Formal specification of Gemplus' electronic purse case study

Néstor Cataño and Marieke Huisman

INRIA Sophia-Antipolis, France
{Nestor.Catano, Marieke.Huisman}@sophia.inria.fr

Abstract. This paper presents a case study in formal specification of smart card programs, using ESC/Java. It discusses an electronic purse application, provided by Gemplus, that we have annotated with functional specifications (*i.e.* pre and postconditions, modifies clauses and class invariants), that are as detailed as possible. The specification is based on the informal documentation of the application. The implementation has been checked *w.r.t.* the specification, using ESC/Java. This revealed several errors or possibilities for improvement in the source code (*e.g.* removing unnecessary tests).
Our paper shows that a relatively lightweight use of formal specification techniques can already have a serious impact on the quality of a program and its documentation. Furthermore, we also present some ideas on how ESC/Java could be further improved, both *w.r.t.* specification and verification.

1 Introduction

Background When developing a large software application, a significant part of the work is spent on writing clear and concise documentation. This documentation serves several purposes: it helps the developers of the application (to understand the implementation decisions taken by a colleague, but also to understand ones own decisions after a certain period of time), and it also is useful when somebody else builds a new application, using features provided by the application at hand.

However, such program documentation is only useful if it correctly describes the implementation, thus one would like to have some trust in its appropriateness. A way to achieve this is to write a formal specification, *i.e.* a description of the program behaviour in logic, and then prove the correctness of the implementation *w.r.t.* this specification, but this is difficult (as it requires a good understanding of the semantics underlying the specification and programming language), and labour-intensive (see *e.g.* [7] for an example of full program verification). Thus, although it is feasible to do formal specification and verification, the costs in general do not outweigh the benefits.

Several suggestions have been made to overcome these problems. First of all, to encourage application developers to write formal specifications, specification languages have been developed which are close to programming languages. The annotation language for Eiffel [14] is the first example of such a specification language, and recently several annotation languages for Java have been proposed, following the same strategy: JML [10], ESC/Java [4], and the Jass annotation language [8]. For JML and ESC/Java, effort has been put into making these specification languages converge [3], so that the respective tools can be used for both languages. Typical for these languages is that expressions are written as Java expressions, extended with some specification-specific constructs.

Secondly, together with the ESC/Java language, a static checker has been developed [4] which can be used to check simple, but useful program properties. This static checker tries to check whether a program satisfies its annotations, by using a dedicated, automated theorem prover. This automated theorem prover has been fine-tuned to find common programming problems like `Nullpointer-Exceptions`, and `ArrayIndexOutOfBoundExceptions`, but it also can be used to check other annotations. If the theorem prover cannot establish that a certain specification is satisfied, ESC/Java issues a warning. Such a warning does not necessarily mean that the program is wrong, as the ESC/Java approach is neither sound, nor complete. When designing the tool, a compromise has been made between soundness, completeness, and efficiency. The result is an efficient checker, that can increase the confidence in the correctness of programs, and that finds many common programming errors. However, if one wishes to establish formally the correctness of a complicated algorithm, other verification techniques have to be used, as advocated in *e.g.* the LOOP project [12] or the Jive project [15]. But even for such complex algorithms it pays off to use ESC/Java first, in order to find a first approximation of the errors in the algorithm and/or specification, before diving into the complete formal verification.

Annotating programs with ESC/Java specifications can thus be helpful to create reasonably quick, clear and concise documentation for a software application, which is according to the implementation.

This paper To demonstrate the usefulness of this approach, this paper describes the ESC/Java annotation of a smart card application case study, which is the implementation of an electronic purse. The original source code of this case study comes from Gemplus [6]. In this paper we discuss the annotations of the source code, and several possibilities for improvement that we encountered.

The result of this work does not give a fully verified specification, but it gives a reasonable description of the electronic purse implementation, which could serve as a basis for further formal verification, *e.g.* by using the LOOP compiler. The main contribution of this paper is that it shows that by making light weight use of formal verification techniques, it is feasible to *(i)* write a formal specification of an application, and *(ii)* have the implementation checked *w.r.t.* the specification so as to increase confidence in the correctness of the implementation. When specifying the purse we have found several (simple) properties which are informally documented, but are not preserved by the implementation. It is straightforward to formally specify these properties and ESC/Java immediately finds the places where these properties are not preserved in the implementation.

