Rigorous development of JAVA CARD applications

Wojciech Mostowski
Chalmers University of Technology, Göteborg, Sweden
Computing Science Department
e-mail: woj@cs.chalmers.se

Abstract

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

1 Introduction

In this paper we present an approach to rigorous, tool supported design and development of JAVA CARD applications. Our goal is to make JAVA CARD applications robust "by design", to make the development process independent of the JAVA CARD platform used and to enable applications to be formally verified by the KeY system [1]. First we analyse the current situation of JAVA CARD application 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 rigorous development and formal verification. We propose some solutions to selected problems by presenting a framework that incorporates the use of UML specifications, software idioms and design patterns, formal specifications and a modern CASE tool support to provide a systematic, rigorous development process for JAVA CARD applications.

1.1 JAVA CARD

We start with a short introduction to JAVA CARD technology [7]. JAVA CARD provides 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's 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 case of JAVA powered smart cards (JAVA CARDs) besides the operating system 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. Some of the actual JAVA CARD devices go beyond those limitations and support for example the `int` data type and garbage collection. Most of the remaining JAVA features, in particular object oriented ones 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, Application IDentifiers (AIDs), JAVA CARD specific system routines, PIN codes, etc. A proper JAVA CARD applet should implement the `install` method responsible for the initialisation of the applet (usually it just calls the applet constructor) and a `process` method for handling incoming command APDUs and sending the response APDUs back to the host. There can be more than one applet existing on a single JAVA CARD, but there can be only one active at a time (the active one is the one most recently selected by JAVA CARD run-time environment).

1.2 Analysis of the current situation

JAVA CARD technology is relatively young and still developing and so are design and development techniques for JAVA CARD applications. Although JAVA CARD language is based on full JAVA the nature of the JAVA CARD environment (e.g. constrained memory, no garbage collection) makes JAVA CARD programming very different from normal JAVA programming. Powerful development and modelling tools for JAVA are not JAVA CARD "aware". Such a JAVA tool can become helpful provided it can be customised to JAVA CARD needs. This however is not the common approach being taken. Instead, each JAVA CARD vendor provides its own development environment and proposes its own JAVA CARD specific solutions. The provided tools try to ease the actual process of writing JAVA CARD programs, installing them to the card and testing, but they hardly ever provide the support for the design of JAVA CARD applications in a more abstract sense. Our experience is based on using the JAVA-powered iButtons (`http://www.ibutton.com/`), which we use in our research, and the development environment (iB-IDE—`http://www.ibutton.com/iB-IDE/`) provided for this platform, but most of the following statements apply to other environments too. The iB-IDE tool provides the following functionality: automatic creation of the skeleton code for both the card (iButton) application and OpenCard Framework [14] compliant JAVA host application with convenience methods for dispatching user defined command APDUs and converting data types, debugging tools with the possibility of running the card applet in an emulated environment and finally a very handy APDU sender which is used to communicate with the card applets without a host application and to provide some card administration services—downloading applets to the card, erasing card's memory, etc. The tool however does not provide any kind of modelling or design support for building JAVA CARD applications, nor does it provide any support for formal specification and verification. One more thing which should be mentioned here is the fact that the JAVA CARD virtual machine in iButton devices implements garbage collection and the iB-IDE skeleton code and example applets make heavy use of that fact which means that those solutions are not (easily) portable to other JAVA CARD platforms. In contrast to this, SUN's JAVA CARD reference development kit (`http://java.sun.com/products/javacard/`) provides very nice examples which take into account common JAVA CARD limitations and proposes a very elegant way of writing JAVA CARD applets, but the development kit itself does not contain any user friendly tools to create the applications, the only thing available are command line tools for compilation and running the applets in a simulated/emulated environment.

The next issue we want to discuss is the need for the use of formal methods in JAVA CARD application development. There are two reasons for this. First of all smart card application 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 application should be done by best means possible. At the same time JAVA CARD applications 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 (like threads, graphical user interfaces, complex data types).

Taking all this into account it becomes clear that the development of JAVA CARD applications needs to be done in a systematic, well defined, rigorous way giving the possibility to formally verify the application's properties.

