Verifying the Mondex Case Study
The KeY Approach

Technical Report 2007-4

Isabel Tonin
tonin@ira.uka.de

University of Karlsruhe
Institute for Theoretical Computer Science
Am Fasanengarten 5
D-76131 Karlsruhe, Germany
`http://key-project.org`

July 13, 2007

Abstract

The Mondex Case study is still the most substantial contribution to the Grand Challenge repository. It has been the target of a number of formal verification efforts. Those efforts concentrated on correctness proofs for refinement steps of the specification in various specification formalisms using different verification tools. Here, the results of full functional verification of a **JavaCard** implementation of the case study is reported. The functional behavior of the application as well as the security properties to be proven were formalized in **JML** and verified using the **KeY** tool, a verification tool for deductive verifying **JavaCard** code. The implementation developed followed, as closely as possible, the concrete layer of the case study's original **Z** specification. The result demonstrates that, with an appropriate specification language and verification tool, it is possible to bridge the gap between specification and implementation ensuring a fully verified result. The complete material—source code, proofs and binaries of the verification system as well this report—is available at `http://www.key-project.org/case_studies/mondex.html`.

key words: Formal Methods, Specification, Program Verification, Deductive Verification, Design by Contract, **JavaCard**, **JML**, **KeY** Tool

Contents

Chapter 1

Introduction

As computer systems pervade society ever more deeply, ever higher demands will be made on their design, maintenance, and certification as greater reliance is placed on the delivery of their services. This will be particularly true for high integrity systems, where failure can cause loss of life, environmental harm, or significant financial loss. Rigorous processes are already in use in industrial sectors where certification is required for their software, for instance, avionics. However, the demand for more robust systems is beginning to be felt in other sectors as well. The further development of a theoretical basis and methodology for system design to ensure dependable and safe operation is an objective (2007.3.3) for Embedded Systems Design in the Seventh Research Framework Program (FP7 — `http://cordis.europa.eu/fp7/home_en.html`). Another example is the European Commission's Intelligent Car Initiative (i2010) which is intended to promote the use of new technologies in order to make cars safer, cleaner, and more efficient (`http://europa.eu/scadplus/leg/en/lvb/l31103.htm`). The only way to reach the desired level of reliability is to use better tools, which leverage advanced analysis technology to statically verify critical aspects of system behavior.

Formal methods are a prime candidate for fulfilling these demands. The term formal methods is used to describe a wide variety of different techniques addressing different needs at different stages of system development. Formal methods are aimed at enhancing the quality of systems and, in Software Engineering specifically, they include specification, verification, and testing techniques used to enhance the quality of software development. Formal specifications techniques introduce a precise and unambiguous description of system properties, useful in eliminating misunderstandings and for further verification of the system. Formal analysis techniques can be used to verify that a system satisfies its specification, or to systematically seek for cases where it fails to do so. Although, software testing is perhaps the most frequently used quality assurance method, it is insufficient to provide the required level of assurance in complex systems. Instead of trying to provide a comprehensive check of a system, testing focuses on sampling the executions, according to some coverage criteria, and comparing the actual behavior with the behavior that is expected according to the specification. Formal methods can be used to examine all possible execution paths.

Formal methods can be used in almost every stage of a software development.

The earlier in the process an error is caught, the less repercussions it has had, and the cheaper is to fix it. Errors and inconsistencies in the specification itself are often the most costly to fix. Formal methods can even help to improve the quality of the system specification.

Unfortunately, the tractability of formal methods tends to diminish with the size of the object to be checked. It is still a common belief that a real application can not be verified formally, leaving formal verification to the scope of academic examples. However, there has been a tremendous improvement in formal methods technology and in the power of the hardware they run on.

At the same time, more powerful microprocessors has meant that substantially more functionality in devices are performed in software, resulting in more life-critical decisions being dependent on software. Ensuring that software runs correctly is becoming critical to the success of industries that make use of embedded systems. Only formal methods offer a solution to the problem of ensuring the correctness of complex software systems. This realization is the prime motivation for *The Verified Software Grand Challenge.*

1.1 The Grand Challenge[1]

The Verified Software Grand Challenge is an ambitious, international, long-term research program for achieving a substantial and useful body of code that has been formally verified to the highest standards of rigor and accuracy. It is the sixth of a series of Grand Challenges in Computing (GC6), launched by the UK Computing Research Committee (`http://www.ukcrc.org.uk/grand_challenges/index.cfm`). The program has three objectives.

- Establish a unified theory of program construction and analysis.
- Build a comprehensive and integrated suite of tools that support verification activities.
- Collect a repository of formal specifications and verified code: the Verified Software Repository, VSR or VSR-net (`http://vsr.sourceforge.net/`).

The challenge's long term goal is to ensure that science and practice converge and, in fifteen years, to have a well developed theory, a comprehensive and powerful suite of tools, and a compelling body of experimental evidence demonstrating that reliable software can be engineered cost-effectively using formal verification techniques.

1.2 The Mondex Case Study

The Mondex case study is the first technical work in the scope of the Grand Challenge. It was a one year project, started in January 2006, intended to demonstrate how different research groups can collaborate and populate the Repository. Thus far, five groups have present results.