Furthermore, to the best of our knowledge this case study is one of the first larger case studies using ESC/Java, and we found several points for improvement in the static checker and its specification language. This lead us to a wish list of improvements in the specification language and to the development of a checker for so-called modifies clauses. This checker will be described in a separate paper.

The rest of this paper is organised as follows. Section 2 describes the general outline of the case study. Section 3 gives a quick introduction to ESC/Java. Section 4 describes the annotations of the purse in more detail, and discusses several interesting aspects of the specification. Section 5 comments on the use of ESC/Java and gives suggestions for improvement. Finally, Section 6 gives conclusions and presents future work.

2 General outline of the Electronic Purse

The electronic purse is a JavaCard application [9], delivered as an advanced smart card programming case study by Gemplus [6]. A JavaCard smart card is capable of running programs developed in JavaCard, a dialect of standard Java. JavaCard does not provide concepts such as dynamic class loading, security management, multi-threading and synchronisation, object cloning and large primitive date types (float, double, long and char). JavaCard applications are called *applets*. The electronic purse applet provides the ability to perform banking operations to the card holder. Typical operations are credit, debit, and changing the currency.

The *debit* operation. Debit operations that involve an amount greater than `maxDebitWOPIN` are protected by a pin code. During one session, a user can do several transactions by presenting his pin code only once. To protect the card against attacks, the number of transactions that can be performed without presenting a pin code is limited to `maxTransactionWOPIN`.

The *credit* operation. If the balance on the card is not sufficient to execute a certain debit operation, the balance can be increased by performing a credit operation. To do this, the terminal asks the bank for credit permission. If the permission is obtained, the account is credited and a confirmation is sent to the bank.

The *currency change* operation. The balance of the purse is expressed in a certain currency. When the card holder travels, the current currency can be changed. In order to do this, the terminal requests a new exchange rate and a certificate from the bank . The purse verifies that the bank is really the expected bank and validates the exchange rate. After changing the balance value, the purse must modify all variables related to the currency.

The purse applet interacts with so-called *loyalty applets* (implementing *e.g.* a frequent flyer program) that may be present on the card. Within the loyalty applet, the card holder gets loyalty points having made certain purchases, and these points can be used later to make other purchases. Further, a *card issuer applet* should be available on the card, which can initialise the purse. Finally, the purse applet also communicates with the point of sale terminal.

The purse application consists of three packages: `utils`, `purse` and `pacapinterfaces`. The `utils` package implements basic classes such as `Annee` (year), `Mois` (month) `Jour` (day), and `Decimal` (floating point numbers). The `pacapinterfaces` package declares sharable interfaces which enable the purse applet to communicate with *e.g.* the loyalty applets on the card. The `purse` package is the core of the application. It contains the class `PurseApplet`, which manages the operations related with installation, selection and deselection, and which communicates with the point of sale terminal. The basic purse functionalities are implemented in the class `Purse`. This class performs the communication with the loyalty applets, using the interfaces described in the `pacapinterfaces` package. Also, this class keeps track of the balance of the purse, the transactions done by the purse (stored as a `TransactionRecord`), the different currency changes that have taken place (in a `ExchangeRecord`) and the different loyalty programs that the card holder is subscribed to (in a `LoyaltiesTable`).

Certain operations can only be performed by a restricted set of users, *e.g.* because a pin code is needed. The class `AccessCondition` defines the different access conditions, and the class `AccessControl` binds the access conditions to the operations. So, when a card holder intends to perform a certain operation, the purse application will check that the card holder has the appropriate permissions. The class `Currencies` stores the different currencies used by the purse application. Finally, the purse application contains several classes implementing cryptographic concepts, namely, `PacapCertificate`, `PacapCipher`, `PacapKey`, `PacapRandom`, `PacapSecureMessaging` and `PacapSignature`. These classes are not studied in full detail in this case study.

3 Static checking of Java programs

ESC/Java is a verification tool developed at Compaq SRC, which permits a user to find common errors in Java programs. The basic idea is that a user specifies the desired behaviour of a class and its methods and the ESC/Java tool checks whether the implementation satisfies the specification. If it cannot establish this, it issues a warning.

The specifications are given as *preconditions*, *postconditions* and *class invariants*. The properties are specified as Java expressions, enriched with several specification-specific constructs. Here, we present some Esc/Java constructs (called pragmas), together with an example of their use. Their full description can be found in [11].