1.3 Related work

We briefly discuss some other work that is done in the areas around JAVA and JAVA CARD program development and verification. The largest project concerned with the use of formal methods for JAVA CARD is the VerifiCard project [16, 9, 3]. Its goal is to provide the basic technology for verification of both the JAVA CARD platform and JAVA CARD applications. The project does not concentrate so much on the actual design process of JAVA CARD applications. GemPlus (a major producer of JAVA CARDs) carries out some work towards automated testing of JAVA CARD applications [12]. The OpenCard organisation [14] bundles efforts to create a common and unified programming framework for writing host/terminal applications for JAVA CARDs coming from different manufacturers (Open Card Framework). Compaq's ESCJava project [8] develops a tool for automatic, static checking of JAVA programs annotated with specifications written in (a subset of) JAVA Modelling Language. Finally [10] shows how UML can be used to express security requirements during system development.

1.4 Our approach

In our approach to development of JAVA CARD applications we use UML modelling techniques, software patterns and incorporate formal methods in a 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, a view stressed in [5, 1]. To enable and ease the usage of formal methods we try to provide means of creating certain kinds of formal specifications semi automatically in two ways. The first by applying software and specification patterns solving some common problem to the application design [2]. Such patterns usually need to be provided with parameters during instantiation to create a proper specification, but giving the parameters is the only job that is required from the developer. The second way is to create a specification out of certain kinds of UML diagrams (possibly taking some parameters from the user). Both enable creating partial specifications 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 applications robust and secure by design and easier for verification. For successful verification one needs a suitable formal verification system, and this is where we turn to the KeY project [1, 11] which we use as our framework. The KeY project is concerned with integrating formal methods and object oriented software design. It incorporates both into one framework by extending a commercial UML CASE tool with formal verification modules in a seamless way. The usage of the KeY formal verification extensions is fully integrated into the tool. The KeY project also proposes the usage of a set of standard specifications that can be applied to common problems. We will use some of those specification patterns in our approach. In order to achieve our goals we also need support from a modern, fully customisable UML CASE tool. The one we use is Together ControlCenter from TogetherSoft (`http://www.togethersoft.com/`). It provides very good support for UML and a Java open API, by which most of the tool features can be accessed and extended, which makes it extremely suitable for our purposes. Since it is the same tool the KeY project uses it is easy to integrate our solutions into the KeY system. Another reason to use an independent CASE tool is that we want to make our solutions to Java Card design issues independent of the actual Java Card platform and vendor specific development environment obtaining generic, powerful UML support at the same time. It also should be mentioned that at this point of our work we limit ourselves to the card applications (Java Card applets). At this stage we don't consider the problems of developing the host application, however, we will address this problem in future.

In this paper we present a motivating Java Card case study (Section 2) based on which we identify Java Card specific design issues and problems we want to tackle (Section 3). In section 4 we present our framework to solve some of the problems: first we show how UML state chart diagram can be used to define a Java Card applet behaviour and command invocation protocol, then we show how the actual implementation is derived from the diagrams in a rigorous way. Next we show how formal specifications are used to assure extra reliability and enable formal verification of a Java Card application. We also give an example of how the KeY system is used to construct specifications. Section 5 gives the future directions and Section 6 summarises the paper.

2 Case study: `pam_iButton`

It is time to present our case study, upon which we build up some of the common design requirements for JAVA CARD applications. 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 device into the reader instead of giving the password. A JAVA-powered iButton is a JAVA CARD device implementing JAVA CARD API version 2.0 (which differs substantially from the current JAVA CARD API 2.1.1 specification) with `int` data type support and garbage collection. The most recent JAVA-powered iButton has an 8 bit processor, cryptographic (RSA and SHA1) coprocessor and 200 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 on the system side, a setup utility to configure the necessary system files and administrate the iButton and the JAVA CARD applet (`Safe_Applet`) which performs the actual authentication on the iButton device.

The following is an example `pam_iButton` usage scenario. First a Linux user account needs to be setup to be able to use the iButton authentication. The user is assigned a unique user id number and a pair of private and public RSA keys is generated on the iButton and stored together with the user id in the iButton's memory (many different users can be registered on one iButton). The public key is then retrieved by the system from the iButton and stored in the system configuration file together with user's 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 user's public key. The encrypted message and the user's id number is 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 value from the decrypted message and sends it back to the system. The system compares the received SHA1 value with its own and if they match the user is authenticated successfully.