[1]This and the next section are adapted from the IEEE Computer Society's article *First Steps in the Verified Software Grand Challenge*[20].

Mondex is an electronic purse hosted on a smart card developed about ten years ago. A consortium led by NatWest, a United Kingdom bank, developed the software following the high-assurance Information Technology Security (ITSec—`www.itsec.gov.uk`) Level E6 standard. The application is completely distributed: each purse must ensure the security of all its transactions locally without any central control.

Electronic purse software is a security critical application. To ensure the correctness as required by the ITSec Level E6 standard, formal methods were used during its development process. The original application protocol was specified using **Z** notation and all proofs were performed by hand.

A commercially sanitized version of the Mondex documentation is publicly available[19]. It contains the **Z** specification of the security properties, an abstract specification, an intermediate and a concrete level design, and the hand-written correctness proofs of security and conformance of each design level. This version is hereafter called the "original" description of the case study. The challenge in this initial case study was to investigate the degree of automation that can be achieved for correctness proofs.

1.3 Related Work

Several groups have already verified the Mondex case study working with different formalisms:

- Massachusetts Institute of Technology (MIT) with the Alloy specifications verified with Alloy Analyzer model finder,
- Escher Technologies using Perfect Developer,
- United Nations University (Macao) and Technical University of Denmark with RSL specifications verified with PVS theorem prover,
- University of York using the original **Z** specification verified with Z/Eves theorem prover, and
- University of Augsburg using abstract state machines verified with KIV.

Some groups could use the original specification, or its translation to some similar specification language, while other groups adapted the original **Z** specification to another formalism. In general, the experience was successful. The proofs, after tuning the specifications or the tool, could be performed mostly automatically. The time consumed to finish the task was also surprising: some groups claimed that the work could be performed within one or two months. More information can be found at the VSR web page.

In addition, the results of a similar case study refining an abstract security protocol description down to a concrete implementation in **JavaCard**, can be found in the paper *Implementing a Formally Verifiable Security Protocol in **JavaCard*** [10].

Even though the one year challenge for verifying the Mondex case study is over, there are still many research groups interested in it. Some of the reasons for this are easy to see. The case study is a critical application, subject to ITSec level E6, the highest specified by the UK ITSec certification body. It is intrinsically small, since it has to run on a smart card, and is therefore suitable for use as a formal verification benchmark. In addition, it is complex enough, and therefore representative, to test and evaluate the capabilities of a verification tool. More importantly, there are still properties not yet verified by any of the published approaches.

1.4 The Goal

Up until now, Mondex verification approaches have concentrated on the specification and its refinement. Apparently. there are also ongoing attempts to automatically generate code for this case study (Escher Technologies) or verifying code (University of Augsburg) but no results have been published thus far.

The work presented here aims to demonstrate that deductive formal verification can not only validate a given specification but also the correctness of its concrete implementation in **JavaCard**. In end effect, it will be shown that with an appropriate specification language and verification tool, it is possible to bridge the gap between specification and implementation ensuring a fully verified result.

Chapter 2

Methodology

The case study described here demonstrates that the verification of the functional behavior of a real **JavaCard** application can be done using the **KeY** tool. For its successful completion, many steps were performed. The original **Z** specification was reformulated in **JML**, a modeling language tailored for **Java** applications that supports the *Design-by-contract* paradigm. Then using this specification, high level properties were verified. Since the application is to be hosted by a smart card, the specification was implemented in the **Java** variant **JavaCard**. Finally, the functional correctness of the implementation was proven using the **KeY** tool.

2.1 Design by Contract

Design by contract (DBC) is at the heart of the methodology used. Similar concepts have been around since the 1960s, but the actual name was introduced by Bertrand Meyer [14] in the early nineties. Since then it has gain wide acceptance in software engineering.

DBC defines associations between software components as formal agreements between "clients" and "suppliers" that are defined in terms of rights and obligations, or "contracts" (using a conceptual metaphor to a legal contract). Following this paradigm, a software designer creates a precise functional specification of an application by defining a contract for each module. The execution of a program can then be seen as an interaction between the modules as bound by their contracts.

The contract of an operation is essentially expressed in terms of a precondition and a postcondition. A client (i.e. a software module that uses the operation) must ensure the requirements stated by the precondition when calling the operation. In return, the operation guarantees delivering a result which corresponds to the constraints of the postcondition. As preconditions and postconditions are usually expressed in some logical notation, this issue can be restated more precisely: a client must ensure, when calling an operation, to be in a state that satisfies that operation's precondition for which the operation guarantees to return in a state which implies its postcondition.

Further elements of an operation's contract may include special assertions in case of error or exception, properties concerning side effects, or accounts for its complexity measure and memory consumption. In addition to its operations' contracts, the contract of a module (such as a class) may specify invariants and history constraints. Invariants express conditions that must "always" hold during execution; whereas, history constraints describe how values change. An example of an invariant would be the assertion that a numerical variable's value is not negative. A history constraint could express that its value increases monotonically.

