t[0]` of some other object `o` even though this field is not accessible from within that method. To cope with this, the object `o` in question would have to be extended to provide a method `clearTransients()` that clears its transient fields.

We should stress again that in Java Card applets transient data typically serves as scratch-pad memory, so that it is unlikely that we are interested in any invariants or postconditions involving transient data.

Note that both specifications in Fig. 2 exclude the effect of clearing transient memory: the `signals` clauses of `arrayCopy` and `arrayCopyNonAtomic` do not state that if `dest` or `src` are transient arrays their contents will have been cleared in the event of a card tear. We could modify the specifications to include this, by introducing a further case distinction in the `signals` clause on whether the arrays in question are transient or not, and including

```
(JCSystem.isTransient(dest) != JCSystem.NOT_A_TRANSIENT_OBJECT)
==>
  (\forall i; 0 <= i && i < dest.length; dest[i] == 0)
```

in the `signals` clause, and a similar statement for `src`. Here we use the Java Card API method `isTransient`, which can be use to test if an array is transient.

However, conceptually it is much more convenient not to make `arrayCopy` or `arrayCopyNonAtomic` responsible for clearing the arrays `src` and `dest` if they are transient, but to leave it up to the methods calling `arrayCopy(NonAtomic)`.

5 Modeling Transactions

We now turn to the issue of modeling transactions in Java. This comes down to the question of how conditional updates to persistent fields can be modeled in such a way that they can be undone in the event of an aborted transaction, caused by a card tear or by an invocation of `abortTransaction`. We do this by mimicking the way this can be implemented in hardware. Such an implementation involves some extra bookkeeping for persistent fields that are changed during a transaction. Two values will have to be recorded for these fields: the ‘new’, updated value, as well as the ‘old’ value the field had at the start of the transaction. There are roughly two strategies for doing this, as discussed in [9]. Suppose a persistent field `x` is modified during a transaction. One strategy, the *optimistic* strategy, is to log the old value of `x` at the beginning of the transaction, and use the logged value in the event of an aborted transaction to restore `x` to its original value. The other, *pessimistic*, strategy is to work on a temporary copy of a persistent field `x` during the transaction, and copy this updated version of `x` back to `x` when the transaction is committed. The optimistic approach entails some extra work in case the transaction is aborted, the pessimistic approach entails some extra work in case the transaction is committed. The Sun specification does not say anything which rules out or favors one of these two approaches. We will use the optimistic approach, introducing an extra ‘backup’ field *x*`bak` for every field *x*, but we could just as easily have used the pessimistic approach.

Below we show how the code given in Fig. 1 can be desugared to model the transaction in this way. As discussed in Sect. 3, we assume that at anytime the special `CardTearException` can be thrown.

```
class A {
  byte p = 0;
  byte pbak;                          // backup value of p
  //@ invariant p % 2 == 0 && 0 <= p && p <= 10;

  byte[] t = makeTransientByteArray(1,CLEAR_ON_RESET);
  //@ invariant t != null && t.length == 1;
  //@ invariant t[0] % 2 == 0;

  static boolean inTransaction = false;

  //@ ensures p == \old(p)+2;
  //@ signals (CardTearException) p == \old(p) || p == \old(p)+2;
  void m() {
   try {
        pbak = p;                     // backup p
        if (inTransaction) TransactionException.throwIt(IN_PROGRESS)
        inTransaction = true;         // begin transaction
        p++; t[0]++; p++;
        if (p < 10)
            inTransaction = false;    // commit transaction
        else {
            p = pbak;                 // restore old value of p
            inTransaction = false;    // abort transaction
        }
        t[0]++;
    } catch (CardTearException e) {
      if (inTransaction) {
        p = pbak;                                           // restore old value of p
        for (int i = 0; i < t.length; i++) t[i] = 0;  // clear transient array t
        throw e;                                            // re-throw the exception
      }
    }
  }
}
```

The changes to the code are:

- An extra field `pbak` is introduced for the bookkeeping of the old value of `p` during a transaction.
- A static field (i.e. a global variable) `inTransaction` is introduced to record whether a transaction is in progress or not.
- The entire method is included in a `try-catch` construction, which, in case of a card tear, undoes the effects of any transaction if a transaction was in progress.
- Any calls to `begin-`, `commit-`, and `abortTransaction` are replaced by a code fragments which set `inTransaction`, and backup or restore the value of `p`.

