Noname manuscript No.
(will be inserted by the editor)

Systematic Development of Java Card Applets

Wojciech Mostowski

Radboud University Nijmegen, The Netherlands
Computing Science Department
e-mail: W.Mostowski@cs.ru.nl

31 July 2006

Abstract We present an approach to systematic, tool-supported design and development of Java Card applets. We employ the Unified Modeling Language (UML) and formal methods for object-oriented software development in our approach. Our goal is to make Java Card applets robust "by design", to make the development process independent of the Java Card platform, and to enable applets to be verified formally by the KeY system. First we analyse the current situation of Java Card applet development, then we present a real life Java Card case study and describe the problems we found that should be addressed by systematic development. Finally, we propose some solutions to selected problems by using UML specifications, software design patterns, formal specifications and a modern CASE tool support.

Key words object-oriented software development – UML – OCL – formal verification – Java Card – smart cards

1 Introduction

In this article[1] we present an approach to systematic, tool-supported design and development of Java Card applets. Our work has been driven by the following goals: Java Card applets should be robust "by design", the development process should be independent of the Java Card platform, and it should be possible to formally verify applet correctness with the KeY system [1]. First we analyse the current situation of Java Card applet development, then we present a real life Java Card case study (`pam_iButton` [6]) and describe the problems we found that should be addressed by systematic development and formal verification. Based on the case study we propose solutions to selected problems by presenting a framework that incorporates the use of UML [29] specifications, software idioms and design patterns, a modern CASE tool support, as well as formal specification and verification to provide a systematic development process for Java Card applets.

1.1 Java Card

Java Card technology[2] [10] provides a means of programming smart cards with (a subset of) the Java programming language. Today's smart cards are small computers, providing 8, 16 or 32 bit CPU with clock speeds ranging from 5 up to 40 MHz, ROM memory between 32 and 64 KB, EEPROM memory (writable, persistent) between 16 and 32 KB and RAM memory (writable, non-persistent) between 1 and 4 KB. Smart cards communicate with the rest of the world through application protocol data units (APDUs, ISO 7816–4 standard). The communication is done in master-slave mode—it is always the master/terminal application that initialises the communication by sending the command APDU to the card and then the card replies by sending a response APDU (possibly with empty contents). In the case of Java powered smart cards (Java Cards) the card's ROM contains a Java Card virtual machine which implements a subset of the Java programming language and allows running Java Card applets on the card. The following are the features not supported by the Java Card language compared to full Java: large primitive data types (`int`, `long`, `double`, `float`), characters and strings, multidimensional arrays, dynamic class loading, threads and garbage collection. Selected Java Card smart cards go beyond those limitations, for example, by supporting the `int` type or garbage collection. However, such extensions to the Java Card standard are vendor-specific and in general should not be relied on. Most of the remaining Java features, in particular object-oriented ones

[1] This article is based on a paper that was published in the proceedings of Rigorous Object-Oriented Methods Workshop held in London, U.K., in March 2002.

[2] `http://java.sun.com/products/javacard/`

like interfaces, inheritance, virtual methods, overloading, dynamic object creation, are supported by the Java Card language. The card also contains the standard Java Card API, which provides support for handling APDUs, Java Card specific system routines, PIN codes, etc. A Java Card applet should implement the `install` method responsible for the initialisation of the applet and a `process` method for handling APDU communication with the terminal. A single Java Card smart card can host multiple applets.

1.2 Analysis of the Current Situation

Although the Java Card language is based on full Java, the nature of the Java Card environment (for example, constrained memory, no garbage collection) makes Java Card programming quite different from normal Java programming. Powerful development and modelling tools for Java are not Java Card "aware". Such Java tools can become helpful provided they can be customised to Java Card needs. However, this is not common practice. Instead, each Java Card vendor provides its own development environment and proposes its own Java Card specific solutions. Examples of such environments include the Sm@rtCafé Professional Toolkit,[3] Aspects Developer,[4] or iB-IDE tool.[5] Such tools try to ease the actual process of writing Java Card programs, installing them to the card and testing. However, they hardly ever provide the support for the design of Java Card applets on a higher level; in particular the use of UML development is not possible. A typical Java Card tool may provide the following functionality:

- automatic creation of the skeleton code for both the card applet and Open Card Framework [30] compliant Java terminal application,
- debugging tools with the possibility of running the card applet in an emulated environment,
- a tool to send APDU messages which is used to communicate with applets installed on a card without a terminal application, and to provide some card administration services—installing applets onto the card, erasing the card's memory, etc.

However, none of the tools provide any kind of high level modelling capabilities to design Java Card applets, nor do they provide any support for formal specification and verification.

A notable exception in this situation is the JCOP toolset from IBM.[6] The toolset is implemented as a plug-in for Eclipse.[7] This gives the possibility to use UML and formal verification plug-ins available for Eclipse within the same framework (see also Section 1.4).

We have just mentioned the need for the use of formal methods in Java Card applet development. This need is motivated by two reasons. First of all, smart card applets are usually security-critical. Secondly, in contrast to normal computer software, making updates on the cards distributed in large amounts is not possible. Thus, correctness of the card applet should be assured by the best possible means. At the same time Java Card applets seem to be suitable for formal verification because they are small in size and the Java Card programming language lacks some of the complications of the full Java language that makes formal verification difficult (such as threads, graphical user interfaces, or complex data types). Finally, a controlled software development process in general (such as the one we propose in this article, or an industrial one, for example the Nokia OK process) will benefit from adding the formal methods support to it. In the best case scenario, the obvious benefit of using formal methods is a formally verified program. In the worst case scenario, unverified formal specifications can always serve as "formal" documentation.

Based on our experience, we believe that the quality of Java Card applets would benefit from a controlled, well-defined, systematic development process with a possibility to formally verify applet properties. Our case study, presented in Section 2, show that this is indeed the case. Our further experience [15] shows that a properly designed and developed Java Card applet substantially eases the formal verification effort.

1.3 Related Work

Most of the literature on Java Card program development concentrates on verification. It can be divided into work on bytecode and source code level. An overview of work done on the bytecode level can be found in [7]. For the source code verification of Java Card applets we only name the most important tools and projects. The LOOP tool [19] employs the PVS theorem prover to prove properties about Java Card programs annotated with JML [22] (Java Modeling Language) specifications. In particular, [18] discusses the specification and verification of control flow properties for a Java Card applet. Similarly to our approach, the properties are described with a state machine, which is formalised in JML. JML is also used as a specification language in ESC/Java2 [11]. ESC/Java2 provides full automation, trading off the completeness and soundness of the static checking used to verify Java Card programs. [17] discusses systematic development of formally verifiable security protocols; state machines are used again and the resulting JML specifications are verified with ESC/Java2. Other tools based on JML are Krakatoa [23], Jive [26], and Jack [9]. The Krakatoa tool uses the Coq theorem

[3] `http://www.gi-de.com`
[4] `http://www.aspectssoftware.com/devtools/`
[5] `http://www.maxim-ic.com/products/ibutton/iB-IDE/`
[6] `http://www-306.ibm.com/software/wireless/wecos/tools.html`
[7] `http://www.eclipse.org`

prover as the basis for verification. The Jive system employs an extended Hoare style calculus implemented in the Isabelle and PVS theorem provers and provides a dedicated graphical user interface. Finally, the Jack tool also provides a dedicated graphical user interface to prove properties about JML annotated Java programs by employing several possible back-end theorem provers.