The exact definition of the set of states that must satisfy an invariant or history constraints varies in different DBC languages and tools. By "must always hold," one could understand that an invariant should hold in all systems states, which might be far too strong to fulfill. This is called *strong invariant* in the book *Verification of Object-Oriented Software, The Key Approach*[3] (hereafter referred to as "the **KeY** book"). One could also understand invariants as only holding in all *visible states* of a system, i.e., before and after the execution of each non private method in the class.

Just like other specifications, contracts represent an abstract view of a software system, leaving details of the implementation hidden. The notion of a contract implies an easy means of "blame assignment." A precondition violation indicates that the client did not fulfill its part of the contract and is therefore erroneous. In turn, a violation of a postcondition must be the supplier's fault, i.e. a defect in the implementation of the corresponding operation. An accurate monitoring of a program's contracts during execution can thus greatly facilitate error detection. DBC is repeatedly mention in the OOTiA[1] Handbook as the suggested methodology for object oriented software construction.

2.2 The Java Modeling Language[2]

The *Java Modeling Language* (**JML**) is designed as a DBC style specification language for **Java**. The development of **JML** is a cooperative effort of several researchers, coordinated by Gary T. Leavens' group at Iowa State University. The group also distributes a set of tools for using **JML**, including a syntax checker and a **JML**-enabled **Java** compiler [4].

JML's basic use is the formal specification of the behavior of Java program modules (i.e., classes and interfaces). **JML** is currently limited to sequential specification. The coupling to **Java** enables **JML** to precisely reflect the semantics of **Java**, making it a specification language particularly close to the implementation. The main benefits in using **JML** are the precise, unambiguous specification of the behavior of **Java** program modules and documentation of Java code, as well as the ease of providing tool support.

JML provides a means of specifying both the interface and the behavior of **Java** modules. The interface of the method is the information needed to use it from other modules. It includes the name of the method, its modifiers (including its visibility and whether it is final) its number of arguments, its return type, what exceptions it may throw, and so on. The behavior of a method describes

[1] *Object Oriented Technology in Aviation*, a collection of recommended coding techniques and certification practices, initiated by US Federal Aviation Administration (FAA) and National Aeronautics and Space Administration (NASA), created to address issues regarding the use of object-oriented technologies in avionics and to prepare the next version of the requirements specification (RTCA/DO-178C, planned to be released in 2008).

[2] This section was adapted from the **JML** reference manual[12].

a set of state transformations that it can perform. A behavior of a method is specified by describing

- a set of states in which calling the method is define,
- a set of locations that the method is allowed to assign to (and hence change), and
- the relations between the calling state and the state in which it either returns normally, throws an exception, or for which it might not return to the caller.

The states for which the method is defined are formally described by a logical assertion, called the method's precondition. The allowed relationships between these states and the states that may result from normal return are formally described by another logical assertion called the method's normal postcondition. Similarly the relationships between these pre-states and the states that may result from throwing an exception are described by the method's exceptional postcondition.

In **JML**, one has the possibility of specifying that a method will not throw an exception, that it may throw an exception, or that it must throw an exception. In addition, class invariants and history constraints can also be specified in **JML**. Invariants are predicates that must hold in every *visible state* of an object. Each method in a class must preserve the class's invariants, i.e., the invariants should hold before and after its execution. Private methods or constructors can be excluded from the need to preserve invariants by declaring them as *helper* methods. History constraints, or constraints for short, are related to invariants. But whereas invariants are predicates that should hold in all visible states, history constraints are relationships that should hold for the combination of each visible state and any visible state that occurs later in the program's execution. History constraints can therefore be used to constrain the way that values change over time.

2.3 Deductive Program Verification[11]

Deductive Program Verification is a formal method for statically proving the correctness of a program with respect to the specification of its requirements. This technique is best suited for verifying functional properties of a system, i.e., determining whether the values computed by the system agree with its functional specification or not. Attempts to extend this technique for verifying temporal properties for arbitrary models, i.e., to include infinite state models, are still at the level of academic research projects.

In order to statically analyze a program, a deductive program verifier symbolically executes it, faithfully reflecting the semantics of the programming language. Mathematical logic and automated theorem proving techniques are used for this process. The correctness of a program p is defined by means of logical formulas ϕ and ψ that express constraints on the pre-state and the post-state of p. A formula $\phi \rightarrow \langle p \rangle \psi$ is valid if for every state s satisfying precondition ϕ a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds. This formula is similar to a *Hoare Triple* $\{\phi\}p\{\psi\}$ introduced by C.A.R. Hoare in 1969 [9] with the difference that in the former formula

ϕ and ψ might contain programs while Hoare triples can only be expressed in terms of pure first-order logic. Thus, Hoare logic is usually not sufficient to express ϕ and ψ for real world computer programs—a special program logic is necessary, such as *Java Dynamic Logic* (**JavaDL**)[2] an instance of *dynamic logic*[8] used in the **KeY** project (`http://key-project.org/`).

