An Integrated Development Environment for Java Card [1]

Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio and Henrik Nilsson [2]

INRIA – CNRS – I3S – UNSA

INRIA Sophia Antipolis, BP 93, FR-06902 Sophia Antipolis, France

First.Last@sophia.inria.fr

Abstract

This article describes a Java Card programming environment which to a large extent is generated from formal specifications of the syntax and semantics of Java Card, the JCRE (Java Card Runtime Environment), and the Java Card APIs. The resulting environment consists of a set of tightly integrated and somewhat smart tools, such as a Java specific structure editor and a simulator which allows an application to be tested before being downloaded to a card. Furthermore, the simulator analyses the applet in question in order to find out the structure of the accepted commands. This information is then used to automatically adapt the GUI of the simulator.

Key words: Java Card, development environment, formal specification, simulation

1 Introduction

This paper describes a programming environment for Java Card [3] [8,19] applications which is being developed within the OASIS project [4] at INRIA Sophia-Antipolis.

[1] This work was partially financed by Bull.

[2] Henrik Nilsson was supported by a post doctoral grant from the Wenner-Gren Foundations, Stockholm, Sweden.

[3] Java and Java Card are trademarks of Sun Microsystems Inc. All other trademarks mentioned are proprietary of their respective owners.

[4] `http://www-sop.inria.fr/oasis`

The Java Card environment is developed within Centaur [5], a generic, interactive programming environment generator. It makes it possible to create programming tools such as structure editors, compilers, interpreters based on formal specifications of the syntax and semantics of the language in question. These are then integrated into a graphical programming environment which also can provide viewers for important structures such as stacks, heaps, and objects during interpretation. Centaur thus allows sophisticated tools to be developed quickly at a high level of abstraction. Furthermore, having access to the formal semantics provides the potential for proving interesting properties about applications within the same framework, and since the semantics is executable it can be tested and debugged which is important if it is to be used as the basis for proofs. Proving properties is of key importance in this context since security is often a prime concern in applications involving smart cards. However, this article focuses on the basic aspects of our programming environment, such as editing, testing, and debugging.

The rest of this article is organised as follows. In the next section, we discuss related work. Then we present the architecture of our environment and explain how it is generated from various formal specifications. The following sections each discuss one major part of the Java Card environment in greater detail, illustrating through a running example (see appendix A for the complete source code). Finally, we present conclusions.

2 Related work

The central aspect of the programming environment presented in this article is that it to a large extent is generated from formal specifications of the syntax and semantics of Java Card. This puts it apart from most other Java Card environments. From a formalisation perspective, the emphasis on programming environments means that our design choices are a bit different from much of the related work in that area. In this section, we will first look into the programming environment aspects, and then continue with the formalisation aspects.

Sun has developed free tools around Java Card: a converter (available since November 1999) and a simulator. The latter allows Java Card applet code to be simulated, but unfortunately it has no debugging facilities such as breakpoints, step by step execution, or variable value inspection. Thus it works as a black box taking a file of command APDUs as input (byte sequences in textual format) and giving back a file of response APDUs as output. Moreover, all the tests cannot be performed since it is not possible to simulate card withdrawal or power failure.

There are also commercial Java Card development kits, for instance *Odyssey lab*^{TM} from Bull, *Cyberflex*^{TM} from Schlumberger, *GemXpresso RAD*^{TM} from Gemplus, *Sm@rtCafé Professional*^{TM} from Gieseke & Devrient, *GalatIC*^{TM} from Oberthur Card Systems. These kits contain a card reader, cards, and software tools (a converter, a loader, a tool to test applets on the card, and sometimes a simulator). According to [6], the programming environment providing the best coding support is GemXpresso$^{\text{TM}}$. It offers a wizard to help developers in creating Java Card applications which takes care of low-level issues such as handling APDUs. For testing and debugging, Sm@rtCafé$^{\text{TM}}$, currently being the the only environment offering a Java Card specific simulator-debugger, has the edge. An ideal Java Card specific simulator-debugger should be able to give information about the stack and the storage, to handle atomic operations when card withdrawals or power failures are simulated and to inspect the JCRE contents.

Our aim is obviously not to compete with these smart card manufacturers, but to propose new methods for creating development environments with innovative functionality. For example, most of the currently commercially available Java Card environments are really Java environments augmented with some extra tools. In particular, a standard Java compiler is used to compile Java source code into byte code. Thus a developer would not know for sure whether the application being developed conforms to the Java Card subset until an attempt is made to convert the *compiled* code into a CAP file. In contrast, our editor makes it impossibleto write code which does not conform to the Java Card subset.

We have put a lot of effort into making it easy to simulate and test the behaviour of Java cards at the APDU level. While communication at the APDU level and the associated code to interpret APDU commands and dispatch on them is the current standard for writing Java Card applets, and will continue to be the standard for the foreseeable future in legacy contexts, it has been proposed that higher-level protocols would be more appropriate for new developments. For example, Vandewalle and Vétillard [21] propose a design framework where a card application is viewed as a remote object. Its methods are invoked through a proxy object executing on the CAD. Behind the scenes, a special protocol called DMI (Direct Method Invocation), built on top of the APDU standard, handles the communication (this is all similar to how Java's RMI works). This framework has been implemented in the GemXpresso RAD environment by Gemplus. It is used by their wizard to help applet developments. A recent work of Hagimont and Vandewalle [13] uses the DMI facility to propose a control of access rights between remote applications based on software capabilities.

Since DMI is still based on APDUs, this and similar protocols could be incorporated directly on top of what we have. This might have advantages since

the simulation would be sufficiently detailed to make it possible to simulate events such as card withdrawal or power failure. On the other hand, since the aim of such protocols is to hide the low-level communication details, it might make more sense to simulate such protocols directly. It would not be difficult to adapt our environment in this way, but one should probably still keep the simulation capability at the APDU level as well for those who needs or prefers this.

Concerning the formal semantics of Java and Java Card, there is already a large body of work in the area. In the following, we will just consider some representative examples.

Drossopoulou and Eisenbach [11] have described the semantics of Java_S, a large, sequential, subset of Java, with the main goal of proving type soundness. The semantics is described in small-step style, which should facilitate an extension to cover concurrent aspects. Syme then formalised most of this semantics in his theorem proving system Declare [20]. An executable ML program can be extracted which makes it possible to test and debug the specification. Since the focus was on type safety, Java_S only includes the features of Java which are essential for that purpose. Aspects which were left out include constructors, class variables, local variables (and concurrency).