Our work has been done in the context of the KeY Project[8] [1]. One of its main objectives is to integrate formal methods with object-oriented software design to provide user-friendly formal verification environment for Java Card. This makes the KeY system to be best suited for our purposes. The design and specification languages used in KeY are respectively UML [29] (Unified Modeling Language) and OCL [32] (Object Constraint Language, now part of the UML standard). As an alternative to OCL, JML can also be used in the KeY system. In this article we show how we use the KeY system to support formal development of Java Card applets.

Apart from the formal approaches to Java Card development, the following should be mentioned. The Open Card organisation [30] bundles efforts to create a common and unified programming framework for writing terminal applications for smart cards from different manufacturers (Open Card Framework). The Global Platform organisation[9] [14] concentrates on providing a high level framework (API) for uniform treatment of certain Java Card applet features, including the applet life cycle (personalisation) states that we will discuss. Finally, Java Card applets are quite often security-critical; [21] shows how UML can be used to express security requirements during system development, and [8] discussed how to secure the remote method invocation (RMI) protocol of the new Java Card standard.

1.4 Our Approach

Our approach to the development of Java Card applets uses UML modelling techniques, software patterns and incorporate formal methods in an incremental way. By incremental we mean that the use of formal methods should be optional and it should be up to the developer (who might be unfamiliar with formal methods) at which level of detail formal methods are used. This view is stressed in [1,5]. To enable and ease the use of formal methods, we try to provide means of creating certain kinds of formal specifications semi-automatically in two ways. The first way is applying software and specification patterns solving some common problems to the application design [13]. Such patterns usually need to be instantiated with parameters, but giving the parameters is the only job that is required from the developer. The second way is to create a specification out of UML state diagrams, also possibly taking some parameters from the user. Both ways enable partial specifications to be created without detailed knowledge about the formal specification language. The created specifications are well-formed by design and ready to be formally verified.

Having all this support we can make Java Card applets robust by design and easier to verify. To achieve our goals we need support from a modern, fully customisable UML CASE tool, as well as a suitable formal verification system. As already mentioned, we use the KeY system [1] as the basis in our framework. The KeY system extends a commercial UML CASE tool with formal verification modules in a seamless way. CASE tools that the KeY system currently supports include the Together tool family from Borland[10] and Eclipse. The Together Control Center that we used for this work provides state of the art support for UML and is fully extensible through its Java open API. Therefore, we can use powerful UML support as well as formal verification tools in one framework. One of the KeY extensions to the CASE tool provides a library of common design and specification patterns to support creation of OCL specifications [13]. Another extension we introduced to the CASE tool supports low level Java Card development tasks, like compiling, installing, or testing applets, either in a simulated environment or on real smart cards [27]. This allows us to make our solutions to Java Card design issues independent of the actual Java Card platform and of any vendor-specific development environment. Another possible tool setup that would allow to achieve such flexibility is Eclipse equipped with the KeY plug-in for verification support, the Together Architect plug-in for UML modelling, and the JCOP toolset for low level Java Card support.

In this work we limit ourselves to Java Card applets residing on smart cards, that is, we do not consider the problems of developing the terminal application. The main reason is that terminal applications are mostly regular Java programs and usually are part of a bigger project, to which existing development techniques can be applied, for example the Unified Process [20]. Moreover, suitable efforts to support uniform development of terminal applications are carried out already, for example in the Open Card Framework [30]. The applets themselves seem to be too small to be subject to a "big" process like UP—we believe that developing a Java Card applet should be seen as a small subprocess, that needs a Java Card specific approach with more focus on formal aspects.

The next section presents a motivating Java Card case study based on which we identify Java Card specific design issues and problems we want to address (Section 3). In Section 4 we walk through and re-engineer our example to present our framework. In Section 5 we discuss the solutions in the framework that address the

[8] `http://www.key-project.org`
[9] `http://www.globalplaform.org`
[10] `http://www.borland.com/together/`

problems we identified. Finally, Section 6 summarises the article.

2 Case Study: `pam_iButton`

We start with briefly discussing our case study, which illustrates some of the common design requirements for Java Card applets. The `pam_iButton` package was written by Dierk Bolten and is available free of charge [6]. The package allows a Linux user to authenticate himself to the system by inserting an iButton into the reader instead of giving the password. A Java-powered iButton[11] is a Java Card smart card embedded in a button shaped case implementing Java Card API version 2.0 (which differs substantially from the current Java Card API 2.2.1). It supports the `int` data type and provides garbage collection. The most recent Java-powered iButton has an 8 bit processor, a cryptographic (RSA and SHA1) coprocessor and 130 KB of non-volatile RAM memory. The `pam_iButton` package consists of:

- a PAM (Pluggable Authentication Module) Linux system library which is responsible for authentication,
- the Java Card applet (`SafeApplet`) which performs the actual authentication on the iButton device,
- a setup utility to configure the necessary system files and administer the iButton applet.

The following is an example `pam_iButton` usage scenario. First a Linux user account needs to be set up to be able to use the iButton authentication. The user is assigned a unique ID number and an RSA key pair (private and public key) is generated on the iButton and stored together with the user ID in the iButton memory. Multiple user IDs can be registered on the iButton. The public key is then retrieved by the system from the iButton and stored in the system configuration file together with the user ID number. The iButton is ready to be used for authentication. When the user wants to be authenticated he types in his login name. The system looks up his ID number and encrypts a random message with the user's public key. The encrypted message and the user ID number are sent to the iButton applet. The applet checks if the user is registered and if so, it decrypts the message with the private key, computes the SHA1 hash code from the decrypted message and sends it back to the system. The system compares the received SHA1 code with its own and if they match the user is authenticated successfully.

The most important commands that the `SafeApplet` accepts are:

- Store data: stores temporary data for a subsequent command.
- Authenticate user: given the user ID performs the challenge-response authentication described earlier. In response sends back the SHA1 code of the message. The encrypted message has to be sent beforehand with the 'store data' command.
- Set PIN (PIN protected): sets a new PIN for PIN protected commands.
- Generate key pair (PIN protected): given the user ID generates an RSA key pair (the generation is done on the card) and stores it together with the user ID in the applet memory. In response sends back the public part of the key.
- Get public key: given the user ID sends back the public part of the key.
- Delete key pair (PIN protected): given the user ID removes this user's key pair entry from applet memory.
- Get key information: sends back the ID numbers of users registered in the applet.

Any command (except for the first and the last) sent to the applet can possibly result in an error. In that case, instead of the expected answer, the error code (status word) is sent back to the terminal indicating the cause of the error. Internally in the Java Card applet this is done by throwing an appropriate `ISOException`.

3 Design Issues for Java Card Applets

Here we describe the design issues that came up while we were studying the example and we try to list some common requirements for a Java Card applet.