In general, at the place where the `commit-` or `abortTransaction` we should check that a transaction is indeed in progress, or else throw a `TransactionException`, by including

```
if (!inTransaction) TransactionException.throwIt(NOT_IN_PROGRESS);
```

We have omitted this in the example above because it is obvious that here this situation does not arise.

One subtlety in the desugaring above is that during the bookkeeping associated with restoring an aborted transaction we should not allow card tears.

Although the example above is a very simple one, we believe that this desugaring of transactions can be used for most Java Card programs.

Similar to the modeling of the clearing of transient memory, the only difficult issue in this desugaring of transactions is deciding for which persistent fields should be restored. This issue can be solved in exactly the same way: only the persistent fields mentioned in the invariant and pre- and postconditions are relevant for the verification of an individual method, and only for these do we have to restore the old values in the event of a card tear, i.e. in `catch` block at the end of a method, and in the event of an `abortTransaction`. If a method calls another method, and a card tear occurs in this inner method call, this may lead to persistent fields being restored several times, but as this restoring is clearly an idempotent operation, this is not a problem.

Similar to the modeling of the clearing of transient memory, the only problem that can arise is when a specification refers to a persistent field of another object to which the current object does not have access. As we have mentioned before in JML specifications the normal visibility constraints imposed by the Java modifiers can be loosened up, so it is possible for a JML specification of a method to mention a persistent field `o.p` of some other object `o` even though this field is not accessible from within that method. To cope with this, the object `o` in question would have to be extended to provide methods `backupPersistents()` and `restorePersistents()` to backup and restore the values of its persistent fields.

6 Implementation

Figure 3 provides an overview of our project. We have described how the preprocessor can simulate some Java Card issues into a Java model. So far, we have not tried to mechanize the desugaring we have introduced in this paper.

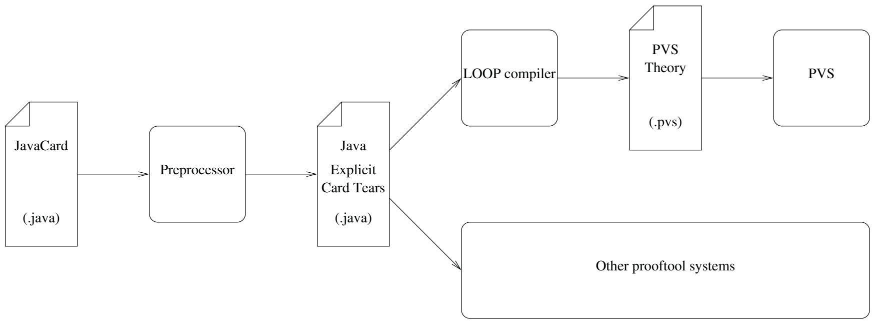

Fig. 3. Inbedding of Java Card transactions into Java based proof checking systems

It is quite easy to do the desugaring for the clearing of transient memory and for transactions by hand. The main question is how to deal with `possibleCard-Tear`. As discussed in Sect. 3.1, there are three ways of dealing with this. The last two of these, i.e. b) and c), require a large amount of work, namely redefining the semantics of all Java Card constructs, reformulating and reproving all Hoare rule, or redefining the whole weakest precondition strategy. In our LOOP/PVS setting the amount of work needed would be huge. Therefore it seems most practical to go for the syntactical approach a).

Furthermore, when verifying Java code in our LOOP/PVS setting, it seems wise to at least start with the more pragmatic approaches mentioned at the end of Sect. 2.1, and just verify that invariants are never broken outside transactions, i.e. the approach that is also taken in [1].

7 Remaining Issue

We believe that our model faithfully formalizes Sun's official specification of the transaction mechanism, with the possible exception of the issue discussed below.

One unclarity in Sun's specification of the Java Card API that we came across concerns the 'non-atomic' methods `arrayCopyNonAtomic` and `arrayFill-NonAtomic` in the API class `javacard.framework.Util`. Indeed, as is often the case, the main value of our formalization may well be to reveal such potential ambiguities in the informal specification.

The API specification of these methods says that

> "This method does not use the transaction facility during the copy operation even if a transaction is in progress. ..."

The precise meaning of this is not clear. For example, suppose we have two persistent arrays `p` and `q` and that during the execution of

```
p[0] = 0; q[0] = 2;
beginTransaction();
  p[0]++;
  arrayCopyNonAtomic(q,0,p,0,1);
endTransaction();
```

a card tear occurs immediately after the `arrayCopyNonAtomic`. In that case, it is not clear if `p[0]` will be restored to `0`, as our semantics would predict, or to `1`. In the former case all side-effects to `p[0]` during the transaction are undone, in the latter case the side-effect of `arrayCopyNonAtomic` is not undone. Experiments on actual smarts[5] confirms that the former happens. However, the quote above could be interpreted to mean that the latter should happen.