We now give some more details about `Safe_Applet`. Here is the list of the most important and interesting command APDUs that the applet accepts:

- 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 code (PIN code protected)—sets a new PIN code for PIN protected commands.
- Generate key pair (PIN code protected)—given the user id generates an RSA key pair (the generation is done on the card) and stores it together with user id in the applet's 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 code protected)—given user's id removes this user's key pair entry from applet's memory.

- Get key information—sends back the id numbers of users registered in the applet.

A command (except for the first and the last) sent to the applet can cause an error condition in which case instead of the expected answer the error code (status word) is sent back to the host indicating what the error was caused by. Internally in the JAVA CARD applet this is done by throwing an appropriate exception (`ISOException`).

3 Design issues for JAVA CARD applications

In the following section we will describe what issues came up while we were studying the example and we will try to list some common requirements that a JAVA CARD application should satisfy.

One of the first questions that came to mind were the following. Who is the owner of the applet PIN code, Linux system administrator or the user? Who is the person to setup iButton for authentication, the system administrator, the user, both? What are the applet deployment steps, who's responsible for installing the applet to iButton, when is iButton ready to be passed to the user for regular usage (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, e.g., if a single applet can be used on many different systems then it certainly should be a user owning the applet's PIN code and it should be a user that sets up the system configuration, probably through some administrator privileged system tool, which itself needs to be very carefully designed.

One way or the other, the answers to the posed questions are not provided by the design of the applet, at least not explicitly, and since this kind of application is security critical, things like those mentioned above need to be well defined and carefully thought through.

Secondly, we took a closer look at the protocol that is used to exchange information between the host application and the iButton applet and we discovered the following. There is no order imposed on command sequences: in one possible type error attack scenario first the 'store temporary data' command is sent to the applet with an intention for this data to be used with a given subsequent command call (say 'authenticate user'), but then a different command is sent which also relies on 'store temporary data', in which case the latter gets wrong data (e.g. intended for 'authenticate user'), which may cause corruption of the applet data. We only mentioned the 'authenticate user' command that relies on 'store temporary data', but there are other commands doing this too. 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 it is definitely possible to corrupt the applet state with wrong data, causing some (recoverable) malfunctioning. There are also no integrity checks on the data being sent along, which connects to the previous as well as the next problem. Some of the commands may require input that does not fit into a single APDU, so there are multiple APDUs being sent, but there is no control whether the proper number of APDUs in a proper order is sent (in particular this may cause the applet to run out of memory). The last thing we found strange about the protocol is that the PIN code is sent along with each command that requires PIN code authentication. Generally there is nothing wrong with it, but it produces overhead and it is different from the commonly established solution of presenting the PIN code once per command exchange (card) session.

Another thing which we find problematic (and it applies to iButton applets in general, not only the one presented) is the unconstrained memory usage. The iButton applets make heavy use of garbage collection and do a lot of dynamic memory allocation. Not watching for memory usage makes life much easier for the developer, but it also makes the applet much less robust—it may decline proper functioning at any point of execution where memory allocation occurs when there is no free memory on the device left (and this is very likely to happen on such a memory constrained device). Such an approach to JAVA CARD programming also makes the applications not portable to other JAVA CARD devices that don't support garbage collection.

While testing `Safe_Applet` we came to another interesting issue. If the user rips out the iButton from the reader during authentication the applet is not functioning properly any more during subsequent authentication sessions. At first it seemed to be a simple programming mistake, but it turned out not to be. We still haven't figured out what is the cause for this behaviour, but we strongly suspect the JAVA CARD run-time environment to be the source of the problem. However the point here to make is that the design of a JAVA CARD application should take similar possibilities into account and try to make the applets as robust and rip-out proof as possible, assuming of course that the underlying JAVA CARD run-time environment is implemented properly.

The last thing we want to point out is that `Safe_Applet` allows two different key pairs registered with the same user id number. While this is the author's deliberate design decision, we think it should be forbidden by the applet to make double entries of this kind, instead of making the user responsible for controlling the state of the key pair entries in the applet.