Pragmas for specifying methods

- `requires P`. This pragma specifies a precondition `P`. When Esc/Java checks the body of the routine, it assumes that `P` holds initially, but when Esc/Java

```
/*@
  modifies nbData, data[nbData];
  ensures (\old(nbData) < MAX_DATA) ?
            (nbData == \old(nbData) + 1 && data[\old(nbData)] == cur) :
            (nbData == \old(nbData));
*/
void addCurrency(byte cur){
   if(nbData < MAX_DATA) {
      data[nbData] = cur ;
      nbData++ ;
   }
}
```

Fig. 1. Example ESC/Java specification

checks a method call, it will issue a warning if it can not establish that `P` holds at the call site.
- **`ensures`** `Q`. This pragma specifies a postcondition `Q`. The postcondition is supposed to hold if the method terminates normally, *i.e.* without throwing an exception.
- **`exsures (E)`** `R`. This pragma specifies a exceptional condition. This condition is supposed to hold if the method finishes abruptly, if the exception `e` that is thrown is a subclass of `E`.
- `modifies L`. This pragma specifies that a method *may* only modify any of the state components listed in `L`, the formal parameters of the method and the variables local to the method.

Specification expressions

- `==>` is the logical implication. So, `P ==> Q` is true if and only if `P` is false or `Q` is true, where `P` and `Q` are specification expressions of `boolean` type. Furthermore, `<==>` represents the equivalence relation and `<=!=>` specifies non-equivalence.
- (`\forall T V; E`) and (`\exists T V; E`) are quantifier expressions (of type `boolean`). The first one denotes that `E` is true for all substitutions of values of type `T` for the bound variable `V`. The second one denotes that `E` is true for a substitution of a value of type `T` for the bound variable `V`.
- `\old(`E`)` is used within a postcondition, where it denotes the value of `E` in the pre-state of the method invocation.
- **`\result`** represents the value returned by a non-void method method. It can only be used within an `ensures` clause.

Figure 1 shows a typical ESC/Java annotation example. This example was taken from the specification of the electronic purse. The `addCurrency` method belongs to the class `Currencies`. This class stores all currencies supported by the purse application. The method `addCurrency` adds a new currency to the list of valid currencies. This list is represented by the array `data`. The `modifies` clause declared in the method's header specifies that this method may modify `nbData` and `data` in the position `nbData`[1]. The postcondition of the method `addCurrency` (written as an `ensure` clause) expresses that if `nbData` has not yet reached the threshold value `MAX_DATA`, `nbData` will increase its value by one and the value of the formal parameter `cur` will be assigned to `data[\old(nbData)]`. Inside the postcondition, the expression `\old(nbData)` refers to the value of `nbData` before the method invocation.

4 Specification of the Electronic Purse

4.1 The general specification approach

ESC/Java forces one to start writing specifications for the classes that are 'used' by many other classes, either because they are used as components or because they are inherited from. In the electronic purse case study most classes inherit directly from classes as *e.g.* `Object`, `Exception` or – in the case of interfaces – `Shareable`, so the inheritance structure is not very complex. Therefore, we started by specifying classes that provide basic (and general) features, *e.g.* those in the `utils` package, that are used by the classes in the `purse` package. The specifications for these basic classes form the basis for the specification of the more application-specific classes, so it is important that they are sufficiently detailed.

For every method, we specify the precondition (`requires` in ESC/Java), the postcondition (`ensures`), the modifies clause (`modifies`), and the exceptional postcondition (`exsures`). ESC/Java does not have a keyword to specify that a method may not modify any variables, but this is implied by the absence of a modifies clause. To make our specifications explicit about this, in such a case we added a comment `modifies \nothing;` – as in JML. Further, ESC/Java requires that every exception that is mentioned in the exceptional postcondition, is also mentioned in the `throws` clause of the method. To avoid having to add `throws` clauses to every method, in many cases we chose to have the assertion `exsures (Exception) false;` as a comment, without having it checked by ESC/Java. However, everywhere where there can be any doubt about the correctness of the `exsures` clause, we add the `throws` clauses and have it checked by ESC/Java.