The **KeY** tool[1], or simply **KeY**, is a deductive program verification tool jointly developed at Karlsruhe University (Karlsruhe, Germany), Chalmers University (Gothenburg, Sweden) and the University of Koblenz (Koblenz, Germany) in the context of the **KeY** project. The aim of the project is to integrate formal software specification and verification into the industrial software engineering processes. The tool uses deductive verification of an annotated **Java** code in order to statically prove its correctness with regards to its specifications. As any analysis tool, **KeY** requires the properties to be analyzed to be formally specified. In **KeY**, functional properties specification has a form of a *contract* as described in the *design by contract* paradigm.

The main component of the **KeY** tool is the **KeY** prover, a semiautomatic prover with a comfortable graphical user interface. Special care has been taken to make user interaction, when it is necessary, as easy as possible, e.g., through flexible selection of proof strategies, the possibility to save and reload (partial) proofs, to rerun proofs on changed proof obligations, built in decision procedures for integers, drag and drop interaction, and a clearly laid out display of the proof. **KeY** is the only verification system that keeps the source code visible and as an integral part of the proof effort. (Other systems translate program code into some higher-order logic representation.) **JML** (or OCL[6]) specifications are automatically translated into proof obligations in **JavaDL**. The **JavaDL** calculus covers 100% of **JavaCard**, version 2.1, and it is continually being extended in order to support full sequential **Java** features (it actually supports most of them).

2.4 Java Card Technology[3]

JavaCard technology adapts the **Java** platform for use on smart cards and other devices whose environments are highly specialized, and whose memory and processing constraints are typically more severe than those of J2ME devices. A smart card is a plastic card that contains an embedded integrated circuit. A smart card resembles a credit card. When used as a SIM (Subscriber Identity Module) card, the plastic card is small—just big enough to fit inside a cellphone. Smart cards are highly secure by design, and tampering with one results in the destruction of the information it contains. Smart card technology is an industry standard defined and controlled by the Joint Technical Committee 1 (JTC1) of the International Standards Organization (ISO) and the International Electronic Committee (IEC). The series of international standards ISO/IEC 7816 defines various aspects of a smart card, including physical characteristics, physical contacts, electronic signals and transmission protocols, commands, security architecture, application identifiers, and common data elements.

A smart card does not contain a battery, and become active only when connected to a card reader. When connected, after performing a reset sequence,

[3]This section was adapted from the article *An Introduction to Java Card Technology - Part 1*[16].

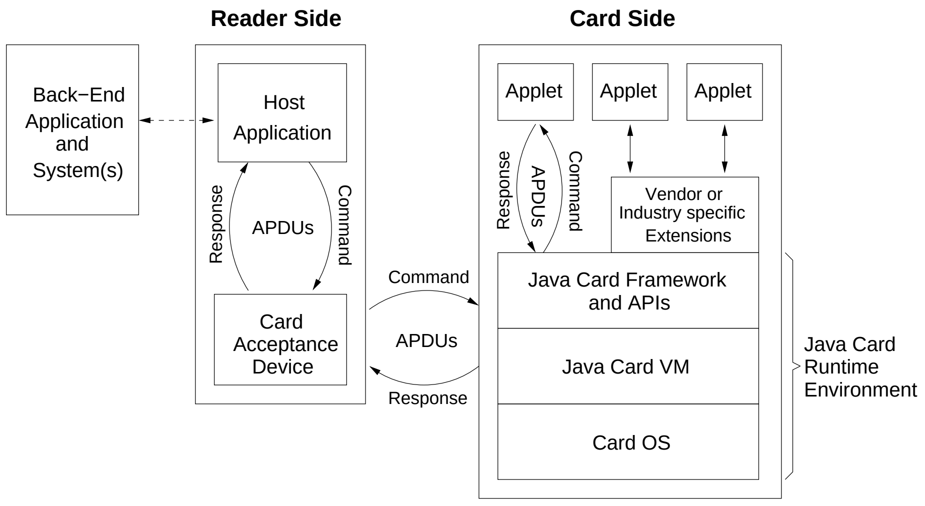

Figure 2.1: Architecture of a Java Card Application

the card remains passive, waiting to receive a command request from a external (host) application. Sun Microsystems defined a set of specifications for a subset of **Java** technology (with additional features) to create applications for smart cards—**JavaCard** applets. A device that supports these specifications is referred to as a **JavaCard** platform. On a **JavaCard** platform, multiple applications from different vendors can coexist securely. A complete **JavaCard** application consists of a back-end application and systems, a host (off-card) application, an interface device (card reader), and the on-card applet, user credentials, and supporting software. This architecture is shown in figure 2.1.

Back end applications provide services that support in-card Java applets. The host application resides on a desktop or a terminal such as a PC, an electronic payment terminal, a cellphone, or a security subsystem. The host application handles communication among the user, the Java Card applet, and the provider's back end application. The Card Acceptance Device (CAD) is the interface device that sits between the host application and the Java Card device. A CAD provides power to the card, as well as electrical or radiofrequency communication with it. A CAD may be a card reader attached to a desktop computer using a serial port, or it may be integrated into a terminal such as an electronic payment terminal at a restaurant or a gas station. The interface device forwards commands from the host application to the card, and forwards responses from the card to the host application. The **JavaCard** platform is a multiple application environment. As figure 2.1 illustrates, one or more **JavaCard** applets may reside on the card, along with supporting software—the card's operating system and the Java Card Runtime Environment (JCRE).