Another example is the work by the Bali team at Munich; see Oheimb and Nipkow [18] for example. Among other things, they formalised the type system and the operational semantics for a significant subset of sequential Java, reasonably similar to Java_S. Again, a major goal was to prove type soundness. Unlike the work by Drossopoulou and Eisenbach, a big-step semantics was used.

In contrast to the formalisations discussed above, we strive to cover as much of Java and Java Card as possible (including concurrency [5]), since we want to use the formal semantics as a component of a programming environment. This also means that we have formalised parts of the JCRE and the Java Card APIs. Furthermore, like Syme, we also emphasise the importance of having an executable specification for testing and debugging purposes. On the other hand, we have not addressed the static aspects of Java or Java Card. That is not to say that this is not an important aspect of a programming environment, but thus far we have chosen to prioritise the dynamic aspects.

There is also a body of work focusing on the J(C)VM; that is, to study the Java and Java Card semantics at the byte code level. Examples include Lanet and Requet [17] (a formalisation of most of the JCVM in B), Cohen [9] (formalisation of the so called Defensive JVM in Common Lisp), and more recently Barthe *et al.* [4] (formalisation of the complete JCVM in Coq). These works

[5] Java Card may well become a concurrent language too in the future.

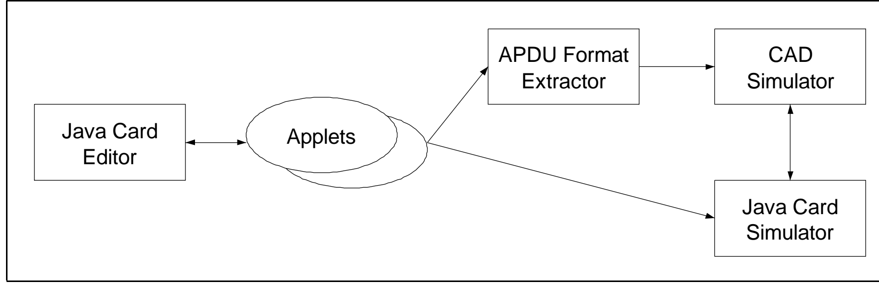

Fig. 1. Programming environment architecture.

all aim to be complete or at least fairly complete, and, like the work presented in this paper, the formalisations of Cohen and Barthe *et al.* are executable. In contrast, we chose to work at the source level since we wanted to describe the semantics of Java and Java Card as such, and since we did not want to rely on external Java tools to provide an interpretative capability in our programming environment.

3 Overview

3.1 The Environment

Our Java Card environment is made up of two parts: a structure editor and a simulator for testing applets. The simulator in turn consists of three parts: the APDU (Application Protocol Data Unit) format extractor, the CAD (Card Access Device) simulator, and the Java Card simulator. Figure 1 shows the architecture of the environment. Both parts of the environment understand normal text files, so it is easy to import existing code, to replace the editor with some standard editor like Emacs, or to use the structure editor on its own. Compilation to byte code and functionality related to loading applets onto physical cards is outside of the scope of this work, but existing free or proprietary tools could easily be interfaced from the environment.

Figure 2 gives an overview of the interfaces of the different parts of the environment. The top left corner shows the structure editor window. In the lower half of the figure, one can see the window for constructing and sending generic APDUs (`New APDU`), the menu for selecting applet-specific commands (`Extracted Commands`), and a window for constructing and sending an APDU for a selected, applet-specific command (`Purse PURSE_DEPOSIT`). The top right corner shows the APDU log and a window showing a failed regression test (see section 5). The command and APDU log windows all belong to the CAD simulator. Finally, in the center: the window for setting interpretation parameters belonging to the Java Card simulator.

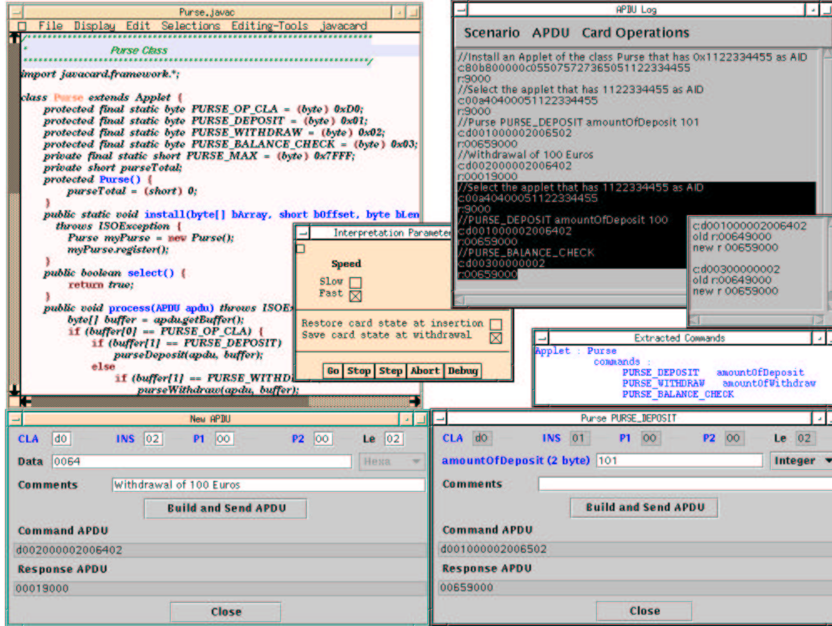

Fig. 2. Overview of the different tools comprising the environment.

The editor is Java Card specific. Thus the developer is assured that the written code is free from syntax errors and respects the Java Card subset of Java. Moreover, the editor facilitates development of Java Card code by displaying context-sensitive menus containing all valid syntactic constructs for any selected part of the source code. This feature is useful for people beginning to write Java Card applications, for example smart card developers used to writing assembler code. But it can also be useful for experienced programmers, who could, for example, enter the skeleton of a complete `if`-statement with a single selection from a menu.

An important aspect of the environment is that it enables the developer to test and debug an applet on a work station. It is not necessary to download it to a real Java Card, or having access to a card reader. This is achieved by providing simulators for the Java Card and the CAD. This should in itself cut down the time for the test-debug-change cycle significantly: if an applet runs well on the simulator, it will run well on the Java Card. However, heterogeneous Java Card platforms run different virtual machines, each of which has specific features in terms of optimisation, allocation, or external library. We aim at providing a high-level view of the applet behavior such that applet providers can claim that their applets verify important security properties that are valid for every platform.