Some of the problems we mentioned may seem not to be an issue for such a small application as `Safe_Applet`, but we want to make the JAVA CARD design and development process scalable, and for bigger applications the problems raised here definitely become serious issues which need to be addressed. Based on what we have already described we now list some of the common design issues in JAVA CARD development we will try to face and give some support to in the next section:

- the applet has to be robust in the sense that it should be protected against malicious host application, tampering with and hopefully ripping out the card from the reader,
- the applet deployment steps and applet's life cycle should be well defined and controlled by the applet itself disabling improper applet usage,
- the message exchange protocol should be well defined, constrained and controlled by the applet to disable illegal command invocation sequences (this also includes proper support for the commands requiring data to be sent in multiple APDUs),
- the applets should be very careful about the memory (to say the least), here we would like to take the safest approach of allocating all the memory an applet may ever want to use during applet installation time [7].

To end this section we want to make an important note. In our work we don't want to impose certain ways of solving design problems for JAVA CARD applications (e.g. what actual deployment steps the applet should have or whether a certain command should be PIN code protected or not), 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 process in a rigorous way. The design decisions we present in the next section are only examples among many possible, the design decisions in real-life JAVA CARD world should be done by a domain expert.

4 Developing JAVA CARD applications

We now present how one can go about designing and developing a JAVA CARD application by going through the case study again and reengineering it, this time doing things in a more defined way. We will also point to places where CASE and verification tools play an important role. We will not develop the whole `Safe_Applet` here, only the things important to exemplify our approach.

4.1 Applet life states

First we define the life states of the applet (deployment steps). These are the distinguished states that the applet will go through during its life time. For our application we can limit ourselves to the following:

- applet is selectable, this is the state of the applet just after installing (downloading) it to the card, but before setting some data in the applet that is necessary for proper functioning of the applet,
- applet is personalised, this is the state after setting the data on the applet. This is also the applet's "normal operation" state,
- applet is locked, this is the state after something went wrong during normal applet usage, e.g. the user entered the wrong PIN code a number of times and the applet access is blocked temporarily,
- applet is dead, this is the state after an unrecoverable misusage of the applet, in our case when the user enters a wrong master PIN code, which can only be presented for verification once and is only allowed to be presented in locked state.

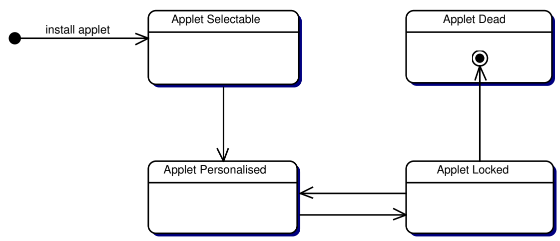

Figure 1: `Safe_Applet` life states

An applet goes only once through the selectable state during its life 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 life time. We will show later what are the exact conditions that cause an applet's life state change. One last thing that we will require from the applet is that it enforces the card terminal session to be restarted after the applet has moved from one life state to another. Figure 1 shows a UML state diagram presenting the life states idea we have just described.

4.2 Applet commands

We are now at the point where we can start defining the command APDUs that the applet should support. For each of the commands we give its name, we say if it can be invoked in a given applet life state and if it is a user or master PIN code protected command (for each state separately). Table 1 shows the list of commands we are interested in. Without specifying formally what are a given command's parameters and responses we now give an informal description of the intended meaning of the commands:

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

updateUserPIN This command changes the user's PIN code to a new one. Depending in which life state the applet is, different security measures are taken to protect the command. For example, since the personalisation step should be done in the issuer's trusted area it is not necessary to require PIN authentication for updating the user PIN in selectable state.

setMasterPIN This command sets the master PIN for the applet. It's the only command required to make the applet personalised, hence after successful invocation it should move the applet from state selectable to personalised.

verifyUserPIN This command performs the verification of the user PIN which after successful verification stays validated until the end of the terminal

Name/State	Selectable	Personalised	Locked	Dead
authenticateUser	No	Yes	No	No
updateUserPIN	Yes	Yes (PIN)	Yes (Master PIN)	No
setMasterPIN	Yes	No	No	No
verifyUserPIN	No	Yes	No	No
verifyMasterPIN	No	No	Yes	No
generateKeyPair	No	Yes (PIN)	No	No
getPublicKey	No	Yes	No	No
disableUser	No	Yes (PIN)	No	No
enableUser	No	Yes (PIN)	No	No
getKeysInfo	No	Yes	No	No

Table 1: Possible `Safe_Applet` commands

session. All the commands that are PIN code protected can check the PIN code validity flag.

verifyMasterPIN Same as previous one, just for the master PIN. This command can only be invoked in the locked state to enable special behaviour 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 pair of keys (public and private one) for a given user ID and stores this information in the applet's memory for future use.