Some questions that immediately came to mind were the following. Who is the owner of the applet PIN: the Linux system administrator or the user? Who is the person to set up iButton for authentication: the system administrator, the user, both? What are the applet deployment steps: who is responsible for installing the applet to iButton, when is iButton ready to be passed to the user for regular use (that is, when does the applet get personalised)? Should it be possible for one iButton applet to be used on two different Linux systems? Answers to some of the questions imply answers to some of the other questions. For example, if a single applet can be used on many different systems then it certainly should be the user owning the applet PIN and it should be the user that sets up the system configuration (probably through some administrator privileged system tool, which itself needs careful design).

One way or the other, the answers to the questions above are not provided by the design of the applet, at least not explicitly, and since this kind of application is security-critical, these issues require careful thought.

Next we took a closer look at the protocol that is used to exchange information between the terminal application and the iButton applet. We discovered the following:

[11] `http://www.maxim-ic.com/products/ibutton/`

- The applet does not impose or check the order on commands it receives. This opens the possibility of attack scenarios where commands are sent to the card in a different order than the applet expects, with the aim to corrupt the state of the applet. In case of this particular applet we did not find a sequence of command calls that could put the applet in an unrecoverable state, but we did manage to corrupt the applet state with garbage data, causing some malfunctioning. The recovery process required tedious manual command sending.
- The problem with out-of-sequence commands is aggravated by the fact that some of the commands require input that does not fit into a single APDU, so there are multiple APDU messages being sent for one command. However, the applet does not check whether the right number of APDU messages in a right order is sent. In particular, this may cause the applet to run out of memory, as all the incoming data is being accumulated in the applet. Moreover, there are no integrity checks on the data sent.
- The last thing we found strange about the protocol is that the PIN is sent along with each command that requires PIN authentication. Generally there is nothing wrong with it, but it produces unnecessary overhead and it is different from the commonly used solution of establishing the PIN once per command exchange (card) session.

Another potential problem (which also applies to some other iButton applets that we have seen, not only the one presented) is unconstrained dynamic memory allocation. For iButtons this is not an issue, as they implement garbage collection. In general, however, to make applets portable between different Java Card platforms, the size of memory dynamically allocated should be reduced as much as possible. Otherwise, due to lack of memory, the applet may stop working at any point of execution.

Extensive testing of `SafeApplet` revealed one more problem: if the user tears the iButton out of the reader during authentication, the applet can be left in an inconsistent state and fail to work correctly in subsequent authentication sessions. The design of a Java Card applet should take the possibility of card tears into consideration and try to make the applets as robust and card tear proof as possible. Our further research [15] shows that formal verification is the right technique to ensure that applets are card tear safe.

The last problem is that `SafeApplet` allows two or more different key pairs registered with the same user ID number. While this was the original author's deliberate design decision, we think the applet should forbid to make double entries of this kind, instead of making the user responsible for avoiding this.

Our analysis of the `SafeApplet` above lead us to the following design requirements and programming guidelines for Java Card applets:

- the applet has to be robust: it should be protected against malicious terminal applications and against tearing the card out of the reader,
- the applet deployment steps and life cycle should be well-defined, and controlled by the applet itself to prevent abuse,
- the command exchange protocol should be well-defined, constrained and controlled by the applet to disable illegal sequences of commands, and, for any commands that consist of multiple APDU messages, illegal sequences of those messages,
- the applet should minimise its memory footprint, and allocate all its required memory at installation time, to avoid `OutOfMemoryException` when the card is in operation.

It should be noted that such requirements are common conventions in Java Card applet development, see for example [10, 24]. The example of our case study merely emphasises the importance of these requirements. In the next section we show how these requirements can be addressed and enforced during design.

To end this section we want to stress that we do not want to impose any particular design decisions for Java Card applets (for example, which deployment steps the applet should have, whether a certain command should be PIN protected, etc.). We only want to support the design process and provide the developer with means and tools to make those design decisions and control the development steps in a systematic way. The design decisions we present in the next section are only examples among many possible alternatives and for real-life applications such decisions should be made by a domain expert.

4 Developing Java Card Applets

We now present how one can go about designing and developing a Java Card applet by going through the case study again and re-engineering it in a well-defined way. The large parts of the example that we present should give the reader the complete overview of the development process. In the next section we will discuss the crucial features of our framework that make the developed applet robust and free of the problems we identified earlier.

4.1 Applet Life Cycle States

First we define the life cycle states of the applet (deployment steps). These are the distinguished states that the applet will go through during its lifetime. Our applet can be in one of the following four state:

Initialised This is the state of the applet just after installing (downloading) it onto the card, but before setting some data in the applet that is necessary for proper functioning of the applet,

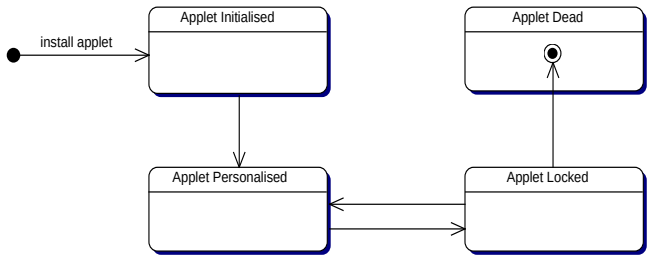

Fig. 1 `SafeApplet` life cycle states

Personalised This is the state after setting the data on the applet. This is also the applet's "normal operation" state,

Locked This is the state after something goes wrong during normal applet usage, for example, after the user entered the wrong PIN a number of times and the applet access is blocked temporarily,

Dead This is the state after an unrecoverable misuse of the applet. In our case after the user enters a wrong master PIN, which can only be presented for verification once and is only allowed to be presented in the locked state.

An applet goes only once through the initialised state during its lifetime and also it can never leave the dead state after entering it. It can however move between personalised and locked states many times during its lifetime. These constraints are best expressed in a UML state diagram, see Figure 1. Later we will show what the exact conditions are that cause an applet life cycle state change. One last thing that is required of the applet is that it enforces the card terminal session to be restarted after the applet has moved from one life cycle state to another.

4.2 Applet Commands

Now we can start defining the commands that the applet should support. The new set of commands is a slightly redesigned set of commands that we described in Section 2. For each of the commands we give a name, we say if it can be invoked in a given applet life cycle state and if it is a user PIN or master PIN protected command (for each state separately). Table 1 shows the list of commands we are interested in. Below we give an informal description of the commands:

authenticateUser This command is used to authenticate a given user through a challenge-response protocol. A single person owns one smart card with a single `SafeApplet`. However, there can be more than one user ID registered in the applet. Hence, the command has to specify which user is to be authenticated, by giving a user ID.

updateUserPIN This command changes the user's PIN. Depending in which life cycle state the applet is, different security measures are taken to protect the command. For example, since the personalisation step should be taken in the issuer's trusted area, it is not necessary to require PIN authentication for updating the user's PIN in initialised state. On the other, if the applet is, for example, in the locked state, the master PIN has to be verified before any user PIN changes can be made.