In addition, our environment makes it easy to study the communication between the applet and the CAD in terms of sent and received APDUs, as well as the inner workings of the applet itself. The simulation environment also provides a facility to extract APDU formats from applet source code.

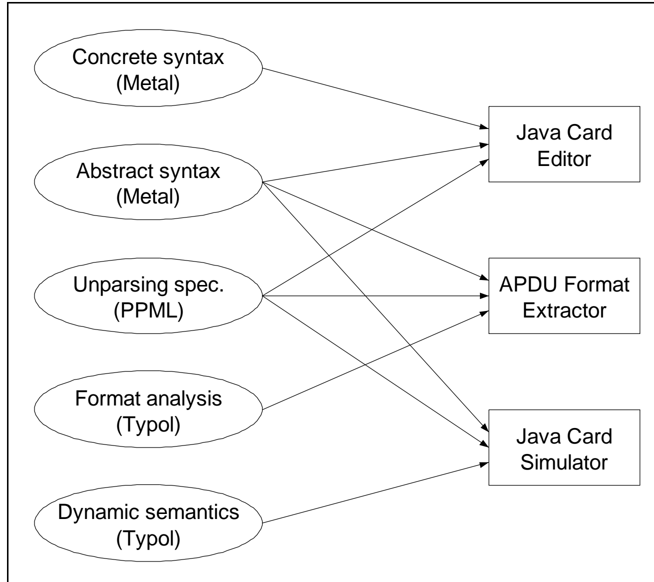

Fig. 3. The relation between specifications and generated tools.

3.2 Tool Generation

All the above mentioned tools were created using the Centaur system. Figure 3 illustrates the relationships between the developed specifications on the one hand, and the generated tools on the other.

To obtain the structure editor, we have specified a Java Card parser and a pretty printer or unparser. The parser specification defines the concrete and abstract syntax of the Java Card language as well as tree building functions. It is written using the Metal formalism which permits the concrete and abstract syntax to be specified simultaneously. This specification is then used to automatically generate a parser. The parser is responsible for translating Java Card source (text) into an abstract syntax tree which is how Java Card applets are represented inside the environment. The PPML specification is then used to translate an abstract syntax tree back to text, e.g. for display purposes within the editor, or for storing the Java Card program in a file.

The central part of the Java Card simulator is the dynamic semantics of Java Card, specified in Natural Semantics using Typol [16]. Typol is the language used to specify semantics in Centaur. It is a typed, declarative, logic language with backtracking, unification and pattern matching mechanisms (see section 6 for an example). The Java Card semantics is specified on the level of the abstract syntax, and thus it also needs to refer to the Java Card abstract syntax specification. For studying the behaviour of an applet during execution, it is possible to view important runtime structures. These structures are represented internally as trees. Thus PPML is used to specify the unparser.

The APDU format extractor also works on the abstract syntax tree of a Java Card applet. In addition, it uses data structures specific to the analysis task, which are again specified using Metal and PPML. The extractor itself is written in Typol.

4 The Java Card Structure Editor

This section describes the features of the editor. We illustrate by explaining how to write a small Java Card program, which also will be used as Ariadne's thread in the following sections.

Our editor is dedicated to the Java Card language. It prevents the developer from making errors w.r.t. the general Java syntax (e.g. a semicolon or a brace missing), but also from making errors related to the specificities of Java Card (e.g. using the `char`, `long`, `double`, `float` primitive types, multidimensional arrays, or the `synchronized`, `volatile`, `transient` keywords) [10]. These latter errors would otherwise only be discovered when converting class files into CAP (Converted Applet Program) files since standard Java compilers are used to compile Java Card programs. Anyone who uses our editor will only encounter errors related to the static semantics (e.g. undefined variables, type errors) when compiling, and no errors at all during the conversion stage.

Applets are stored as plain Java Card text files. This makes it easy to import existing Java Card applications into our environment, or to use traditional text editors for development should that be desired.

Internally, the applet being edited is represented as an abstract syntax tree. Syntax colouring is made possible by specifying different styles (font, size, colour) in the unparsing rules. Note that, unlike a conventional text editor like Emacs which does not do a proper parsing, we are sure that the colours will always be correct. Moreover the formatting scheme (i.e. indentations, where the braces are set) is the same for all the Java Card programs ensuring consistency regardless of author.

The editor provides two ways for writing source code: free text editing or syntax-directed editing. In the first mode, the editor behaves like a normal text editor, but the written part is parsed when this mode is exited. Thus, only syntactically correct program fragments can be inserted into the abstract syntax tree. The other mode facilitates the development by displaying the valid syntactic constructs which can replace the selected part of the code in a context-sensitive menu. When any of these constructs is selected, its structure replaces the selected part and only the placeholders need to be filled in by the developer (a placeholder starts with a dollar sign). If the placeholder is for

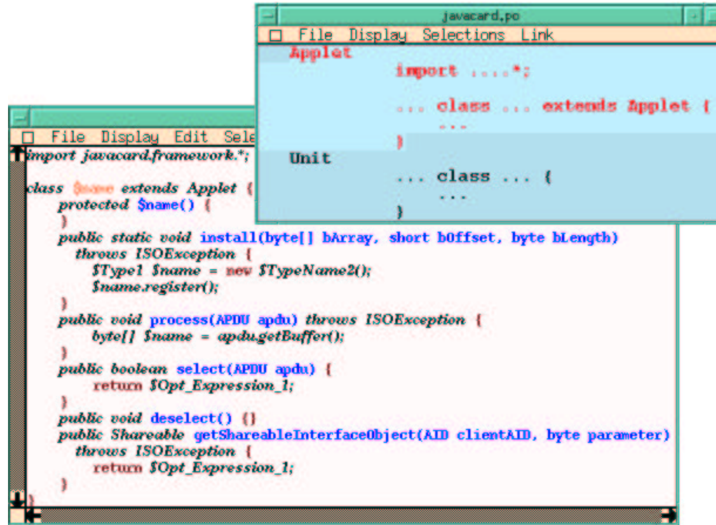

Fig. 4. Start menu and syntax-coloured applet skeleton.

an identifier, the completion mechanism can be used to avoid writing the full name of an identifier which occurs somewhere else in the program.