When writing method specifications, two different approaches can be taken: either a precondition is given which ensures that no exceptions will be thrown, or one specifies a light precondition (*e.g.* true), and an exceptional postcondition which describes under which conditions an exception will be thrown. For example, given the left specification below, one has to show that `P` is satisfied before the method is called, and then it is guaranteed that the method cannot

[1] More precisely, it specifies that the method only may modify these instance variables and the local variables and formal parameters of the method.

produce an exception, while the right hand specification makes no requirements on the method call, but specifies that if an exception occurs, this is because `P` did not hold.

```
/*@ modifies M;
    requires P;
    ensures Q;
    exsures (E) false;
*/
void m() {
..
}
```

```
/*@ modifies M;
    requires true;
    ensures Q;
    exsures (E) !P;
*/
void m () {
..
}
```

In our specifications, we usually follow the first approach, which is close to the *design by contract* approach [14], unless the informal documentation clearly suggests that the second approach is intended.

Furthermore, we specify appropriate class invariants for each class, typically restricting the set of legal values for the instance variables. In some cases, the class invariant immediately follows from the informal documentation (*e.g.* the documentation in class `Decimal` states `the decimal part must be done in the interval [000,999]`), and in other cases the appropriate class invariant follows from closer inspection of the code, *e.g.* a variable is never `null`. Section 4.2 discusses the specification of class invariants in more detail.

Sometimes discrepancies between the informal documentation and the implementation occur. In general we try to follow the informal documentation, and we correct the implementation where necessary (and document these changes). In several cases we consulted the case study developers at Gemplus, to get a better understanding which behaviour was actually intended.

In the case study, several functions from the JavaCard API [16] are used. When we specify methods using API functionalities, we use the API specification as constructed by Erik Poll and Hans Meijer (see *e.g.* [13]). In the classes `Purse` and `PurseApplet`, several classes are used that we do not have access to. To overcome this problem, we construct specification files, specifying optimistically (*cf.* [5]) the methods and fields that we need, but without giving an implementation.

Our aim is to give a functional specification of the behaviour of the purse. However, we did not study the algorithms to manage secret keys, and therefore we only give a light weight specification (*i.e.* specifying the precondition and modifies clauses, but no postcondition) of the classes dealing with key generation and certification. This enables us to write and check the specifications of the classes `Purse` and `PurseApplet`. How to specify and verify cryptographic algorithms is a topic of future research.

We aim at giving specifications which produce a minimal amount of ESC/Java warnings, but not all warnings can be avoided. Typically, if a complex control structure occurs in a method (*e.g.* loops in which method calls are made), ESC/Java is unable to establish complicated postconditions. If one wishes to certify these methods, other verification techniques, as advocated *e.g.* in the LOOP

```
/*@
  requires d != null;
  ensures \result == (intPart>d.intPart ||
                     (intPart == d.intPart &&
                        (decPart == d.decPart||decPart > d.decPart)));
*/
public boolean isGreaterEqualThan(Decimal d){
  boolean resu = false ;
  if(intPart>d.getIntPart()) resu = true ;
  else if(intPart<d.getIntPart()) resu = false ;
  else if(intPart==d.getIntPart()){
    if((decPart>d.getDecPart()) || (decPart>d.getDecPart())) resu=true ;
    else if(decPart<d.getDecPart()) resu = false ;
  }
  return resu ;
}
```

Fig. 2. Method `isGreaterEqualThan`

project [12], should be used. However, in the case study at hand such complex control structures are not very frequent and ESC/Java is able to check most of the specifications without any problems.

At [2] the full annotated version of the purse case study can be found. In the code it is documented which postconditions cannot be established by ESC/Java. It is also documented which changes we have had to make to the code.

4.2 Interesting aspects of the specification

Below, several interesting aspects of the specification are discussed in more detail. First we elaborate on some implementation errors that we found in the purse application. Then we discuss the specification of implicit class invariants, and how this can help to simplify the code. Finally, we discuss some miscellaneous aspects of the case study, and present some possible improvements. The problems that we have found probably also would have been found by doing thorough testing, but using theorem proving techniques one is sure not to forget some cases. Also, writing the formal specifications forces one to think very precisely about the intended behaviour of programs, which helps in finding errors.

Implementation mistakes This section presents some examples of common programming errors, and how we found them using ESC/Java.