The message-passing model is the basis for all **JavaCard** communications. At its center is the Application Protocol Data Unit (APDU), a logical data packet that is exchanged between the CAD and the Java Card Framework. The **JavaCard** Framework receives and forwards to the appropriate applet any incoming command APDU sent by the CAD. The applet processes the command APDU, and returns a response APDU. The APDUs conform to the international

Command APDU						
Header (required)				Body (optional)		
CLA	INS	P1	P2	Lc	Data	Le

Figure 2.2: The Command APDU

Response APDU		
Body (optional)	Trailer (required)	
Data	SW1	SW2

Figure 2.3: The Response APDU

standards ISO/IEC 7816-3 and 7816-4.

The structure of a command APDU is controlled by the value of its first byte and in most cases is structured as shown in figure 2.2. A command APDU has a required header and an optional body, composed of the following.

- CLA (1 byte)—This required field identifies an application-specific class of instructions. Valid CLA values are defined in the ISO 7816-4 specification.
- INS (1 byte)—This required field indicates a specific instruction within the instruction class identified by the CLA field. The ISO 7816-4 standard specifies the basic instructions.
- P1 (1 byte)—This required field defines instruction parameter 1.
- P2 (1 byte)—This required field defines instruction parameter 2.
- Lc (1 byte)—This optional field is the number of bytes in the data field of the command.
- Data field (variable, Lc number of bytes)—This optional field holds the command data.
- Le (1 byte)—This optional field specifies the maximum number of bytes in the data field of the expected response.

The format of a response APDU is shown in figure 2.3. As with a command APDU, a response APDU has optional and required fields. The values of the status words are defined in the ISO 7816-4 specification.

- Data field (variable length, determined by Le in the command APDU)—This optional field contains the data returned by the applet.
- SW1 (1 byte)—This required field is the status word 1.
- SW2 (1 byte)—This required field is the status word 2.

A **JavaCard** applet extends the Applet API and must implement the install(), select(), process(), and deselect() methods. Each applet on a card is uniquely identified by an Application ID. The applet life cycle begins when the applet is downloaded onto the card and the JCRE invokes the applet's static Applet.install() method, and the applet registers itself with the JCRE by invoking Applet.register(). Once the applet is installed and registered, it is in the unselected state, available for selection and APDU processing. While in the unselected state, the applet is inactive. An applet gets selected for APDU processing when the host application asks the JCRE to select a specific applet in the card (by instructing the card reader to send a SELECT APDU or MANAGE CHANNEL APDU). To notify the applet that a host application has selected it, the JCRE calls its select() method; the applet typically performs appropriate initialization in preparation for APDU processing. Once selection is done, the JCRE passes incoming APDU commands to the applet for processing by invoking its process() method. The JCRE catches any exceptions the applet fails to catch. Applet deselection occurs when the host application tells the JCRE to select another applet. The JCRE notifies the active applet that it has been deselected by calling its deselect() method, which typically performs any necessary cleanup and returns the applet to the inactive, unselected state. A card applet behaves as a server and is passive. After a card is powered up, each applet remains inactive until be selected. The applet is active only when an APDU has been dispatched to it. Since Java Card specification 2.2.1, it is possible for multiple applets to be selected at the same time (i.e. cards have so called logical channels). Still only one applet is really active at a time, i.e., there is no concurrency among applets in the same card.

JavaCard distinguishes data stored in EEPROM and RAM, persistent and transient data respectively. Because power to a smart card can suddenly be interrupted, for instance, when the card is withdrawn from the CAD (also known as *card tear*) or when a mechanical or electrical failure occurs on the card, **JavaCard** provides a notion of transaction to ensure safe update of persistent objects. All operations executed between the beginning and end (or abort) of a transaction is guaranteed to be atomic. If a program error or a power reset occurs, the transaction mechanism will ensure that the state (values of fields) before the transaction is restored. When power is reapplied to the card after a power loss, the JCRE ensures that the transactions in progress when power was lost are aborted, and that all applet instances that were active when power was lost become implicitly deselected among other things.

A complete documentation, examples, and articles about the **JavaCard** technology can be found at the Sun MicroSystems' web page (`http://www.sun.com/`).

Chapter 3

The Mondex Purse

In its original description, the Mondex applications consists of a number of electronic purses that carry financial value, each hosted on a smart card. The purses interact with each other via communication devices to exchange value. Each purse has to ensure the security of all its transactions without recourse to a central controller.

In the original specification, three models were defined: an abstract, an intermediate, and a concrete model. The proofs were performed as refinements from one model to the next. Only the concrete model is relevant for this case study. Its behavior is defined as a non-atomic transaction protocol based on message exchange through an insecure communication media, called *ether*. In the protocol definition, security issues are considered: a purse can be disconnected at any time (card tear) and the ether is faulty. Thus a message can be lost by the ether, can be replayed several times, and can be read by any other purse. A message irrelevant to the protocol must be ignored. The relevant messages are assumed to be protected and unforgeable (they might be for instance encrypted, but this issue is not addressed in the specification). The protocol has to ensure that messages will be handled correctly and that the card keeps a valid state even being suddenly disconnected (i.e., the application must be *tear-safe*).