When starting writing a Java Card program from scratch, a menu allows either an applet pattern or a class pattern to be inserted into the empty editor window; see figure 4. If the developer chooses the applet pattern in the menu, the skeleton of an applet is displayed. This skeleton contains the structures of the applet constructor and the main methods i.e. `process`, `install`, `select`, `deselect` and `getShareableInterfaceObject`. The `install` method already contains the creation of the applet object and the registration call to the JCRE. Thus, the developer only needs to fill in the placeholders.

Figure 5 shows a menu of the available block statements and the result of selecting and inserting the `if`-pattern into an applet being developed. This is a simple purse applet, which we are using as the running example in the rest of the article. The complete program can be found in appendix A. The services provided by this applet are deposits, withdrawals, and balance inquiries. See Chen [7] for an introduction to how to write Java Card applets. Finally, also note that comments are handled. They are seen as decorations in the abstract syntax tree and correctly handled by the pretty printer (in green on lavender background).

5 The CAD Simulator

The CAD simulator gives the user the possibility to interactively test applets. It provides a graphical user interface which allows both generic and application-specific APDUs to be constructed and sent to the Java Card sim-

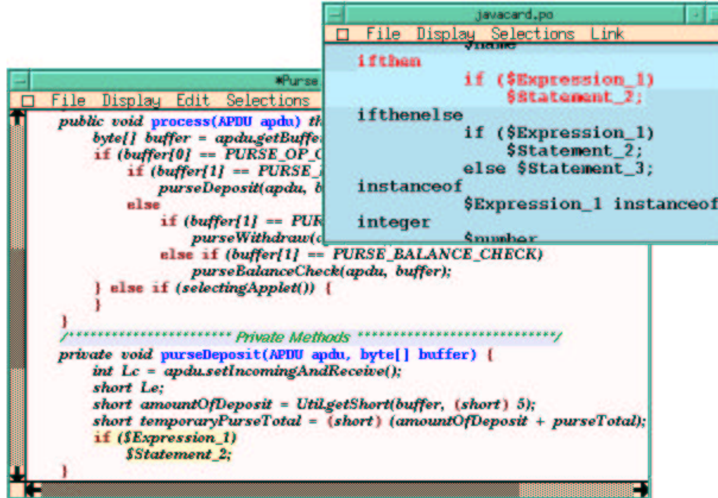

Fig. 5. Block statement menu.

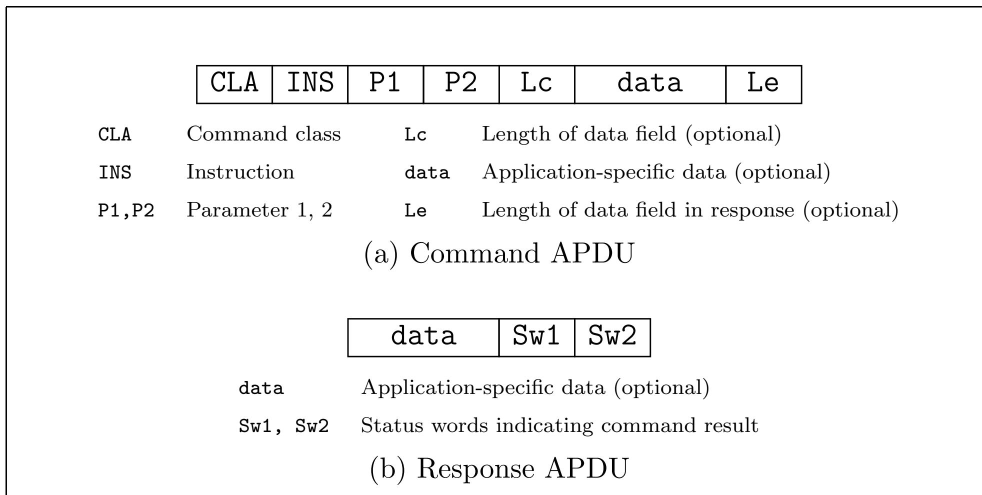

Fig. 6. The general layout of APDUs.

ulator. It also allows the returned response APDUs to be inspected. An APDU log records all sent and received APDUs. The CAD simulator allows these sessions to be stored and later re-used in order to undertake regression testing.

While the basic APDU layout and some commands are standardised (figure 6), any particular application necessitates the definition of application-specific commands. Since an APDU is just a byte sequence, it would in principle be enough if the CAD simulator simply allowed arbitrary byte sequences to be assembled and sent. But that would clearly be rather inconvenient for the user under normal circumstances. It is far easier to just specify the contents of an APDU and let the simulation environment take care of the packing (and unpacking) details.

However, this means that the CAD simulator needs to be told about the details of application specific APDUs, such as what command names there are,

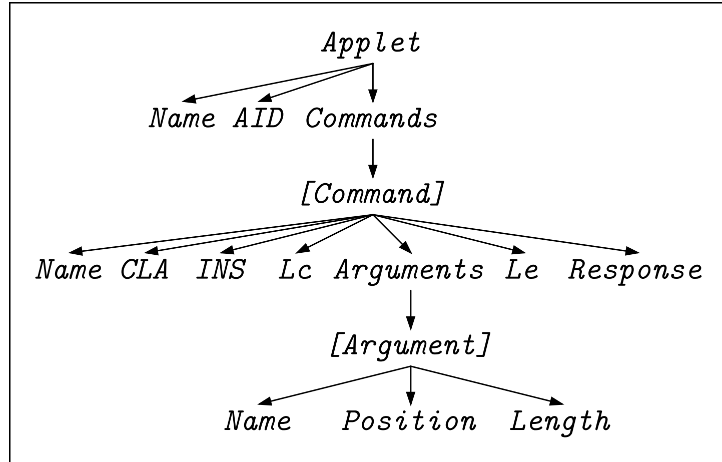

Fig. 7. The structure of an applet description. Square brackets indicate a list.

the command number for each one, the names and types of any arguments (in the data field), etc. To make it possible to use the CAD simulator without first having to provide a separate specification of the application specific APDU formats, we have developed a tool, the *APDU format extractor*, which extracts this information from an applet by static analysis. The extraction scheme works reasonably well since a Java Card applet usually is written in a very stylized way (there is always a method `process` which does the initial command decoding, the basic layout of the APDUs is standardised, etc.). Thus a developer typically just has to write the applet code, or indeed, get code from somewhere else, and the CAD interface will automatically adapt itself to the applet (or applets) in question. The results from an analysis can be saved to avoid having to re-analyse known applets.