The isGreaterEqualThan method The class `Decimal` represents a floating point number composed of a decimal part and an integer part, represented by instance variables `decPart` and `intPart`, respectively. The method `isGreaterEqualThan`

```
/*@
  modifies jour, mois, annee;
  requires j >= Jour.MIN  && j <= Jour.MAX;
  requires m >= Mois.MIN  && m <= Mois.MAX;
  requires a >= Annee.MIN && a <= Annee.MAX;
  ensures jour == j && annee == a && mois == m;
*/
public void setDate(byte j, byte m, byte a) throws DateException{...}
```

Fig. 3. Fragment of class `Date`

(see Figure 2) belongs to this class and (as suggested by its name and the informal documentation) it is supposed to decide whether the decimal represented by `this` is greater or equal than the decimal represented by parameter `d`. This behaviour is specified in the method specification.

However, after running ESC/Java on this asserted method, a warning is issued, suggesting that the postcondition might not hold. Inspection of the code reveals a *"copy paste"* error in the fourth `if` statement, where the same condition is tested twice, on both side of an `||` (or) operator. Replacing this condition by `decPart >= d.getDecPart()` would solve the problem, although it would probably be better to rewrite the method so that it simply tests the condition as expressed in the postcondition.

Final modifiers The class `Annee` represents a *year*. It declares two static variables called `MIN` and `MAX`, which represent the minimum and maximum year allowed by the application. Its declarations are as follows:

```
public static byte MIN = (byte)99;
public static byte MAX = (byte)127;
```

The class `Annee` also defines a method `check`, which is used to determine whether a value is between `MIN` and `MAX`. The class `Date` has three instance variables, representing the components of a date: `jour` (day), `mois` (month), and `annee` (year). The method `setDate` (see Figure 3) in this class assigns its arguments to these instance variables, provided they are in a valid interval (see the `requires` pragma). Surprisingly, ESC/Java complains when it finds a statement such as `date.setDate((byte)1, (byte)1, (byte)110);` (where date is a instance of class `Date`). The warning message states that the the third precondition of this call might not hold, even though 110 is between 99 and 127.

The problem is caused by the erroneous declaration of the variables `MIN` and `MAX` in class `Annee`. Because these variables are not declared `final`[2], their values can be changed at runtime by a direct assignment (as they are declared public).

[2] According to the Java semantics, final variables may only be modified when they are initialised, and afterwards they remain constant.

Thus, because the values of MIN and MAX might have changed, ESC/Java warns that possibly the precondition might not be satisfied.

Class invariants Typically, invariants are used to restrict the state space of a class, *i.e.* the set of allowed values for its instance variables. The most common example is an invariant which states that a reference may never be a null pointer, *e.g.* the variable `Purse purse`, as declared in the class `PurseApplet` should never be null.

```
//@ invariant purse != null;
```

Another common example of an invariant is to restrict the possible values of a numeric variable to a certain range. As remarked above, in the class `Decimal` it says that the value of the decimal fraction must be between 0 and 999. Inspection of the code reveals that the integer part of the decimal number is supposed to be a positive short, and combining this gives the following class invariant[3]

```
/*@
  invariant decPart >= 0 && decPart < PRECISION ;
  invariant intPart >= 0 && intPart <= MAX_DECIMAL_NUMBER;
*/
```

In Java(Card) there are no enumeration types, and to simulate them typically several constants with suggestive namesare defined, and variables are silently assumed to contain always one of these values. This implicit assumption can be made explicit by specifying invariants. For example, the class `Transaction` contains the following declarations.

```
/* the transaction status*/
public static final byte INDETERMINE = (byte)0;

/* the transaction status*/
public static final byte TYPE_CREDIT = (byte)50;
/* the transaction status*/
public static final byte TYPE_DEBIT = (byte)51;

/* the transaction type: debit or credit*/
/*@ spec_public*/ private byte type;
```

This suggests that the variable `type` always should have a value `TYPE_CREDIT` or `TYPE_DEBIT`. However, in the code (in the method `reset()`), an assignment `type = INDETERMINE;` occurs, suggesting that this is also a correct value for `type`. Having a specification which states the allowed values for this variable avoids all confusion[4].

[3] `MAX_DECIMAL_NUMBER` is equivalent to the maximal value of a short and the clause `intPart <= MAX_DECIMAL_NUMBER` of the invariant will thus be ensured by the type of the variable. We chose to state this explicitly for clarity of specification.