3.1 The Protocol Definition

The protocol is defined in five steps of message exchange. It is shown in figure 3.1[1]. At the beginning, two purses receive a start message from an unmodeled central authority. One purse is called the *fromPurse*, i.e. the purse that will send value, and the other is called the *toPurse*, i.e. the purse that will receive value. After that, messages are exchanged between the two purses to perform the transaction. The communication between the purses is done through the ether: a purse sends a message to the ether and a the other purse receives a message from the ether.

At the end of each step performed, the purse's status is updated and all control is done accordingly to the actual status. The status might have the

[1]This figure was adapted from the technical report Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method[17].

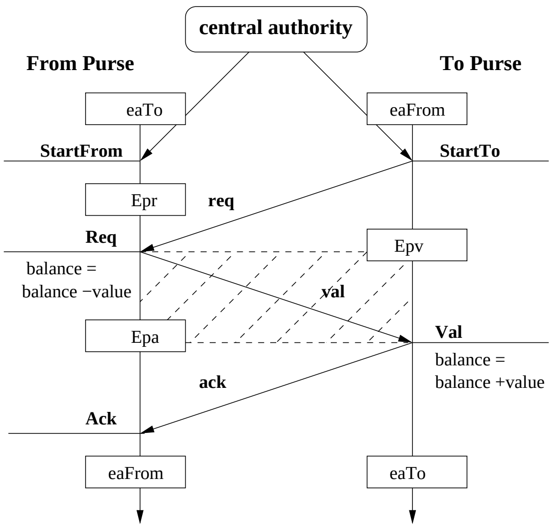

Figure 3.1: The Original Protocol Definition

following values:

- **eaTo**—expecting any to;
- **eafrom**—expecting any from;
- **epr**—expecting request;
- **epv**—expecting value; and
- **epa**—expecting acknowledge.

The operation to be performed by a purse depends on the message received.

- **StartFrom**—the fromPurse receives a "startFrom" message indicating the beginning of a transaction. This message contains the transaction counter part details, i.e., the data identifying the toPurse, and the transaction value. The purse increases its next transaction number field.
- **StartTo**—the toPurse receives a "startTo" message indicating the beginning of a transaction. This message contains the transaction counter part details, i.e., the data identifying the FromPurse, and the transaction value. The purse increases its next transaction number field and sends a "req" message.
- **Req**—the fromPurse receives a "req" message. It decreases its own balance by the transaction value and send a "val" message.
- **Val**—the toPurse receives a "val" message. It increases its own balance by the transaction value and send a "ack" message.
- **Ack**— the fromPurse receives a "ack" message.

At any time, the transaction can be interrupted. Money therefore can be lost in case it was already sent by the fromPurse (i.e. already decreased from its balance) and not received by the toPurse (i.e., not yet increased in its balance). The region in the protocol where money might be lost is shown highlighted with dashed lines in the figure 3.1. Since a purse can not know the status of its counter part, in order to prevent money lost, a purse in either status "Epv" or "Epa" logs the transaction details (called *PayDetails*). Nevertheless, money is considered lost only when both purses log the same transaction. The decision whether money is lost or not is considered to be done by an external entity. The logged transactions of each purse are transferred to an archive (a secure store of log records) and can then be compared. Thus, three extra operations are necessary to perform the protocol underlying control. These operations are not part of the protocol. Their scope is the current purse and therefore they do not refer to any counter part purse and do not cause any change in the purse's status or balance.

- **Abort**—for bringing an interrupted purse back to a valid initial state. It checks whether a transaction has to be logged and sets the purse's status to the initial "eato" or "eafrom."
- **readExLog**—for copying the logged transactions (the *Exceptional Log*) to the archive.
- **clearExLog**—for deleting the logged transactions from the purse's exceptional log. To enable this operation, the purse has to receive a valid *clear code* (an injective function generated from a set of logs stored in the archive provided by an external source).

The Mondex case study has been adapted from a larger, deployed banking application.

> *In order to produce a case study of a size appropriate for public presentation, much of the real functionality has had to be removed. ... This omitted functionality, whilst important from a business perspective, is peripheral to central security requirements.[19]*

For instance in the case study definition, nothing is mentioned about pin control or message encryption. However, although the purse defined for this case study is simpler than the full application, the removed functionality did not make the application less complex, only made it smaller.

The simplified purse has the following fields:

- **balance**—the purse's current balance;
- **exLog**—exceptional log for recording problematic transactions;
- **name**—the purse's identifier, unique among other purses;
- **nextSeqNo**—a transaction sequence number to be used for the next transaction (the combination of purse name and sequence number uniquely identify a transaction);
- **status**—indicating the purse's position in the current. transaction; and

- **pdAuth**—the payment details (*PayDetails*) of the current transaction. Payment details are defined as follows: the transfer details, i.e. the “from” and “to” purses names and value, and the sequence numbers of the transaction in the “from” and the “to” purses. Thus, a purse can only take part in at most one transaction: the one stored in this field.