Name/State	Init.	Person.	Locked	Dead
authenticateUser	No	Yes	No	No
updateUserPIN	Yes	Yes (P)	Yes (MP)	No
setMasterPIN	Yes	No	No	No
verifyUserPIN	No	Yes	No	No
verifyMasterPIN	No	No	Yes	No
generateKeyPair	No	Yes (P)	No	No
deleteKeyPair	No	Yes (P)	No	No
getPublicKey	No	Yes	No	No
disableUser	No	Yes (P)	No	No
enableUser	No	Yes (P)	No	No
getKeysInfo	No	Yes	No	No

Table 1 `SafeApplet` commands. (P) denotes PIN protected commands, (MP) denotes Master PIN protected command.

setMasterPIN This command sets the master PIN for the applet. It is the only command required to make the applet personalised, hence it moves the applet from the state initialised to personalised.

verifyUserPIN This command performs the verification of the user's PIN, which after successful verification stays validated until the end of the card/terminal session. All PIN protected commands can check the PIN validity flag.

verifyMasterPIN Same as the previous one, except for the master PIN. This command can only be invoked in the locked state to unlock the applet. Usually, the master PIN is only allowed to be presented once; after an unsuccessful try the applet becomes dead.

generateKeyPair This command generates a key pair (public and private) for a given user ID and stores this in the applet memory for future use.

deleteKeyPair This command removes the keys for a given user ID from the applet memory.

getPublicKey This command retrieves the public part of a key for a given user ID.

disableUser, enableUser These commands disable and enable the authentication of a given user ID. The user may wish to block the usage of `SafeApplet` when he has to pass the smart card to somebody else (for example, to download some other applets).

getKeysInfo This command returns all registered user IDs (for administrative purposes).

4.3 Command Invocation Protocol

The information we gathered so far is sufficient to define the protocol that `SafeApplet` should follow. We

do this by presenting further state diagrams, one inside each state representing a single applet life cycle state. We will call the new sub-states the command states. In our applet we distinguish four different command states/ categories:

Selected This is the initial state after the applet is selected by the Java Card run-time environment (this is triggered by the terminal application),

Application This is the state for "every day use" commands. For `SafeApplet` authenticateUser is the only such command,

User administration This is the state for user-level administration commands, for example updateUserPIN,

System administration This is the state for system-level administration commands, for example, generateKeyPair.

Commands belonging to one category should not be interleaved with commands belonging to another category. Not all of the applet life cycle states will contain all of these command states. In particular, initialised and locked life cycle states contain only one meaningful command state—user administration. In such cases, we merged the user administration command state with the selected state for simplicity.

At this stage of the design we also precisely define when the applet changes its life cycle state.

Let us start with the initialised life cycle state. Figure 2 shows the corresponding state diagram. The black dot represents the state in which the applet is not active and needs to be selected. When the applet is deselected by the Java Card run-time environment or a card reset occurs the applet has to be selected again. There is only one command state inside the life cycle state initialised and only two commands are possible, namely updateUserPIN and setMasterPIN. The invocation of updateUserPIN is optional during the personalisation process—the applet issuer may wish to release the applet without the user PIN set. Once setMasterPIN is invoked successfully (no error occurs and the input data for setting the master PIN is not corrupted) the applet changes its life cycle state to personalised and never goes back to initialised. The card/terminal session has to be restarted after a life cycle state change, which means that no further commands can be invoked after a successful setMasterPIN until the applet is selected again.

Figure 3 shows the details of the personalised life cycle state. This is the applet's main operational state in which most of the application and administration commands are enabled. As before, after selection the applet is in selected command state. Once a command belonging to one of the three categories (application, system administration, user administration) is invoked the command state is changed accordingly and the applet stays in this state until the end of the session. To enter a different command mode the session has to be restarted. The verifyUserPIN command is treated in a special way—since the PIN is required by the commands both in system and user administration modes, invoking verifyUserPIN does not change the command state of the applet. However, if the PIN verification fails the maximum allowed number of times (userPINBlocked) the applet life cycle state is changed to state locked where special rules apply for unblocking the PIN. The only application mode command is authenticateUser, the only user administration command is updateUserPIN. In system administration mode the following commands are enabled: generateKeyPair, deleteKeyPair, getPublicKey, getKeysInfo, disableUser, and enableUser.

Finally we describe the command protocol for the applet life cycle state locked (Figure 4). As in the case of life cycle state initialised there is only one command state, user administration, which is merged with the state selected. The only two commands that are allowed here are verifyMasterPIN and updateUserPIN. After successful master PIN verification (MasterPINOK) the updateUserPIN command sets the new user PIN and unblocks it, moving the applet back to personalised life cycle state. In case the master PIN verification failed the applet life cycle state changes to dead from which there is no return—the applet becomes unoperational.

All command invocation sequences that are not allowed by the diagrams are forbidden—in case of any attempt to violate the defined protocol the applet should end the communication immediately by throwing a suitable exception.

Note that we already gave a lot of semi-formal information about the applet we are building without writing or presenting a single line of Java Card code. The state diagrams are useful documentation that helps to understand the intended behaviour of the applet.

4.4 Command Processing

For each of the applet commands we define what parameters it takes, whether there should be extra integrity checks on the data (to detect possible transmission errors), and whether the command has to be split into several APDU messages (again with the indication of whether extra integrity checks are required). Tables 2 and 3 show the complete list.

Now we can show the actual implementation of the command dispatching methods. As examples we discuss updateUserPIN, getPublicKey and authenticateUser. Let us start with updateUserPIN. Recall that this command had a conditional PIN check depending on the current applet life cycle state. It also expects 8 bytes of input data and there is a required integrity check on the data. There is no response data, just a status word is sent back to the terminal indicating the (un)successful invocation of the command. The command should also follow the protocol we defined. Here is the code:

```
/**
```