[4] However, notice that it does not prevent that other – arbitrary – variables can have one of these values, thus this does not completely simulate enumeration types.

```
/*@ invariant type == INDETERMINE ||
              type == TYPE_CREDIT ||
              type == TYPE_DEBIT;
*/
```

Invariants of this kind occur often in the specification of the electronic purse. It is easy to specify them, and useful as well, as there are examples in the electronic purse where such implicit invariants are violated. For example, the class `AccessCondition` declares constants to state the different access conditions for the actions in the purse. Following [1], variables that denote access conditions should be restricted as follows.

```
/*@ invariant condition == FREE ||
              condition == LOCKED ||
              condition == SECRET_CODE ||
              condition == SECURE_MESSAGING ||
              condition == (SECRET_CODE | SECURE_MESSAGING);
*/
```

However, in the constructor of this class, the variable `condition` is set to 0, which breaks this invariant. Correcting this and maintaining the invariant also allows to improve other parts of the implementation in this class. For example, in the method `verify()`, the following statement occurs:

```
switch(condition) {
  case FREE: ...
  case SECRET_CODE: ...
  case SECURE_MESSAGING: ...
  case SECRET_CODE | SECURE_MESSAGING: ...
  case LOCKED: ...
  default: //@ assert false;
           t = AccessConditionException.CONDITION_COURANTE_INVALIDE;
           AccessConditionException.throwIt(t);
  }
```

Because of the invariant we know that the default case will never be reached (as assured by the `//@ assert false;` annotation, which states that false should hold, every time this program point is reached), and thus that the exception never will be thrown. Thus, the default case can be removed from the code.

Similar cases occur frequently with `try-catch` statements. An operation is executed within a `try`, but as the class invariants assure that the operation never will throw an exception, the `catch` clause will never be executed. In the specification, we have annotated these cases with `//@ assert false;`. We think that the removal of this "dead code" can improve the readability of the class and, importantly for smart cards, it can reduce the size of the byte code.

Miscellaneous aspects of the specification There are many other aspects of the specification that are worth mentioning. Here we mention some.

- As explained above, in the class `Decimal`, two shorts are maintained denoting the integer and the decimal part (`intPart` and `decPart`, respectively) of a decimal number. The integer part ranges between 0 and `MAX_DECIMAL_NUMBER` (which is 32767, the maximal value for shorts). It is left unspecified whether numbers such as `MAX_DECIMAL_NUMBER`.999 are allowed. However, a method `round()` is defined, which according to the documentation returns a decimal number with `decPart` set to 0 and `intPart` set to the closest integer value. An obvious specification of this method reads as follows:

```
/*@
  modifies intPart, decPart;
  ensures decPart == 0;
  ensures intPart == (\old(decPart) >= (PRECISION/2) ?
                         (short)(\old(intPart) + 1) :
                            (short)(\old(intPart)))
*/
public Decimal round(){ ... }
```

  But, as pointed out by ESC/Java, an implementation of this specification breaks the class invariant `intPart >= 0`. The counterexample that is produced has `intPart` set to `MAX_DECIMAL_NUMBER` and `decPart` *e.g.* to 999. Possible solutions are to specify explicitly the outcome of `round()` in the case that `intPart == MAX_DECIMAL_NUMBER`, or to restrict the set of valid decimal numbers by further strengthening of the class invariant through addition of the following clause:

```
//@ invariant intPart == MAX_DECIMAL_NUMBER ==> decPart == 0;
```

  We chose this last solution.
- Among the developers of the electronic purse application there apparently have been different ideas about the implementation of the class `Decimal`. The implementation of several `setValue(...)` methods reveal that `intPart` is assumed to be greater or equal than 0, but on the other hand there are methods `isNegatif()` and `isPositif()`, which test whether a decimal value is negative or positive, respectively. As we specify that `intPart` should be greater or equal than 0[5] these methods become obsolete. We can show this by specifying that their results can be predicted, *e.g.* `isNegatif()` we specify as follows:

```
/*@
  ensures \result == false;
*/
public boolean isNegatif(){ ... }
```

- Two classes, `TransactionRecord` and `ExchangeRecord` implement a cyclic table (of `Transactions` and `ExchangeSessions` (storing currency changes),

[5] *cf.* our email exchange with H. Martin, Gemplus.

respectively). These implementations are clearly copied from each other, but this is nowhere documented. Also the fact that a cyclic data structure is implemented is not very well documented. Class `TransactionRecord` contains a single remark that it is implemented as a cyclic table, and for class `ExchangeRecord` this is only stated in the documentation of a private method. Also, no specification of the operations on the cyclic data structure are given.