Figure 7 shows a somewhat simplified structure of the applet descriptions returned by the format extractor. For the purpose of the CAD simulator, an applet is described by its name and AID (applet ID, for selecting the applet), and descriptions of the accepted commands and their associated formats. The format extractor works by symbolically executing the applet code, keeping track of data dependences and constant and variable names in the process.

The APDU extraction process consists of four steps :

(1) Statement filtering
(2) Command identification
(3) Command APDU arguments analysis
(4) Response APDU analysis

The APDU format extractor is written in Typol, using inference rules and axioms for each step of the extraction process.

Step 1 transforms the program in order to keep only the statements that

```
instructions |- call(Util.arrayCopy,SrcBuffer, SrcOff, DestBuffer, DestOff, Length)
              -> arrayAssign((SrcBuffer, SrcOff,Length ),
                         (DestBuffer,DestOff,Length)).instructions)
```

Fig. 8. Extraction of the `arrayCopy` statement.

```
Lookup(instructions |- bufferAccess is APDUIdent.getBuffer()[ISO.OFFSET_INS])
Intersection(|- INS, GetValue(Command)-> InsForThen)
UpdateCommandsList( AppletSpecification |- (CLA, InsForThen), GetName(Command)
                          ->  NewAppletSpecification)
Intersection(|- INS, Not(GetValue(Command)) -> InsForElse)
-----------------------------
(CLA, INS), instructions, APDUIdent
  |- if (bufferAccess == Command), AppletSpecification
   -> (CLA, InsForThen ), (CLA, InsForElse), NewAppletSpecification
```

Fig. 9. Analysis of the conditions concerning the second byte of the APDU buffer.

we are able to analyse and to find important method calls such as calls inside the current class and calls to system methods like `arrayCopy`. Figure 8 shows how this filtering is done in Typol for a call to arrayCopy: given a list of instructions, each statement is considered in sequence, to build a new list of instructions. Here the pattern-matching on the one hand focuses on invocation (`call` statement) and on the other hand imposes the name of the method (`Util.arrayCopy`). Parameters of the considered statement (`SrcBuffer, SrcOff, DestBuffer, DestOff, Length`) are re-used to build the instruction `arrayAssign` which is then stored in the resulting list. Array copying is important because the APDU buffer is an array, and array operations may thus concern the APDU.

Step 2 analyses the conditions of `if`-statements in order to find accesses to the first and second byte of the APDU buffer. These two bytes (designated as *`CLA`* and *`INS`*) are used to distinguish between different instructions. A list of *`CLA`* and *`INS`* bytes being tested for are extracted from the condition of each `if`-statement. The `if`-statements are then transformed into blocks labelled with the corresponding *`CLA`* and *`INS`* for further analysis. Moreover, this step creates the list of pairs (*`CLA`*, *`INS`*) which are valid for the given applet being tested for) and the list of pairs (*`CLA`*, *`INS`*) which are invalid (which cause a `CLA_NOT_SUPPORTED` or `INS_NOT_SUPPORTED` exception). Figure 9 gives one of the main rules of the second step which analyses the condition of an `if`-statement, updates the list of existing commands, and returns the corresponding *`CLA`* and *`INS`* sets. In this rule, `Lookup` is a function which performs a lookup in a set of statements and follows dependences graph to determine whether two expressions are equal (after evaluation) or not. `GetName` is a function that takes an expression and returns a string in order to give a name to the corresponding statement (for example when the expression is an identifier it returns the identifier name).

Step 3 examines accesses to the APDU buffer and updates the list of arguments a command APDU should contain. The fifth byte of the APDU is the length

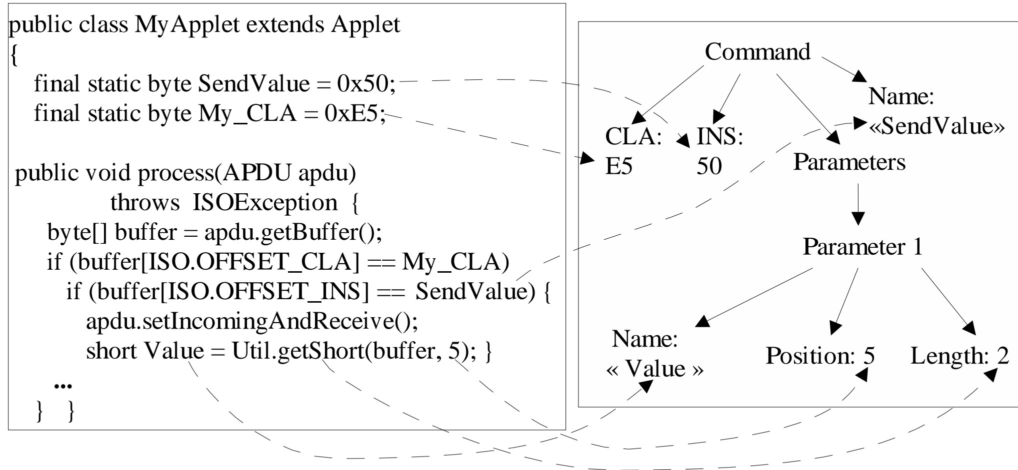

Fig. 10. Illustration of the extraction process.

of the data field which itself starts at the sixth byte. The APDU extractor considers the data field as a list of arguments of different lengths and positions, which are to be identified by the analysis. When an access to the APDU buffer is encountered, the position (given by the offset of a call to `arrayCopy`, `getShort`, or similar), the length (2 for a `getShort` method, for example), and the name to be associated with the thus identified argument can normally be determined. Then the list of arguments of the corresponding commands has to be updated. The currently analysed block (corresponding to the nearest `if`-statement) contains the list of commands concerned by the analysed buffer access.

Step 4 is similar to step 3 and determines the structure of the response APDU.

Figure 10 gives an example which schematically illustrates the principle behind the extraction process. See Henrio [14] for further details.

It could happen that the analysis carried out by the format extractor fails due to an applet not being sufficiently stylized. In that case, the user can create the applet description manually by using an editor.