3.2 The Security Properties

Security properties are properties of a system related to its behavior with respect to abrupt termination or data properties preservation (like consistency or confidentiality). In the original Mondex Purse certification report,[15] fifteen *Security Enforcing Functions*(SEF) are listed (chapter II, Specific Functionality). Five of these SEFs are part of the case study under the name *Security properties*. The discarded SEF are related to issues excluded from the case study such as currency, transfer permission, migration level, etc. The primary goal of this case study is to demonstrate that it is possible to prove that these properties are preserved by the implementation.

Security Property 1 (SP1): No value creation —no value may be created in the system. The sum of the balances of all purses does not increase.

Security Property 2.1 (SP2.1): All value accounted —all values must be accounted in the system. The sum of the balances and lost components of all purses does not change. A lost component is the transaction value when the transaction is logged by both purses.

Security Property 2.2 (SP2.2): Exception Logging —if a purse aborts a transfer at a point where value could be lost, then the purse logs the details.

Security Property 3 (SP3): Authentic purses —a transfer can only occur between authentic purses.

Security Property 4 (SP4): Sufficient Funds —a transfer can occur only if there are sufficient funds in the from purse.

Chapter 4

The Implementation

When implementing the Mondex application, the intention was to follow as much as possible the original **Z** specification. The implementation was restricted to the functionality mentioned in the specification. There are, thus, basic functionality missing in this implementation. For instance, when starting the transaction, each purse receives the counter part details with the "start" message. There are certainly some previous steps in the protocol where the host application requests the purse's details and checks the counter part purse's authenticity (accepting a purse or not for taking part in a transaction with that purse). However, since it is not mentioned in the definition how these steps are performed, they were not included in the implementation. In addition, some details had to be changed or specialized in order to fulfill implementation specific requirements. The code is presented in the appendix B.

The code was implemented in **JavaCard** following the explanation provided by Sun Microsystems in its *Wallet* example (included in the **JavaCard** development kit). The intention was also to keep the implementation as close as possible to a real purse application in order to preserve its complexity. Therefore, the application behavior was modeled following the **JavaCard** architecture. For instance, the APDU components, such as CLA, INS, SW, follow the ISO7816 standard, as recommended for a real application. On the other hand, some components of a real application not modeled in the case study, like pin code control and Application Identifier among others, were also omitted from the implementation.

4.1 A JavaCard Application

The class ConpurseJC (from *Concrete Purse*, as defined in the **Z** specification, implemented in **JavaCard**) implements the Mondex application. As any **JavaCard** implementation, it presents the following characteristics:

- the class ConpurseJC extends the abstract class Applet part of the **JavaCard** API;
- it implements practically all functionality (there is only one extra class, PayDetails (from payment details), to describe a transaction), not using any interface or inheritance;

- the only exceptions thrown by the ConpurseJC class's methods is the ISOException (no uncaught exception has to be caught by the JCRE);
- it does not create any object after its initialization (since most **JavaCard** implementations do not have garbage collector);
- it uses the transaction mechanism to ensure atomicity of the most critical operations.

In addition, all fields in this implementation had to be persistent, in order to present the expected behavior.

The implementation also preserves the security properties mentioned in the **KeY** book, in the chapter describing *The Demoney Case Study*, another electronic purse application analyzed with **KeY** (the specification of this properties can also be found in the technical report *Security Properties and Java Card Specificities To Be Studied in the SecSafe Project*[13]) .

- Only ISOExceptions are allowed at top level to avoid leaking the information about error conditions inside the applet. This property was formally proved to hold.
- All transactions are Well Formed: no transaction is started before aborting or committing a previous one; no transactions is aborted or committed without have being started; and no open transaction is left for the JCRE to close. This property was not proven but due to the code size, one can easily be convinced that it holds.
- All updates are atomic: all updates relevant for preserving the invariants are performed within a transaction mechanism, resulting in a *tear safe* implementation. The strong preservation of the invariants was proved using the **KeY** *throughout* configuration option.
- There is no unwanted overflow: integer operations should not overflow. The absence of overflow for all operations performed by the application was proved, using the **KeY** *Arithmetic semantics prohibiting overflow* configuration option.

4.2 Deviations from the Original Specification

Ideally one would be able to implement a software from its specification without having to adapt or extend any definition. For that, however, one has to be aware of this and specify an application focusing on its implementation. That was not the case when specifying the Mondex case study but even then, very few extra definitions had to be done when implementing it.

- A purse may not take part in a transaction when its exceptional log is full. This check is performed by the "StartFrom" and "StartTo" operations and, when full, an exception "Log Full" is sent and the starting command is rejected. Not allowing a purse to start a transaction is sufficient to prevent it from taking part in a transaction at all. The only way to avoid this exception is by clearing the exceptional log via the "clear_ex_log" operation.