As a result, in class `ExchangeRecord`, part of the code that is crucial for its cyclicity has been commented out by other developers of the electronic purse. Having a formal specification would probably have been helpful to explain the complexity of the implementation to the other developers, and the "wrong correction" would have been signaled earlier[6].

Finally, when writing the formal specifications of the cyclic tables, we found an error in the implementation. When a delete operation is called for an element that is not in the range of the table, the operation nevertheless will be executed and as a side-effect it will corrupt the table by erroneously moving its first element outside the range of the table.

5 On the use of ESC/Java

We find ESC/Java a useful tool, which is pleasant to work with, but nevertheless we have some suggestions for improvements, both for the specification language and for the verifier.

Concerning the specification language, we feel that certain specification constructs that are available in JML [10], should be provided in ESC/Java as well. They could be implemented as syntactic sugar only, but would be useful to give clear and concise specifications. First of all, we would like to be able to use method names in specifications, as is allowed in JML for so-called *pure* methods, *i.e.* methods without side-effects. Further, we would like to have some extra specification constructs for modifies clauses, *e.g.* `\fields_of`, to denote all the fields of an object, and `\nothing`. Also we feel that having some extra quantifiers, such as `\min`, `\max`, and `\choose` could be useful to increase expressiveness of the specification language.

Figure 4 shows as an example how we would like the specification of the method `clone()` in class `ExchangeSession` to be (see [2] for the ESC/Java specification as it is). We only specify that all the fields of the current class may be modified, without explicitly mentioning them. As the fields of the component classes `date` and `heure` may be modified as well, we mention this explicitly. Similarly, we mention explicitly that the elements in the arrays `id`, `terminalTC` and `terminalSN` may be modified[7]. Further, instead of having postconditions

[6] Of course, having a general implementation of a cyclic table and instantiating this for the different kinds of data would have been even more elegant.

[7] In JML one can write this modifies clause also simply as `\fields_of`(`\reach`(this)), which would probably be useful in many cases as well, but has a more complex semantics.

```
/*@
  modifies \fields_of(this), \fields_of(date), \fields_of(heure),
           id[*], terminalTC[*], terminalSN[*];
  requires es != null ;
  requires es.id != terminalTC & es.id != terminalSN &
           es.terminalTC != terminalSN;
  ensures this.equal(es);
  exsures (TransactionException e)
              e._reason == TransactionException.BUFFER_FULL
              && JCSystem._transactionDepth == 1;
  exsures (NullPointerException) false;
  exsures (ArrayIndexOutOfBoundsException) false;
*/
void clone(ExchangeSession es) { ...
}
```

Fig. 4. Specification of `clone` in `ExchangeSession` in ideal ESC/Java

stating that all the fields are ensured to be equal to the corresponding fields of `es`, this is denoted by writing `this.equal(es)`, where `equal` is overwritten appropriately in `ExchangeSession`.

Also, it would be nice if one could specify runtime exceptions in `exsures` clauses, while they are not mentioned in the `throws` clause of the method.

With respect to the verification that is done by ESC/Java, we found that it is unfortunate that ESC/Java does not try to check the modifies clauses, because an incorrect modifies clause can influence the acceptance of other specifications. Suppose one has the following (annotated) methods:

```
/*@ modifies x;
    ensures x == 3;
*/
void m() { x = 3;
           n (); }

void n() { x = 4; }
```

Remember that a method without a modifies clause is assumed to modify only freshly allocated memory, if any [11]. The specification for method `m()` is thus accepted by ESC/Java, although it is incorrect. When annotating existing programs, as we did, it is easy to forget to mention that a variable may be modified, and we felt the need to overcome this problem.

Therefore, we have implemented a static checker for modifies clauses. In the tradition of ESC/Java, this checker is designed to be efficient, but it is neither sound, nor complete. It does a syntactic analysis of the annotated program to recognise the various assignment statements and then checks whether the variables that are the "destination" of an assignment are mentioned in the appropriate modifies clauses. This checker will be described in more detail in a separate paper.