The CAD simulator also allows standard applet installation and selection APDUs to be sent, as well as arbitrary APDUs assembled from scratch. The latter is useful for sending non-conforming APDUs in order to test that an applet handles invalid commands correctly, for instance. Furthermore, there is an option to simulate card withdrawal, which optionally can be combined with saving the state of the running card. The card simulator can be initialized from such a saved state.

When applied to the Purse example, the extractor determines that the name of the applet is `Purse`, and that this applet accepts three commands: `PURSE_DEPOSIT`, `PURSE_WITHDRAW`, and `PURSE_BALANCE_CHECK` with instruction

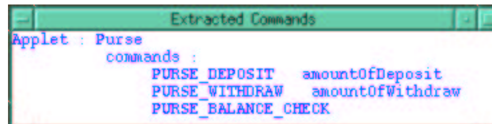

Fig. 11. The CAD simulator command selection window.

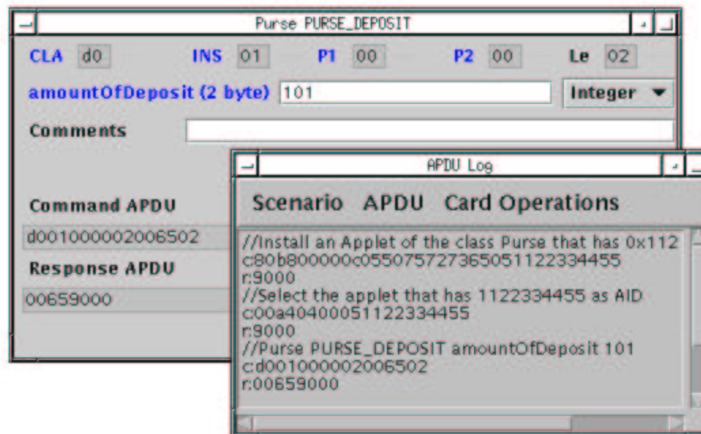

Fig. 12. The CAD simulator command send window and the APDU log.

numbers 1, 2, and 3 respectively. `PURSE_DEPOSIT` is found to have one short (2-byte) argument `amountOfDeposit` at position 5, and `PURSE_WITHDRAW` is similarly found to have one short argument `amountOfWithdraw` at position 5. Finally, the extractor determines that all commands return a short result `purseTotal`, i.e. the resulting balance.

The result of the analysis is sent to the CAD simulator which opens a window which allows one of the three identified commands to be sent to the applet; see figure 11. Once one of these commands has been selected (by clicking on it), a second window pops up which allows any parameters to be filled in prior to sending the complete APDU to the applet; see figure 12. Many of the numeric fields of this window (*CLA*, *INS*, etc.) are fixed as they are given by the command in question and only present for information purposes. The window also shows the full details of the sent command APDU and, eventually, the received response APDU in the form of byte strings. Figure 12 shows the result once a deposit of 101 Euros has been made to an empty purse. Note that the resulting balance is 101 (0x0065) Euros. The figure also shows the APDU log which keeps track of all sent and received APDUs and allows them to be examined.

6 The Java Card Simulator

The central part of the Java Card simulator is an executable specification of the dynamic semantics of the Java Card language [19]. It is expressed in Nat-

ural Semantics, small-step style, using Typol. Thus the semantic rules express the stepwise transformation of the abstract syntax tree of an applet into the final result. The semantic specification is derived from a Java specification also developed within our group [2,3]. Almost the entire Java semantics has been re-used without changes to make it easy to profit from further developments of it (or vice versa). As there is no concurrency on a Java card, the scheduler was changed [10]. There is only one active thread and a thread may only be activated when some special program points are reached. Furthermore, executable specifications of Java Card APIs (i.e. specifications of Java Card classes such as `AID`, `APDU`, `APDUException`, `Applet`, `JCSystem`) have been added. On the top, we have specified a minimal JCRE with the duties to receive command APDUs, select applets, dispatch commands APDUs between applets, and send response APDUs to the terminal.

The principle of Typol rules is the following: inference rules and axioms are structured in sets, that deal with a same object (`evaluateExpression` for instance). Each rule $\frac{P_1 \ P_2 \ \dots \ P_n}{P}$ has a number of premises $P_1, P_2, \dots, P_n$ and one conclusion P. If all premises are true, then the conclusion holds and the rule applies.

Figure 13 shows some Typol rules from our Java Card semantics. They are concerned with giving the semantics of an `if`-statement. The first rule says that an `if`-statement where the condition is not yet in normal form, is to be rewritten to an `if`-statement where the condition has been evaluated one step further by the rule `evaluateExpression`. The parameters `ObjL1` and `ClVarL1` (and variations) are the object list and class variable list, respectively, i.e. they represent the current global state of the Java machine. `OThId` is the id of the currently executing thread. The actual `if`-statement is embedded in what is referred to as a 'closure' (constructed with the operator `clr`), which represents the rest of the computation for a thread (in some global state). In particular, a closure contains a list of instructions, of which the first one is the current instruction. Other important parts of the closure are the object id (corresponding to `this` object), and an environment for local variable bindings.

The two following rules state that an `if`-statement where the condition is in normal form (i.e. `true` or `false`), should be replaced by either the `then`-branch or the `else`-branch depending on the condition.

It should be pointed out that the current semantics is incomplete in some respects. For example, it is a dynamic semantics which currently does not perform static type checking. As a consequence, overloading resolution is not handled completely accurately. However, since Java Card applets tend to be small and fairly simple, the current limitations have as yet not caused any major problems.

```
If_EvaluateExpression:
   evaluateExpression(ObjL1, ClVarL1, Env1, OThId, ObjId1, Mode
            |- Expr1 -> ObjL1_1, ClVarL1_1, Env1_1, Expr1_1, InstL2, ThStatus)  &
   appendtree(InstL2, insts[if(Expr1_1, Stat1, Stat2).InstL1], InstL3)
   ----------------
   ObjL1, ClVarL1, OThId
      |- clr(ObjId1, Mode, MethName, Env1, insts[if(Expr1, Stat1, Stat2).InstL1])
         -> ObjL1_1, ClVarL1_1,
            clr(ObjId1, Mode, MethName, Env1_1, InstL3), ThStatus ;
      provided diff(Expr1, true())  &  diff(Expr1, false());

If_False:
   ObjL, ClVarL, _
      |- clr(ObjId, Mode, MethName, Env, insts[if(false(), _, ElseStatement).InstL])
         -> ObjL, ClVarL,
            clr(ObjId, Mode, MethName, Env, insts[ElseStatement.InstL]), executing() ;

If_True:
   ObjL, ClVarL, _
      |- clr(ObjId, Mode, MethName, Env, insts[if(true(), ThenStatement, _).InstL])
         -> ObjL, ClVarL,
            clr(ObjId, Mode, MethName, Env, insts[ThenStatement.InstL]), executing() ;
```

Fig. 13. Excerpts from the Java Card dynamic semantics specification.