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

disableUser, enableUser Those commands are used to disable and enable the authentication of a given user specified by a user ID. The user may wish to block the usage of `Safe_Applet` when he has to pass the JAVA CARD device (iButton) to somebody else (in order to download some other applets for example).

getKeysInfo This command should inform the owner of the applet about all user IDs registered in it (for administrative purposes).

The commands that can be invoked during the operational mode of the applet (personalised) fall into certain categories, which in turn define possible sequences of command invocations. For example authenticateUser is (the only) application command that is going to be used on a daily basis, while updateUserPIN is a user administration command, which is invoked rarely (if ever) and should not be mixed with application mode commands. Commands like generateKeyPair or getPublicKey fall into system administration category.

4.3 Command invocation protocol

The information we gathered so far is sufficient to define the protocol that `Safe_Applet` should follow. We do this by presenting further state charts, one inside each state representing a single applet life state. We will call the new substates the command states. In our application we distinguish four different command states. The initial one is selected state. This is after the applet is selected by the JAVA CARD run time environment (this is usually triggered by host application). Then, depending on the commands invoked, the applet can be in one of the three command states: application, user administration or system administration.

Both in selectable and locked life states the command states selected and user administration are in some sense equivalent and we put them together as one state. At this stage we also define precisely under what conditions the applet changes its life state.

Let us start with applet life state selectable. Figure 2 shows the corresponding state chart diagram. The black dot represents the state in which the applet is not active and needs to be selected. When the applet gets deselected by the JAVA CARD run-time environment or a card reset event occurs the applet has to be selected again. There is only one command state inside the life state selectable and only two commands possible. The invocation of updateUserPIN

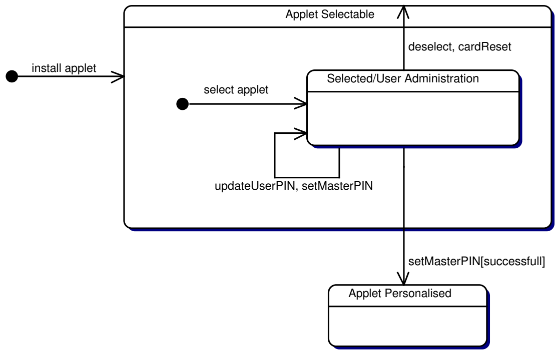

Figure 2: Command states in selectable life state

is optional during the personalisation process—the applet issuer may wish to release the applet without user PIN code 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 state to personalised and never goes back to selectable. The card/terminal session has to be restarted after a life state change, which means that no further commands can be invoked after a successful setMasterPIN until the applet is selected again.

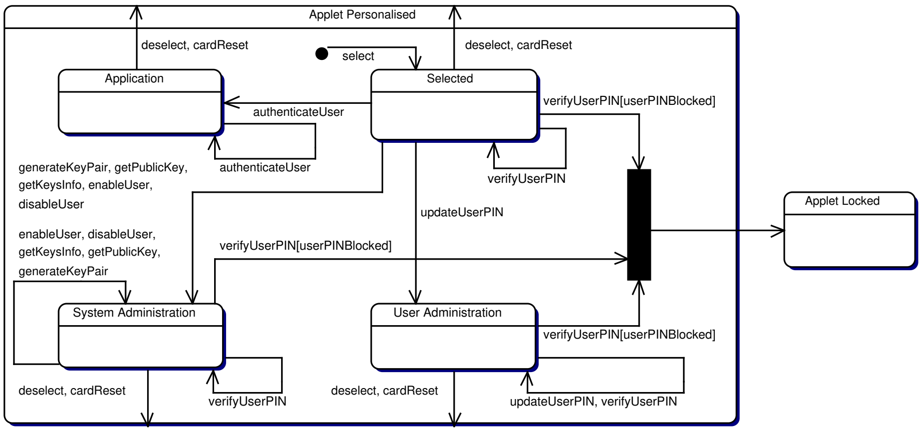

Figure 3: Command states in personalised life state

Figure 3 shows the details of personalised life 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 classes (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 code is required by the commands both in system and user administration modes invoking verifyUserPIN does not change command state of the applet. However if the PIN verification failed the maximum allowed number of times (userPINBlocked) the applet's life state is changed to state locked where special rules apply for unblocking the PIN code. The only application mode command is authenticateUser, the only user administration command is updateUserPIN and in system administration mode we have the following commands enabled: generateKeyPair, getPublicKey, getKeysInfo, disableUser and enableUser.