Finally, we feel that it would be an improvement if the ESC/Java theorem prover would deal more precisely with arithmetic operations (also on bytes and shorts). For example, several times we needed for *e.g.* a byte `b` with a value between 0 and 10, but after the operation `b++` this variable still has a positive value.

6 Conclusions

We have presented a case study in formal specification of smart card programs, using ESC/Java. We have taken an electronic purse application and annotated it with a functional specification, describing its behaviour, basing ourselves on the informal documentation of the application. We have checked the implementation *w.r.t.* the specification, using ESC/Java, thereby revealing several errors in the implementation. Using ESC/Java we were also able to find that some parts of the program will never be reached, thus allowing reduction of the code size – which is important for smart card applications.

The errors that we found in general are not very complex, and they could have been found by careful code-inspection or testing. But, writing the formal specification forces one to do code-inspection, and having the theorem prover ensures that all possible cases will be considered when checking the specification.

The specifications that we have constructed for the electronic purse application are not very complex, but describe the functional behaviour of methods as precisely as possible. Nevertheless, we found errors in the code, and we would like to emphasise that even simple formal verification can help significantly to increase confidence in a program. In particular, explicitly specifying class invariants turns out to be very useful.

Future work We plan to work in the field of specification languages for Java: how to improve them, and how to develop appropriate verification techniques for them. In particular, we will focus on the following points.

- We plan to develop a full smart card application from scratch, with annotations. We are interested whether this will affect the quality of the specification and/or the program. Also we would like to know how easy it is to construct the specifications at the same time as developing the code. Is it useful, because one can get immediate feedback on an implementation or, is it a burden because more time is spend on keeping the the specification up-to-date than on implementing the program?
- Future versions of JavaCard will probably allow multi-threading. Therefore we plan to study how ESC/Java (and JML) can be used to specify (and check) concurrent programs.

– Related with this is an extension of JML with temporal logic. Currently we are studying how to integrate temporal logic in the specification language, future work will be to study appropriate verification techniques.
– Most loop structures that are used in typical smart card programs are very restricted and it is relatively easy to show their termination. We plan to develop an automated verification technique for termination of loops in the tradition of ESC/Java, covering the most common cases.
– We skipped the cryptographic aspects of the application at hand. It is future work to see whether ESC/Java (or JML) is useful to specify such algorithms more precisely, and to develop appropriate (automated) verification techniques.

References

1. E. Bretagne, A. El Marouani, P.Girard, and J.-L. Lanet. Pacap purse and loyalty specification. Technical Report V 0.4, Gemplus, 2000.
2. N. Cataño Collazos and M. Huisman. Annotated files Electronic Purse case study, 2001. `http://www-sop.inria.fr/lemme/verificard/electronic_purse`.
3. Differences between Esc/Java and JML, 2000. Comes with JML distribution, in file `esc-jml-diffs.txt`.
4. Extended static checking for Java. `http://research.compaq.com/SRC/esc/`.
5. C. Flanagan and K.R.M. Leino. Houdini, an annotation assistant for ESC/Java. In J.N. Oliveira and P. Zave, editors, *Formal Methods Europe 2001 (FME'01): Formal Methods for Increasing Software Productivity*, number 2021 in LNCS, pages 500–517. Springer, 2001.
6. Gemplus. Applet Benchmark Kit. `http://www.gemplus.com/smart/r_d/publications/case-study/`.
7. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. *Software Tools for Technology Transfer (STTT)*, 3/3:332 – 352, 2001.
8. The JASS project. `http://semantik.informatik.uni-oldenburg.de/~jass/`.
9. JavaCard Technology. `http://java.sun.com/products/javacard`
10. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06, Iowa State University, Department of Computer Science, 1998.
11. K.R.M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical Report SRC 2000-002, Compaq System Research Center, 2000.
12. The LOOP project. `http://www.cs.kun.nl/~bart/LOOP/`.
13. H. Meijer and E. Poll. Towards a full formal specification of the Java Card API. In I. Attali and Th. Jensen, editors, *Smart Card Programming and Security*, number 2140 in LNCS, pages 165–178. Springer, 2001.
14. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2^{nd} rev. edition, 1997.
15. J. Meyer and A. Poetzsch-Heffter. An architecture of interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 63–77. Springer, 2000.
16. Sun Microsystems. The Java Card 2.1.1. Application Programming Interface (API), 2000.