The formalisation of the JCRE covers the basic aspects of the runtime system such as loading and registration of applets, selection and deselection of applets, and the primitives necessary for communication with the runtime environment, i.e. the CAD simulator in our case. There is also support for saving and loading the state of a running card. This is used to simulate card insertion and withdrawal, but it is also useful for debugging since it makes it possible to get back to some interesting state without having to restart debugging from scratch, or if one simply would like to take a pause and continue from where one left off at some later point.

When it comes to the APIs, there are two basic approaches. One possibility is to describe them in Java, somehow making use of built-in primitives where necessary. The API classes would then be loaded into the simulation environment prior to the loading of any applets. The other approach is to describe the semantics of the API classes and their methods directly in Typol. Basically, this means that there is a set of hard-coded rules for each method which describes what happens when the method in question is invoked. We have chosen the latter approach for the following reasons:

- If we implement some of the APIs in Java Card, all the corresponding code will be interpreted by Typol rules which would take much more time. This API code would also be visible during debugging, even though it can be expected to be reasonably correct (once released for applet development, not during the development of the API code itself, of course).
- It is natural and simple to let the API methods be the built-in primitives. Otherwise, it would be necessary to provide access to lower level primitives by some other means. Moreover, in that case, many of the API methods would not do more than just invoke such a primitive anyway.

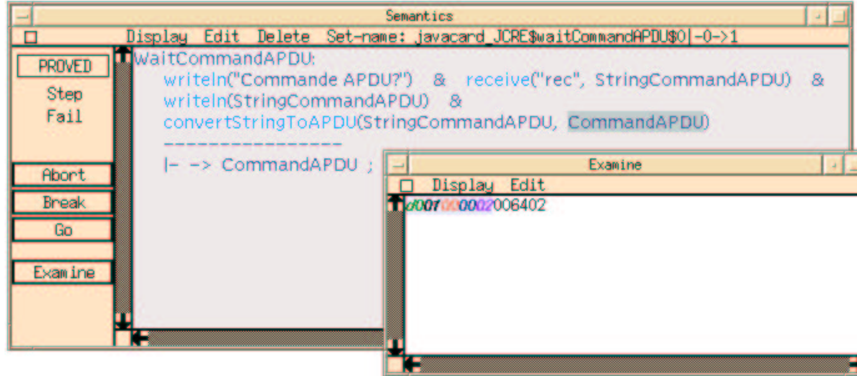

Fig. 14. The Java Card simulator during execution of the Purse applet.

- Having the semantics of important methods directly available as Typol rules hopefully makes it easier to prove properties about the specification as well as individual applets.

The specification of the Java Card APIs is not as yet complete. For example, the mechanisms for atomic transactions and object sharing, as well as the related API methods, are currently not in place.

During simulation, the developer can follow the execution in detail at a selectable speed through a graphical user interface. It is also possible to stop (interactively or by setting breakpoints), single step, and continue. The breakpoint and single stepping facilities currently work at the level of the semantic rules, not at the level of the Java Card source code. This is suitable for debugging the Java Card specification, or for a user who would like to learn about the semantics of Java card, but it is not ideal for debugging applications. Improved debugging support in this respect is something which we plan to look into.

Other features include textual browsers for inspection of variables and objects. Our Java environment [3] provides a dynamic, graphical view of the created object structures which would be useful also for the Java Card environment. This functionality could easily be carried over.

Figure 14 shows the Java Card simulator executing the Purse applet just after having received an APDU requesting the deposit of 100 Euros. The large window is the main Typol debug window which shows the current rule. It is equipped with buttons for controlling the execution. The small window shows the value of the variable `commandAPDU` which has been selected in the main window. The shown value is the received command APDU: note the value 0x0064 = 100 in the APDU's data field.

7 Conclusions and future work

This article described a Java Card programming environment which to a large extent has been generated from formal specifications of the syntax and semantics of Java Card. Through this approach, we were able to develop a set of tightly integrated tools with useful and novel functionality, such as the APDU format extraction, at a high level of abstraction. The main features of the environment from a user perspective are:

- Java Card-specific structure editor.
- CAD simulator which makes it easy to send and receive APDUs.
- Automatic extraction of the APDU formats from applet source code and automatic adaptation of the user interface of the CAD simulator accordingly.
- Semantics-based Java Card simulator with facilities for monitoring the execution and important data structures.

This makes the environment useful for applet developers, applet testers, and people wanting to learn about the Java Card semantics.

Furthermore, the fact that a formal, dynamic Java Card semantics is the basis of the simulator, means that it has been possible to test and debug this formalisation. This is important since a correct formalisation of the language semantics is a prerequisite for proving properties about programs and analyses concerning dynamic properties of programs, something we believe is particularly important in the context of typical smart card applications. For example, in the framework of Java cards with several applications, we are starting some work on static analysis of object sharing between applications [15]. One outcome of this research could be a facility for statically detecting sharing violations or the absence of such violations. Since a violation will cause an exception, it would clearly be reassuring to know that this cannot happen.

Our environment provides a Java Card specific editor. Thus the developer is assured that his code is Java Card code and is free from syntax error. This greatly reduces the number of Java compiler and converter errors and saves development time. Thus our aim is not to build an environment fulfilling commercial requirements, but we do think it provides innovative features that could be integrated to existing environments in order to improve them.

It still remains to include the complete set of APIs, and to support loading applets from more than one file. The current semantics is still missing a few features. This should be addressed. We would also like to further improve the debugging support, for instance by making it possible to set breakpoints, single step, etc. at the Java Card source level. We also aim to integrate capability for graphical object structure browsing along the lines found in Attali *et*

al. [3]. Finally, it is planned to move towards a SmartTools [6]-based solution, SmartTools being the 100 % Java successor to Centaur.

Acknowledgements

The authors would like to thank Valérie Pascual for helping out with Centaur.