We do not have the space to discuss all the implications of this issue here, and, moreover, we are still investigating it, but we will make a report available about the outcome, so check our webpages for additional information.

8 Conclusion

We have shown how Java Card features such as card tears, transactions, and transient as opposed to persistent memory can be faithfully modeled inside Java, making it possible to use existing programming logics for Java to reason about these features. An advantage of the approach is that it is to a large extent independent of the Java semantics being used. An added benefit of such a model inside Java is that it is understandable to a larger audience – the desugarings should provide anyone with a good knowledge of Java with a clear understanding of the semantics of card tears and transactions, and with useful basis about reasoning about the features – and that we can use standard JML to specify properties of these features.

A disadvantage of the approach is that it is somewhat ad-hoc. A definition of a semantics for Java Card from scratch, be it denotational or operational semantics, or an axiomatic semantics as Hoare logic or weakest precondition calculus, would provide less ad-hoc and more rigorous semantics. Furthermore, it is somewhat unsatisfactory that we have to assume that during our event handling code there will not be another card tear.

It is important to realize that Java Card programs are extremely simple programs, without much complicated class hierarchies or OO structure to speak of. In many applications the only object, apart from some byte arrays used as fields, is a single object of class `javacard.framework.applet`. For such programs it is trivial to decide at compile time what the relevant persistent and transient data at runtime will be.

To our knowledge, the only other work on Java Card that does not ignore transactions and card tears is [1]. The approach presented there can be used to

[5] IBM JCOP 2.1.1 cards, 21id and 31bio to be precise.

prove that certain invariants are never broken, but cannot establish postconditions in the event of a card tear, as discussed in Sect. 2.1, or distinguish between the atomic and non-atomic API methods for copying arrays discussed in Ex. 2.

Unfortunately, the Java model is quite unreadable if it takes all explicit card tears into account. The implementation of the existing `TransactionExceptions` also really decreases readability, which makes this method less practical. On the other hand, knowing these syntactical problems it should not be too difficult to implement some of these solutions into the semantics of the proof system. For our own LOOP tool we are sure that this can be done.

Acknowledgment

Thanks to Joachim van den Berg of TNO-EIB for carrying out experiments on smart cards to confirm our own experiments to resolve the semantics of the 'non-atomic' methods discussed in Sect. 7.

References

1. Beckert, B., Mostowski, W.: A program logic for handling Java Card's transaction mechanism. In Pezzè, M., ed.: Fundamental Approaches to Software Engineering, FASE'2003. Volume 2621 of Lecture Notes in Computer Science. (2003) 246–260
2. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., Poll, E.: An overview of JML tools and applications. In Arts, T., Fokkink, W., eds.: Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03). Volume 80 of Electronic Notes in Theoretical Computer Science (ENTCS)., Elsevier (2003) 73–89
3. Jacobs, B., Poll, E.: Java program verification at Nijmegen: Developments and perspective. Technical Report NIII-R0318, Dept. of Computer Science, Univiversity of Nijmegen (2003)
4. Hartel, P.H., de Jong Frz, E.K.: A programming and a modelling perspective on the evaluation of Java card implementations. In Attali, I., Jensen, T., eds.: 1st Java on Smart Cards: Programming and Security (e-Smart). Volume 2041 of LNCS., Springer-Verlag, Berlin (2000) 52–72
5. Poll, E., Hartel, P., de Jong, E.: A Java reference model of transacted memory for smart cards. In: Fifth Smart Card Research and Advanced Application Conference (CARDIS'2002), USENIX (2002) 75–86
6. Chen, Z.: Java Card Technology for Smart Cards. The Java Series. Addison-Wesley (2000)
7. Sun: Java Card 2.1 Runtime Environment (JCRE) Specification. Sun Micro systems Inc, Palo Alto, California (1999) `http://java.sun.com/products/javacard/`.
8. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996)
9. Oestreicher, M.: Transactions in Java Card. In: 15th Annual Computer Security Applications Conf. (ACSAC), Phoenix, Arizona, IEEE Comput. Soc, Los Alamitos, California (1999) 291–298
`http://www.acsac.org/1999/abstracts/thu-b-1500-marcus.html`.