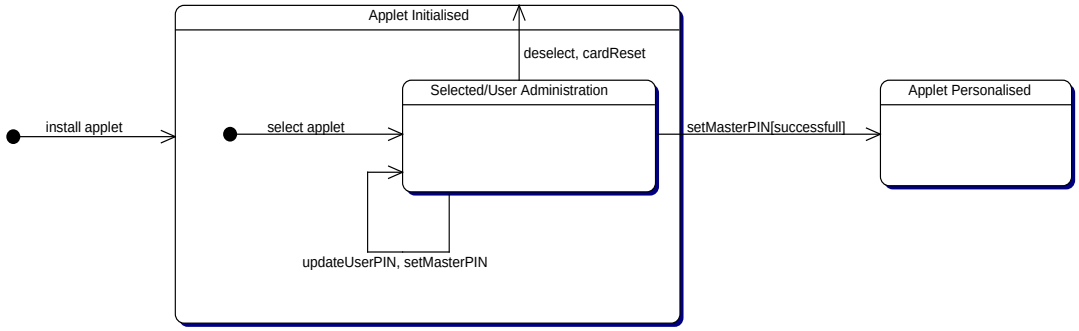

Fig. 2 Command states in the initialised life cycle state

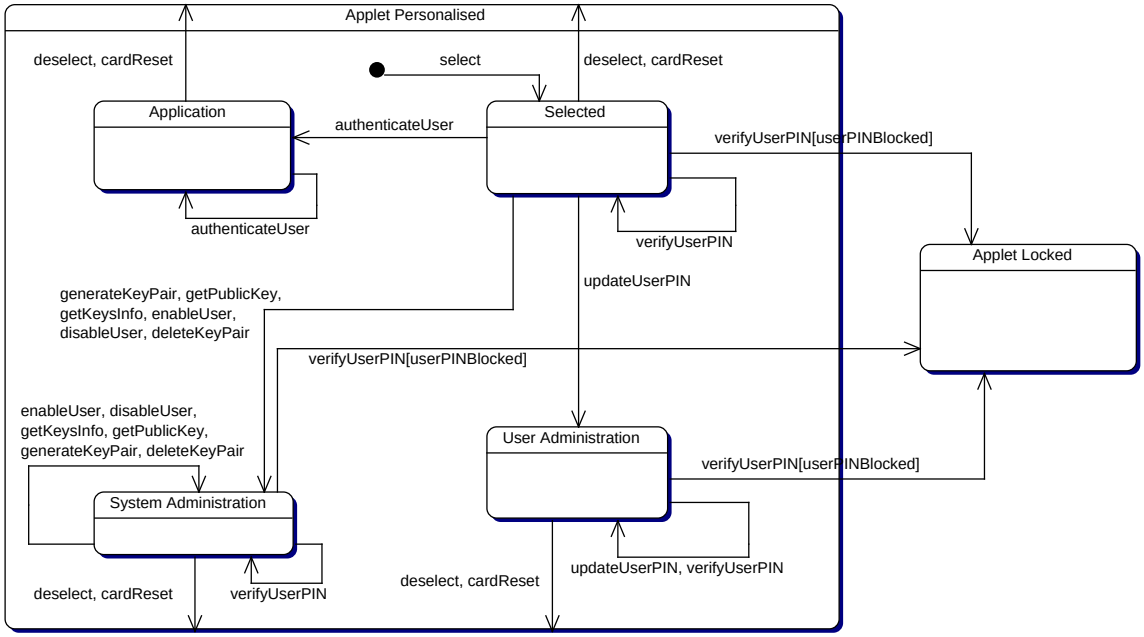

Fig. 3 Command states in the personalised life cycle state

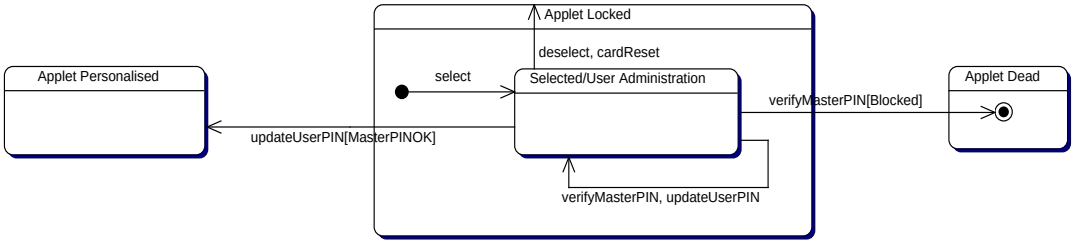

Fig. 4 Command states in the locked life cycle state

```
  * @param apdu the incoming APDU message
  *             to dispatch
  */
public void dispatchUpdateUserPin(APDU apdu) {
  updateCommandState(UPDATE_USER_PIN);
  switch (curr_applet_state) {
    case AS_INITIALISED:  break;
    case AS_PERSONALISED: checkPIN(); break;
    case AS_LOCKED:       checkMasterPIN(); break;
  }
  readInput(apdu, (short)28); // puts input in temp
  verifyInput((short)8);
  userPIN.update(temp, (short)0, (byte)8);
  if (curr_applet_state == AS_LOCKED) {
    setAppletState(UPDATE_USER_PIN,
                   AS_PERSONALISED);
  }
}
```

Name	Input	Length	Integ.	APDU
authenticateUser	User ID, challenge	1+256	No	Many
updateUserPIN	New PIN	8	Yes	1
setMasterPIN	PIN	16	Yes	1
verifyUserPIN	PIN	8	Yes	1
verifyMasterPIN	PIN	16	Yes	1
generateKeyPair	User ID	1	No	1
deleteKeyPair	User ID	1	No	1
getPublicKey	User ID	1	No	1
disableUser	User ID	1	No	1
enableUser	User ID	1	No	1
getKeysInfo	None	0	No	1

Table 2 Command parameters

Name	Response	Length	Integ.
authenticateUser	SHA1 code	20	No
updateUserPIN	None	0	No
setMasterPIN	None	0	No
verifyUserPIN	None	0	No
verifyMasterPIN	None	0	No
generateKeyPair	None	0	No
deleteKeyPair	None	0	No
getPublicKey	Public key	131	Yes
disableUser	None	0	No
enableUser	None	0	No
getKeysInfo	User IDs	#Users	No

Table 3 Command responses

The call to `updateCommandState` makes sure that the command is invoked according to the protocol. The `updateCommandState` implements a state machine that follows the diagrams shown before. The `switch` statement performs the conditional PIN check (the `AS` prefix stands for applet state). Then the input is read, which has to be 8 bytes long plus 20 bytes for the SHA1 code for data integrity verification. After the data is retrieved from the APDU message it is stored in the `temp` array. The `temp` array is allocated once during applet installation and is sufficiently big to serve all command dispatching methods, thus keeping memory consumption fixed. The method `verifyInput` performs the actual verification of the data stored in the `temp` array. Then the actual user PIN update happens. If the applet happens to be in locked life cycle state then it switches back to personalised state after successful update (`setAppletState`).

Let us take a look at getPublicKey now. This command does not require any PIN checks, it expects one byte of input data without integrity verification, and sends back 131 bytes of response plus an additional 20 bytes of SHA1 code for integrity verification on the terminal side. We skip the actual key retrieval code as it is not relevant at this point. Here is the code:

```
/**
  * @param apdu the incoming APDU message
  *             to dispatch
  */
public void dispatchGetPublicKey(APDU apdu) {
    updateCommandState(GET_PUBLIC_KEY);
    readInput(apdu, (short)1);
    // retrieve the key, prepare the response data
    // in temp
    integrifyOutput((short)131);
    sendResponse(apdu, (short)151);
}
```

The `sendResponse` method simply sends the data prepared in the `temp` array back to the terminal.

Let us take a look at authenticateUser now. This command is the only one that requires multiple APDU messages. No PIN check or input data integrity verification is required. The response is 20 bytes of SHA1 code calculated from the received message. Here is the implementation:

```
/**
  * @param apdu the incoming APDU message
  *             to dispatch
  */
public void dispatchAuthUser(APDU apdu) {
  updateCommandState(AUTH_USER);
  readBigInput(apdu, (short)257);
  if (multiple_package == (byte)0) {
    // everything read, process bigtemp,
    // prepare the response in temp
    sendResponse(apdu, (short)20);
  }
}
```

The methods `readBigInput` and `updateCommandState` make sure that the data contained in different APDU messages are sent in the correct order and are not interleaved by any other commands. This is done by using global applet variables and requiring the multiple APDU messages sent over to the applet to be properly marked as we will show shortly.