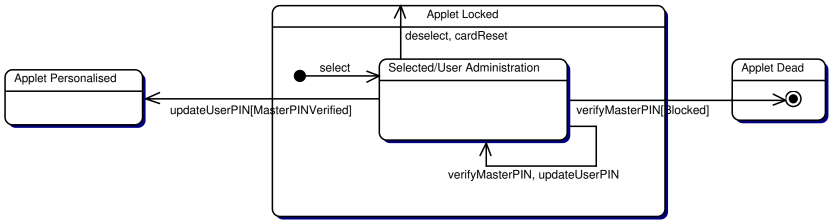

Figure 4: Command states in locked life state

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

All the command invocation sequences that are not defined 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 proper exception.

Before continuing we would like to make a comment: notice 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 so far.

4.4 Command processing

It is time to focus on the actual command processing. For each of the commands we listed we now define which parameters a given command takes, whether there should be extra integrity checks on the delivered data, if the command is allowed to spread across multiple APDUs and what is the response data (again with the indication of whether extra integrity checks are required). Tables 2 and 3 show the complete list.

Taking into account everything we have said so far about commands we now show how the actual dispatching of the commands can be done inside the JAVA CARD applet based on the examples of updateUserPIN and authenticateUser. Let's start with updateUserPIN. Recall that this command had a conditional PIN check depending on the current applet's life 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 host indicating the (un)successful invocation of the command. The command should also follow the protocol we defined. Here is the code:

Name	Input parameters	Length	Integrity	APDUs
authenticateUser	User ID, the challenge	1+256	No	Many
updateUserPIN	New PIN data	8	Yes	1
setMasterPIN	PIN data	16	Yes	1
verifyUserPIN	PIN data	8	Yes	1
verifyMasterPIN	PIN data	16	Yes	1
generateKeyPair	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 data	Length	Integrity
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
getPublicKey	User's public key	128	Yes
disableUser	None	0	No
enableUser	None	0	No
getKeysInfo	User IDs	0...Max Users	No

Table 3: Command responses

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

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. 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 packet it is stored in the `temp` array, which 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 state then it switches back to personalised state after successful update (`setAppletState`).

Now let us see how the code for authenticateUser command is constructed. This command is the only one that is allowed to be sent in parts in multiple APDUs. There is no PIN check required nor input data integrity verification. The response is 20 bytes of SHA1 code calculated from the received message (we skip this part here as it is not relevant at this point). Here is the code:

```
/** @param apdu the incoming apdu packet 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 `sendResponse` method simply sends the data prepared in the `temp` array back to the host. The `readBigInput` method requires a bit more attention. Both `readBigInput` and `updateCommandState` make sure that the data parts contained in different APDUs are sent in proper order and are not interleaved by any other commands. This is done by using global applet variables and requiring the multiple APDUs 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 dispatch methods. The `readInput` method reads the input from the incoming APDU into the `temp` array in a standard way reporting any possible data length mismatches by throwing an appropriate exception which in turn causes a status word indicating an error condition to be sent back to the host. The `sendResponse` method also works in the standard way.

The more interesting methods are `setAppletState`, `updateCommandState` and `readBigInput`. The first one is responsible for setting and changing the applet's life state. It is the calling method's responsibility to ensure that a proper condition for changing this state is satisfied (e.g. that master PIN is 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;
        break;
    // ...
  }
}
```