References

[1] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[2] I. Attali, D. Caromel, H. Nilsson, and M. Russo. From executable formal specification to Java property verification. In Drossopoulou et al. [12], pages 1–7.
Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[3] I. Attali, D. Caromel, and M. Russo. A formal and executable semantics for Java. In *Proceedings of Formal Underpinnings of Java, an OOPSLA'98 Workshop*, Vancouver, Canada, October 1998. Technical Report, Princeton University.
`ftp://ftp-sop.inria.fr/oasis/Marjorie.Russo/OopslaWorkshop98.ps.gz`

[4] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Sousa, and S.-W. Yu. Formalization in Coq of the Java Card virtual machine. In Drossopoulou et al. [12], pages 50–56.
Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[5] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3)*, Boston, USA, 1988.

[6] J. Castella, J. Domingo-Ferrer, J. Herrera, and J. Planes. A performance comparison of java cards for micropayment implementation. In A. Watson J. Domingo-Ferrer, D. Chan, editor, *Smart Card Research and advanced Applications - Proceedings of CARDIS'2000*, pages 19–38, Bristol, UK, September 2000.

[7] Z. Chen. How to write a Java Card applet: A developer's guide. *Java World*, July 1999.
`http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard_p.html`

[6] `http://www-sop.inria.fr/oasis/SmartTools`

[8] Z. Chen. *Java Card Technology for Smart Cards - Architecture and Programmer'd Guide*. The Java Series. Addison-Wesley, Sun Microsystems, Palo Alto, California, USA, 2000.

[9] R. M. Cohen. Defensive Java Virtual Machine Specification version 0.5. Manuscript, 1997.

[10] C. Courbis. Simulation d'applications Java Card. `Rapport du DEA d'Informatique de Lyon (DIL)`, `ftp://ftp-sop.inria.fr/oasis/publications/1998/CarineCourbisStageDEA0798.pdf`, 1998.

[11] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In Alves-Foss [1], pages 41–82.

[12] S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors. *Formal Techniques for Java Programs 2000*. Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[13] D. Hagimont and J.-J. Vandewalle. Jccap: Capability-based access control for java card. In A. Watson J. Domingo-Ferrer, D. Chan, editor, *Smart Card Research and advanced Applications - Proceedings of CARDIS'2000*, pages 365–388, Bristol, UK, September 2000.

[14] L. Henrio. Tests interactifs d'applications Java Card. `Rapport de stage d'option scientifique de l'école Polytechnique`, `ftp://ftp-sop.inria.fr/oasis/publications/1999/LudovicHenrioStageX0699.pdf`, 1999.

[15] L. Henrio. Analyse de partage pour applications Java Card. stage de DEA, 2000.

[16] G. Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, Passau, Germany, 1987.

[17] J.-L. Lanet and A. Requet. Formal proof of smart card applets correctness. In J.-J. Quisquater and B. Schneier, editors, *Third Smart Card Research and Advanced Application Conference*, Louvain-la-Neuve, Belgium, September 1998.

[18] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In Alves-Foss [1], pages 119–156.

[19] Sun Microsystems. Java Card 2.1 platform. `http://java.sun.com/products/javacard/javacard21.html`, 1999.

[20] D. Syme. Proving Java type soundness. In Alves-Foss [1], pages 83–118.

[21] J.-J. Vandewalle and E. Vétillard. Developing smart card-based application using Java Card. In *Third Smart Card Research and Advanced Application Conference*, 1998. `http://www.gemplus.com/smart/r_d/publications/art1.htm`

A Source code for the Purse applet

```
/*******************************************************************
 *                          Purse Class
 *******************************************************************/
import javacard.framework.*;

class Purse extends Applet {
    protected final static byte PURSE_OP_CLA = (byte) 0xD0;
    protected final static byte PURSE_DEPOSIT = (byte) 0x01;
    protected final static byte PURSE_WITHDRAW = (byte) 0x02;
    protected final static byte PURSE_BALANCE_CHECK = (byte) 0x03;
    private final static short PURSE_MAX = (short) 0x7FFF;
    private short purseTotal;
    protected Purse() {
        purseTotal = (short) 0;
    }
    public static void install(byte[] bArray, short bOffset, byte bLength)
      throws ISOException {
        Purse myPurse = new Purse();
        myPurse.register();
    }
    public boolean select() {
        return true;
    }
    public void process(APDU apdu) throws ISOException {
        byte[] buffer = apdu.getBuffer();
        if (buffer[0] == PURSE_OP_CLA) {
            if (buffer[1] == PURSE_DEPOSIT)
                purseDeposit(apdu, buffer);
            else
                if (buffer[1] == PURSE_WITHDRAW)
                    purseWithdraw(apdu, buffer);
                else if (buffer[1] == PURSE_BALANCE_CHECK)
                    purseBalanceCheck(apdu, buffer);
        } else if (selectingApplet()) {
            /* Do nothing. */
        }
    }
    /********************** Private Methods ***************************/
    private void purseDeposit(APDU apdu, byte[] buffer) {
        int Lc = apdu.setIncomingAndReceive();
        short Le;
        short amountOfDeposit = Util.getShort(buffer, (short) 5);
        short temporaryPurseTotal = (short) (amountOfDeposit + purseTotal);
        if (amountOfDeposit > 0 && temporaryPurseTotal <= PURSE_MAX) {
            purseTotal = temporaryPurseTotal;
            Util.setShort(buffer, (short) 0, purseTotal);
            apdu.setOutgoingAndSend((short) 0, (short) 2);
        }
    }
    private void purseWithdraw(APDU apdu, byte[] buffer) {
        int Lc = apdu.setIncomingAndReceive();
        short Le;
        short amountOfWithdraw = Util.getShort(buffer, (short) 5);
        short temporaryPurseTotal = (short) (purseTotal - amountOfWithdraw);
        if (amountOfWithdraw > 0 && temporaryPurseTotal >= 0) {
            purseTotal = temporaryPurseTotal;
            Util.setShort(buffer, (short) 0, purseTotal);
            apdu.setOutgoingAndSend((short) 0, (short) 2);
        }
    }
    private void purseBalanceCheck(APDU apdu, byte[] buffer) {
        Util.setShort(buffer, (short) 0, purseTotal);
        apdu.setOutgoingAndSend((short) 0, (short) 2);
    }
}
```