Now we give some more details about the auxiliary methods that are used by command dispatching methods. The `readInput` method reads the input from the incoming APDU into the `temp` array in a standard way, reporting to the terminal any possible data length inconsistencies by throwing an appropriate exception:

```
/**
  * @param apdu the incoming apdu to read data from
  * @param expectedLength the expected data length
  *                       to read
  */
public void readInput(APDU apdu,
                      short expectedLength) {
  byte buffer[] = apdu.getBuffer();
  short apduDataOffset = 0;
  short dataLength = unsigned(
                       buffer[ISO7816.OFFSET_LC]);
  if (dataLength != expectedLength) {
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
  }
  short bytesRead = apdu.setIncomingAndReceive();
```

```
  while (bytesRead > 0) {
    if ((short)(bytesRead + apduDataOffset) >
           expectedLength) {
      ISOException.throwIt(
        ISO7816.SW_WRONG_LENGTH);
    }
    Util.arrayCopy(buffer, ISO7816.OFFSET_CDATA,
                   temp, apduDataOffset,
                   bytesRead);
    apduDataOffset += bytesRead;
    bytesRead = apdu.receiveBytes(
      ISO7816.OFFSET_CDATA);
  }
}
```

The methods `setAppletState`, `updateCommandState`, and `readBigInput` are more interesting. The first one is responsible for changing the applet life cycle state. It is the calling method's responsibility to ensure that the condition for changing this state is satisfied (for example, that master PIN has been verified when updateUserPIN changes the state from locked to personalised). The method manipulates the global applet variable called `curr_applet_state`. Here is a small part of `setAppletState`:

```
/**
  * @param command the code of the command changing
  *                the state
  * @param newstate the new state to be set
  */
public void setAppletState(byte command,
                           short newstate) {
  switch (command) {
    // ...
    case UPDATE_USER_PIN:
      // 'masterPIN.isValidated() == true'
      // should hold here
      if (curr_applet_state == AS_LOCKED) {
        curr_applet_state = newstate;
        curr_command_state = CS_START;
      }
      break;
    // ...
  }
}
```

The `updateCommandState` and `readBigInput` methods share some global applet variables to ensure that the protocol is followed. The global variable `multiple_package` indicates whether a command spread over multiple APDU messages is being processed—when equal to 0 the currently processed command is a single APDU command, when greater than 0 it is equal to the identifier of the multiple APDU command being processed. The method `updateCommandState` first checks if the life cycle state of the applet is the dead state and if so, it throws an exception interrupting the communication. Then it checks if there is multiple APDU processing in progress. If so, it further checks whether the current command belongs to the sequence of currently processed multiple APDU messages, an exception is thrown if there is a mismatch. Finally, the method checks if the command invocation is according to the protocol defined. The global applet variable `curr_command_state` stores the current command state (selected, application, user administration or system administration). The code follows the diagrams shown before, throwing an exception if the command is not invoked according to the protocol:

```
/**
  * @param command the code of the invoking
  *                command
  */
public void updateCommandState(byte command) {
  if (curr_applet_state == AS_DEAD) {
    ISOException.throwIt(SW_APPLET_DEAD);
  }
  if (multiple_package != (byte)0 &&
      command != multiple_package) {
    ISOException.throwIt(
      SW_COMMAND_OUT_OF_SEQUENCE);
  }
  switch (command) {
    case VERIFY_USER_PIN:
      if (curr_applet_state != AS_PERSONALISED) {
        ISOException.throwIt(
          SW_COMMAND_OUT_OF_SEQUENCE);
      } else {
        if (curr_command_state == CS_APPLICATION) {
          ISOException.throwIt(
            SW_COMMAND_OUT_OF_SEQUENCE);
        } else {
          // do nothing, there is no state change
        }
      }
      break;
    case UPDATE_USER_PIN:
      // ...
  }
}
```

The `readBigInput` method uses both global variables and the form of the APDU to control the multiple APDU communication. The `p2` header byte of the incoming APDU indicates the total number of APDU messages to come, the `p1` header byte indicates which APDU message is being received ("`p1`-th out of `p2` messages"). The global variables `multiple_curr` and `multiple_total` are used to keep track of this. Whenever a multiple APDU message is received `p1` and `p2` are checked against global variables to verify that the proper sequence is maintained. Then the data from the APDU is appended to the `bigtemp` array which collects the data from the multiple APDU messages. The code for `readBigInput` is:

```
/**
  * @param apdu the incoming apdu to read data from
  * @param expectedLength the expected data length
  *                       to read
  */
public void readBigInput(APDU apdu,
```

```
                          short expectedLength) {
  byte buffer[] = apdu.getBuffer();
  byte ins = buffer[ISO7816.OFFSET_INS];
  byte p1 = buffer[ISO7816.OFFSET_P1];
  byte p2 = buffer[ISO7816.OFFSET_P2];
  if (p1 == (byte)0 && multiple_total == (byte)0) {
    multiple_total = p2;
    multiple_package = ins;
  } else {
    if (p1 >= p2 || p2 != multiple_total ||
        p1 != (byte)(multiple_curr + (byte)1)) {
      ISOException.throwIt(ISO7816.SW_WRONG_DATA);
    }
  }
  multiple_curr = p1;
  // append the data from APDU to bigtemp array
  multiple_readnum = apduDataOffset;
  if ((byte)(multiple_curr + (byte)1) ==
        multiple_total) {
    resetMultiple(); // data in bigtemp ready
  }
}
```

5 The Framework

We did not discover any problems during extensive testing of the resulting, re-engineered applet. In particular, the card tear problem is no longer present. This was achieved mostly by enforcing the applet to follow strict command exchange protocol we defined during the design. Thus, the most important guideline in our framework is to define the applet life cycle and communication protocols with UML state diagrams and enforce the applet to adhere to these protocols by embedding corresponding state machines into the applet itself. Moreover, the actual command processing (types of parameters, responses, etc.) should be defined and implemented in a strict way. To achieve that, we used dedicated methods (`readInput`, `sendResponse`, etc.) in our applet responsible for reading the input and sending the output. Such methods are largely independent of the actual applet and can be used in other applications.

One artifact of the design is the semi-formal documentation that makes understanding the workings of the applet much easier for the developers and prospective users of the applet. Further support for the developers can be provided by the CASE tool, the Global Platform framework and Java Card RMI, and the use of formal specification and verification techniques. We discuss these possibilities in the following sections.