Finally we get to the updateCommandState and readBigInput methods that share some global applet variables to ensure that the protocol is followed. One of them is multiple_package which indicates whether a multiple APDU command is being processed—when equal to 0 there is no multiple APDU command process in progress, when greater than 0 it is equal to the code of the multiple command being processed. The updateCommandState method first checks if the life 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 and if so if the current command belongs to the sequence of currently processed multiple APDUs throwing an exception 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 invoked out of the allowed sequence:

```
/** @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 APDUs to come, the p1 header byte indicates which APDU packet is being received ("p1-th out of p2 packets"). The global variables multiple_curr and multiple_total are used to control this. Whenever a multiple APDU packet 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 APDUs.

All the code presented here was engineered "by hand", however as we mentioned earlier we want to use the support of a CASE tool to do part of the job for us. The possibilities here that we see are the following:

- Having methods like `readInput`, `readBigInput`, `verifyInput`, etc. among the standard set of JAVA CARD helper methods, idioms and design patterns, together with the specifications (see the following subsection). This can be easily done in a tool like Together ControlCenter.
- Generating (possibly with a little of developers help) the code for `setAppletState` and `updateCommandState` methods from the state chart diagrams like the ones presented here also incorporating formal specifications for verification. Again this should be implementable in the tool of our choice (e.g. in Together ControlCenter there are ready tools to create code from sequence diagrams and vice versa).
- The PIN check routines 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, there is one APDU command that verifies the delivered PIN, sets the validation flag of the PIN object accordingly for the current terminal session and returns the result of PIN verification back to the host also indicating the number of tries left in case of failure. Then any command requiring PIN authentication can refer to PIN object by a single method call.
- Generating the skeleton code for command dispatching out of the definitions (tables) we showed earlier. As the code we presented matches the definitions nicely it should not be a difficult task, provided that methods like `readInput` or `verifyInput` are already available.

4.5 Formal specification

It is almost clear that the presented dispatching methods follow the semi formal specifications we gave earlier. The `setAppletState`, `updateCommandState` and `readBigInput` and possibly `readInput` methods require a bit more attention and this is were we turn to formal specification.

First we can define the state chart behaviour more formally by giving Object Constraint Language (OCL, part of UML) specifications like the following. Those specifications don't reflect the whole diagram set that we have shown, they are just examples. First we can tie a given applet life state to a condition that causes the applet to be in a given state, e.g.:

```
context Safe_Applet
inv: (self.userPIN.getTriesRemaining() = 0) =
       (self.curr_applet_state = AS_LOCKED)
inv: (self.masterPIN.getTriesRemining() = 0) =
       (self.curr_applet_state = AS_DEAD)
```

Next we can limit a set of possible command states in a given life state by the following expression:

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

Finally we can describe some of the behaviour of `setAppletState` and `updateCommandState` with the following expressions:

```
context Safe_Applet::setAppletState(command : byte, newstate : short)
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

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

As mentioned before, such specifications follow exactly the diagrams and it should be possible to just generate them automatically, possibly with a little bit of user intervention.

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

```
context Safe_Applet
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 APDUs. The next are two preconditions that make sure the read methods don't exceed the temporary array space they operate on:

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

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

Such specifications should be associated with a pattern that produces our read methods and put into design automatically together with the actual code.

Of course one may want to give some more in-depth specifications of the application describing its functionality or some safety properties. In the next subsection we show how the already existing KeY tool features can be used to produce such a specification, here we briefly discuss another situation where formal specification can prove itself helpful. Suppose we would like to extend our application to keep track of unsuccessful authentication attempts and disable the access once a certain number of unsuccessful attempts has been reached (similarly to PIN code verification). This is pretty 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 get increased during rejected attempts as well. After reaching the maximal value for a data type used (say `byte`) it will leap back to 0 ending up in an undesired, security breaching state. 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, e.g. by giving the master PIN. To verify such a property one needs a proper formalisation of JAVA integer arithmetics. The KeY system both supports the specification idioms and contains formalisation of JAVA integer arithmetics as part of the KeY specification library.

4.6 Employing the KeY system

Recall that one of the problems we found in `Safe_Applet` was that a single user ID can be registered more than once in the applet. Let's first look at the class representing a single user record in the applet:

```
public class KeyRecord {
  boolean empty = true;
  byte userID = 0;
  RSAKeyPair key;
}
```

Given this we would like to specify that there shouldn't 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 as Figure 5 shows. After the pattern is applied the following invariant is produced for `KeyRecord` class:

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

After a small modification we get what we want:

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

5 Future work

The specifications we have shown are subject to formal verification. The specification that describes the behaviour of the state machine controlling the protocol should be easily verified using a tool like Extended Static Checker for Java [8].

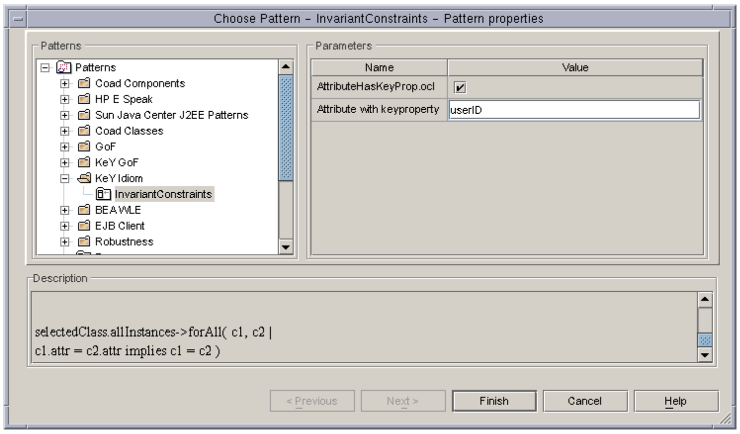

Figure 5: Applying specification patterns in the KeY system

Some of the other specifications (like the `multiple` variables invariants) require a more powerful tool like the KeY system. We plan to experiment with different verification tools to confirm the above claims. To successfully verify any Java Card application one needs formal specification of parts of the Java Card API used by a given application. There is work done in this direction [15, 13], however it needs some adjustments to fit into our framework.

One thing that we did not specify is that the applet data stays consistent in case when an applet's execution terminates abnormally by ripping out the card from the reader. This would require to specify a kind of invariant for our program that holds at any point of execution of the program, not only before and after the program is executed. This is not possible to express in OCL, neither is it possible to express in the KeY system's Dynamic Logic for Java, however, this problem has been solved for pure Dynamic Logic [4] and we plan to tackle this problem for Java Card in the future.

Finally to provide support for the Java Card design process we will develop a set of Java Card specific idioms and design patterns with specifications wherever possible. Such patterns should be among the set of standard Java Card development tools.

6 Conclusions

We presented an approach to rigorous development of Java Card applications. We have shown how UML can be used to specify an applet's behaviour and how such specifications can be translated into actual code. We have also presented how we can support formal specification and verification in Java Card development. A modern CASE tool plays an important role in our approach giving support for UML specifications, software patterns, formal verification (KeY system) and last but not least easy testing of Java Card applets (we have developed a Together ControlCenter plug-in supporting this— `http://www.cs.chalmers.se/~woj/javacard/`). Most of the code we have shown was developed by hand, but we were precisely following the UML diagrams we constructed, 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. However, as we already mentioned, designing the application in UML requires some expertise in a given domain and is a bit more lengthy process.

References

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. Proceedings of FASE at ETAPS 02, to appear.

[2] T. Baar, R. Hähnle, T. Sattler, and P. 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, Sept. 2000.

[3] G. Barthe, S. Sousa, G. Dufay, and M. Huisman. Jakarta: a toolset for reasoning about JavaCard. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*. Springer-Verlag, September 2001.

[4] B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.

[5] D. Bolignano, D. Le Métayer, and C. Loiseaux. Formal Methods in Practice: the Missing Link. A Perspective from the Security Area. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, editors, *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19–23, 2000*, volume 2067 of *LNCS*. Springer-Verlag, 2001.

[6] D. Bolten. PAM authentication with an iButton. `http://www-users.rwth--aachen.de/dierk.bolten/pam_ibutton.html`.

[7] Z. Chen. *Java Card Technology for Smart Cards*. Addison Wesley, 2000.

[8] ESCJava homepage. `http://www.research.compaq.com/SRC/esc/`.

[9] B. Jacobs, H. Meijer, and E. Poll. VerifiCard: A european project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001.

[10] J. Jürjens. Developing secure systems with UMLsec — from business processes to implementation. In A. P. Dirk Fox, Marit Köhntopp, editor, *Proc. Verlässliche IT-Systeme 2001 — Sicherheit in komplexen IT-Infrastrukturen, Kiel, Germany*. Vieweg Verlag, 2001.

[11] KeY project homepage. `http://i12www.ira.uka.de/~projekt/`.

[12] H. Martin and L. du Bousquet. Tools for automated conformance testing of JavaCard applets. Technical report, Gemplus, September 2000.

[13] H. Meijer and E. Poll. Towards a full formal specification of the Java Card API. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*. Springer-Verlag, September 2001.

[14] OpenCard homepage. `http://www.opencard.org/`.

[15] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. CSI Report CSI-R0005, Computing Science Department, Nijmegen, Mar. 2000.

[16] VerifiCard project homepage. `http://verificard.org/`.