5.1 Support from the CASE Tool

We used the support of the Together Control Center tool in many places. For the low level tasks, such as compiling, installing and testing the applet, we used an extension module we wrote ourselves [27]. Some parts of the applet were created with the same module by using Java Card specific code patterns, which were used to introduce skeletons of the command dispatching methods automatically. Furthermore, the KeY extensions were used to introduce formal specifications into the design and to perform formal verification (see next subsections). The remaining code was written "by hand", however we still see possibilities to introduce further automation to the process with the support of a CASE tool in the following ways:

- Having methods like `readInput`, `readBigInput`, `verifyInput`, etc. among the standard set of Java Card helper methods. This can easily be supported by the Together Control Center.
- Generating (possibly with a little of developers help) the code for `setAppletState` and `updateCommandState` methods from the state diagrams also incorporating formal specifications for verification. A tool (AutoJML) that does exactly this, only it produces JML specifications instead of OCL ones, is described in [16]. Such a tool should be easily pluggable into the Together Control Center, for example Together Control Center can automatically create code from sequence diagrams and vice versa.
- The PIN check routine seem to be a good candidate for a pattern, too, as it is done in a very similar way in every Java Card applet. There is a global applet object representing the PIN and there is one APDU command that verifies the delivered PIN, sets the validation flag of the PIN object for the current terminal session, and returns the result of PIN verification back to the terminal. Then any command requiring PIN authentication can refer to the PIN object by a single method call. This scheme seems to be very basic and there is only one simple class in the Java Card API that provides PIN functionality. However, the problems we discovered in the original `SafeApplet` (Section 3) show that even such simple ideas can be easily misinterpreted. Thus, a simple software pattern can take the responsibility to create a correct PIN check mechanism in the applet.

5.2 Support from the Global Platform Framework and Java Card RMI

The Global Platform framework [14] provides a standard on-card API to control some of the aspects of applet management, including applet life cycle states. In principle there is no obstacle to use the standard Global Platform API to control applet life cycle states in the context of our work. We do, however, see two shortcomings of this approach:

- the set of life cycle states in Global Platform is predefined (for example, there is no distinction between locked applet and dead/terminated applet). Thus the

use of Global Platform API may not always give the required flexibility to control applet life cycle states,
– state transitions between different life cycle states are hard-coded into the API without the (source code) implementation being available. So, it is impossible to verify the correctness of the transitions with respect to the specification, represented by the UML state diagram or expressed in OCL. In such case, it is only possible to rely on the natural language description of the state transitions that the Global Platform documentation [14] provides.

Other facilities that the Global Platform framework provides are mechanisms to control the integrity and confidentiality of the APDU messages sent to the card. So, by using the Global Platform framework, it is not necessary to implement methods like `verifyInput`; instead Global Platform functionality can be used. We did not do this in our case study, because iButtons do not support Global Platform framework.

Finally, we should mention the Remote Method Invocation framework for Java Card offered by the most recent Java Card standard (2.2.1) [31]. RMI greatly reduces applet complexity and considerably eases the applet design and development process. The use of RMI would be a perfect solution to many problems we have described, but unfortunately, there is still only a handful of smart cards on the market that support the new Java Card standard with RMI.

5.3 Formal Specification and Verification

For most of the methods of our applet, it is very clear from looking at the code that they correctly implement the semi-formal specifications given by the state diagrams. The `setAppletState`, `updateCommandState` and `readBigInput` and possibly `readInput` methods require a bit more attention and this is where we turn to formal specification and verification.

First, we can define the behaviour described by state diagrams more formally by giving OCL specifications. We can use OCL invariants to express the required relation between the applet life cycle state and other aspects of the applet's state, for example:

```
context SafeApplet
inv: self.curr_applet_state = AS_LOCKED implies
     self.userPIN.getTriesRemaining() = 0

inv: self.curr_applet_state = AS_DEAD implies
     self.masterPIN.getTriesRemaining() = 0
```

Next we can limit a set of possible command states in a given life cycle state by invariants such as:

```
context SafeApplet
inv: self.curr_applet_state = AS_LOCKED implies
       self.curr_command_state = CS_START or
       self.curr_command_state = CS_SELECTED
```

Finally, we can formally specify methods such as `setAppletState` and `updateCommandState` with pre- and postconditions, for example:

```
context SafeApplet::setAppletState(
  command : Integer, newstate : Integer)
pre: command = UPDATE_USER_PIN and
       newstate = AS_PERSONALISED implies
     self.masterPIN.isValidated() and
       self.curr_applet_state = AS_LOCKED
post: self.curr_applet_state = AS_PERSONALISED and
        self.curr_command_state = CS_START

context SafeApplet::updateCommandState(
  command : Integer)
post: command = VERIFY_USER_PIN
      and
      self.curr_applet_state@pre = AS_PERSONALISED
      and
      self.curr_command_state@pre <> CS_APPLICATION
      and
      self.curr_command_state@pre <> CS_START
           implies
        self.curr_command_state =
          self.curr_command_state@pre
```

These specifications precisely specify the constraints expressed by state diagrams and it should be possible to just generate them automatically, possibly with a little bit of user intervention [16].

The second set of specifications makes sure that the `readInput` and `readBigInput` methods behave in a consistent way. The following OCL invariants express the consistency conditions that the global applet variables (`multiple_package` and `multiple_readnum`) used by the read methods should satisfy:

```
context SafeApplet
inv: self.multiple_readnum <= self.bigtemp->size()
inv: self.multiple_package <> 0 implies
       self.multiple_curr < self.multiple_total
inv: self.multiple_package = 0 or
       self.multiple_package = AUTH_USER
inv: self.multiple_total > 0 implies
       self.multiple_package <> 0
```

Here we also stated that the authenticateUser command is the only one that can spread over multiple APDU messages.

The next are two preconditions that make sure the read methods do not exceed the temporary array space they operate on:

```
context SafeApplet::readInput(
  apdu : APDU, expectedLength : short)
pre: self.temp <> null and
       expectedLength <= self.temp->size()

context SafeApplet::readBigInput(
  apdu : APDU, expectedLength : short)
pre: self.bigtemp <> null and
       expectedLength <= self.bigtemp->size()
```

The Java Card pattern for read methods discussed in Section 5.1 should not just produce the code for such methods, but also these associated specifications.

Of course one may want to give some more detailed specifications of the applet describing its functionality or some safety properties [25]. In the next section we show how existing features of the KeY system can be used to produce such a specification. Here we briefly discuss other situations where formal specification can prove itself helpful.

Suppose we would like to extend our applet to keep track of unsuccessful authentication attempts and disable the access once a certain number of unsuccessful attempts has been reached (similarly to PIN verification). This is quite straightforward to program—a counter variable needs to be increased after each failed attempt and once some threshold value is reached the following access attempts are rejected. However, when coded uncarefully, the counter may be increased during rejected attempts as well. Then after reaching the maximal value for a data type used (say `byte`) it will leap back to -128, meaning we end up in an undesired state, breaching security. A typical security related specification idiom that could be used here would be that a card stays blocked after the maximum number of tries has been reached until it is explicitly released, for example, by giving the master PIN. To verify such a property one needs formalisation of Java integer arithmetics that handles properly the overflow behaviour of Java integer types. The KeY system both supports the specification idioms [13] and contains formalisation of Java integer arithmetics as part of the KeY specification library [4].

A very tricky aspect of smart card programs is ensuring data consistency in the case when the applet's execution terminates abnormally by tearing the card out of the reader. Here we can also turn to formal methods. It is not sufficient to express such constraints on data consistency as invariants; invariants may be temporarily broken when a method executes, but a card tear at such a point in time would break data consistency. Instead we need a stronger notion of invariant, which has to hold throughout program execution. This cannot be expressed in plain OCL, but the Dynamic Logic for Java Card used in the KeY system can express such properties (*strong invariants*) [3].

The OCL specifications discussed above—and the additional data consistency constraints expressed in Dynamic Logic—have been formally verified using the KeY system. Verification of other, more advanced properties can also be performed relatively easily with the KeY system [15,28]. As we noted in Section 1.3, other tools can be used for verification as well—[18] discusses verification of similar protocol properties with the LOOP tool.

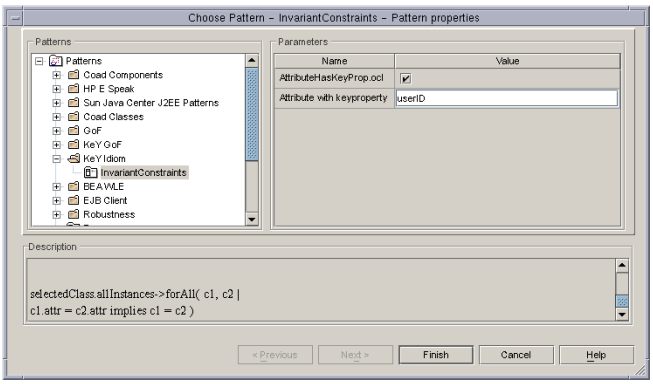

Fig. 5 Applying specification patterns in the KeY system

5.4 Employing the KeY System

Here we show how the KeY system can be used to support creation of the formal specifications. Recall that one of the problems we found in `SafeApplet` was that a single user ID can be registered more than once in the applet. First let us look at the class representing a single user record in the applet:

```
public class User {
    boolean empty = true;
    boolean enabled = true;
    byte userID = (byte)0;
    KeyData keydata = null;
}
```

We would like to specify that there should not exist two (non empty) objects of this class in our applet having the same user ID. Then it can be verified formally that any code that operates on those records does not violate this condition. The condition just mentioned is a slight modification of a standard specification pattern in the KeY system called AttributeHasKeyProp, shown in Figure 5. After the pattern is applied, the following invariant is produced for the `User` class:

```
context User:
inv: User.allInstances->forAll(c1, c2 |
       c1.userID = c2.userID implies c1 = c2)
```

After a small modification we get what we want:

```
context User:
inv: User.allInstances->forAll(c1, c2 |
       not c1.empty and not c2.empty and
       c1.userID = c2.userID implies c1 = c2)
```

The KeY system provides a whole library of such specification patterns (some also based on the GoF patterns [12,2]) applicable to any Java Card program and, more generally, any Java program.

6 Summary

Based on the `SafeApplet` example we presented our approach to the systematic development of Java Card applets. We have shown how UML can be used to specify an applet behaviour, using state diagrams, and how

such specifications can be translated into actual code. We have also shown how we can apply formal specification and verification in JAVA CARD development. A modern CASE tool plays an important role in our approach: it supports UML specifications, software patterns, formal verification (KeY system), and, last but not least, easy testing of JAVA CARD applets through the CASE tool extension module. Large parts of the code we have shown were developed by hand, but we were precisely following the UML state diagrams. Consequently, the coding was quite straightforward and almost a one pass process—we made the applet work in the expected way in a very short time and extensive testing revealed no problems in the applet. Further research [15,28] showed that formal verification of JAVA CARD applets is feasible and can formally ensure robustness of the applet.

Acknowledgements We would like to thank anonymous reviewers for their helpful comments, and Erik Poll for his insights and involvement in improving this article.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, February 2005.
2. Thomas Baar, Reiner Hähnle, Theo Sattler, and Peter H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In K. Mehlhorn and G. Snelting, editors, *Informatik 2000, 30. Jahrestagung der Gesellschaft für Infomatik*, pages 389–404. Springer, September 2000.
3. Bernhard Beckert and Wojciech Mostowski. A program logic for handling JAVA CARD's transaction mechanism. In Mauro Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.
4. Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, volume 2999 of *LNCS*, pages 207–226. Springer, April 2004.
5. Dominique Bolignano, Daniel Le Métayer, and Claire Loiseaux. Formal methods in practice: the missing link. a perspective from the security area. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19–23, 2000*, volume 2067 of *LNCS*. Springer, 2001.
6. Dierk Bolten. PAM authentication with an iButton. `http://www-users.rwth-aachen.de/dierk.bolten/pam_ibutton.html`.
7. Robert Boyer. Proving theorems about JAVA and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003.
8. Richard Brinkman and Jaap-Henk Hoepman. Secure method invocation in JASON. In *5th USENIX Smart Card Research and Advanced Application Conference, San Jose, CA, U.S.A.*, pages 29–40, 2002.
9. Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. JAVA applet correctness: A developer-oriented approach. In *Proceedings, Formal Methods Europe 2003*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.
10. Zhiqun Chen. *JAVA CARD Technology for Smart Cards*. Addison Wesley, June 2000.
11. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for JAVA. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1999.
13. Martin Giese, Reiner Hähnle, and Daniel Larsson. Rule-based simplification of OCL constraints. In Octavian Patrascoiu et al., editor, *Workshop on OCL and Model Driven Engineering at UML2004, Lisbon*, pages 84–98, 2004.
14. Global Platform Organization. *Card Specification, Version 2.2.1*, March 2003. `http://www.globalplaform.org`.
15. Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.
16. Engelbert Hubbers and Martijn Oostdijk. Generating JML specifications from UML state diagrams. In *Proceedings of the Forum on specification & Design Languages (FDL 2003)*, pages 263–273. University of Frankfurt, 2003. Proceedings appeared as CD-Rom with ISSN 1636-9874.
17. Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. Implementing a formally verifiable security protocol in JAVA CARD. In *Proceedings of the 1st International Conference on Security in Pervasive Computing*, volume 2802 of *LNCS*, pages 213–226. Springer, 2004.
18. Bart Jacobs, Martijn Oostdijk, and Martijn Warnier. Source code verification of a secure payment applet. *Journal of Logic and Algebraic Programming*, 58(1-2):107–120, 2004.
19. Bart Jacobs and Erik Poll. JAVA program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4–6, 2003*, volume 3233 of *LNCS*, pages 134–153. Springer, 2004.
20. Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
21. Jan Jürjens. Towards development of secure systems using UMLsec. In Heinrich Hußmann, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference*, volume 2029 of *LNCS*, pages 187–200. Springer, 2001.

22. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.
23. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa tool for certification of Java/Java Card programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.
24. Renaud Marlet and Cédric Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.
25. Renaud Marlet and Daniel Le Métayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.
26. Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000, Berlin, Germany*, volume 1785 of *LNCS*, pages 63–77. Springer, April 2000.
27. Wojciech Mostowski. Java Card tools for Together Control Center. `http://www.cs.chalmers.se/~woj/papers/jctools.pdf`.
28. Wojciech Mostowski. Formalisation and verification of Java Card security properties in Dynamic Logic. In Maura Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference, Edinburgh, Scotland*, volume 3442 of *LNCS*, pages 357–371. Springer, April 2005.
29. Object Modeling Group. *Unified Modelling Language Specification, version 2.0*, October 2004.
30. Open Card homepage. `http://www.opencard.org/`.
31. Sun Microsystems, Inc., Santa Clara, California, USA. *Java Card 2.2.1 Application Programming Interface*, October 2003.
32. Jos Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, 2003.