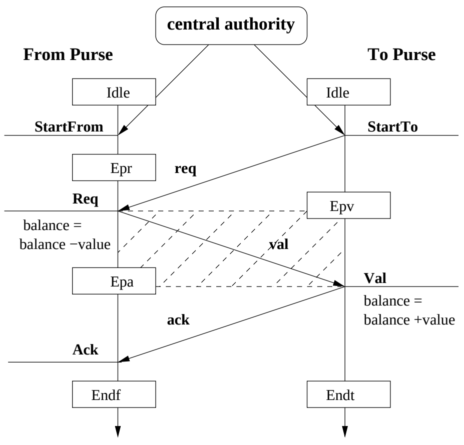

Figure 4.1: The Modified Protocol Statuses

- The value of a transfer has to be greater than zero. This change simplifies specifying whether a purse had already booked a value or not (when the value is zero, \old(value) is equal to value, resulting in considerably more complex specification).
- As already done by other groups, the purse's statuses before a transaction are both equal to "Idle." No differentiation between the purses statuses was considered necessary at this point of the protocol.
- The fromPurse's status and the toPurse's status at the end of a successful transaction are equal to "Endf" and "Endt" respectively. This change was necessary in order to differentiate the statuses after and before a transaction. The protocol annotated with the new statuses is shown in figure 4.1.
- The *Abort* operation is called by the select() method. For this case study, it was assumed that the application is selected before starting a new transaction. In this way, the system guarantees that any necessary log is done before the purse takes part in another transaction. The log operation can not be called at the beginning of the *start_from_operation* or the *start_to_operation*, as in the original definition, in order to guarantee that this operation can be executed only once within a transaction and that a purse's status does not become Idle (again) within the same transaction[1].

In the original specification, nothing is said about size and limits of the fields. Thus, the size of the fields and the limit of their values had to be defined as well:

- balance, name, and next sequence number were defined as *short* (0 — 32,767),

[1]Considering that the *start_from_operation* is called when the purse's status is equal to "Epa." According to this implementation, the received APDU is considered to be old and simply ignored. However, if the abort method is called instead, due to the purse's "Epa" status, the actual transaction would be logged, the purse's status would be set to "Idle," and the same transaction would be reinitiated.

- status and the exception log pointer as *byte* (0 — 127), and
- the exceptional log was defined as an array of the type *PayDetails* having a limited size of a tenth of the APDU buffer size[2]

Another implementation decision concerns the methods' behavior in face of erroneous APDUs: any APDU whose information does not match with the expected is considered faulty and is simply ignored. Thus, when a method receives a command APDU that requires a different status than the actual purse's status, or that the transaction details do not match with the expected, the command APDU is simply ignored and the method's execution is terminated (and no field is updated). Therefore, any duplicated APDU, is considered faulty and ignored by the application. In this way, the protocol prevents any erroneous behavior that could be caused by a failure in the message passing mechanism.

In addition, since the implementation was done in **JavaCard**, its architecture had to be taken into account. In a typical **JavaCard** application as shown in figure 2.1, there is an applet running in the card and, in the card reader, a host application controlling the card's applet execution. The applet only communicates with the host application (through the JCRE) via the APDUs (command and response). Therefore in fact, it is not possible for an applet running in a card to send a message directly to another applet running in another card. Hence, the message exchange happens between the two host applications (or one, if it controls both cards).

The case study, however, implements only the card part of the application, assuming that there is a host application communicating with the card applet that knows the Mondex application and behaves as follows.

- It is assumed that whenever a "fromPurse" host application starts a transaction (by sending a command APDU containing the instruction code equal to "StartFrom") there exists a counter part (to) purse's host application that also starts a transaction of its purse (by sending a command APDU containing the instruction code equal to "StartTo"), and vice verse.
- The host application of a purse interprets and reacts correctly to the messages received from host application communicating with the counter part purse, sending command APDUs to its own purse when required. More specifically,
  - when the host application receives a "Req" message, it sends a command APDU with the instruction byte equal to "Req;"
  - when the host application receives a "Val" message it sends a command APDU with the instruction byte equal to "Val;" and
  - when the host application receives a "Ack" message, it sends a command APDU with the instruction byte equal to "Ack."
- The host application of a purse understands and reacts correctly to the response APDUs received from its own purse, sending a message to the counter part purse host application when required. More specifically,

[2]Since each PayDetails element requires 10 bytes to be copied to the APDU buffer, the size of the exceptional log array should not exceed a tenth of the APDU buffer size.

- when the host application receives a response APDU indicating the successful end of the “startTo” operation, it sends a “Req” message;
- when the host application receives a response APDU indicating the successful end of the “req” operation, it sends a “Val” message; and
- when the host application receives a response APDU indicating the successful end of the “val” operation, it sends a “Ack” message.

Chapter 5

The Specification

The desired behavior of the Mondex application was specified using **JML**. In **JML**, the specification is included in the **Java** code as special comments. The written contracts and invariants were specified considering only the properties to be verified in this case study. Thus, there might be cases were the specification could have been more detailed.

5.1 Class Invariants and Constraints

Class invariants were used mainly to establish properties of the purse's fields and the relationship among them.

```
  /*@ invariant
1   @ (exLog != null) && (exLog.length > 0) &&
    @ (exLog.length < (APDU.BUFFER_LENGTH / 10)) &&
2   @ (logIdx >= 0) && (logIdx <= exLog.length) &&
3   @ (balance >= 0) && (balance <= ShortMaxValue) &&
4   @ (nextSeq >= 0) && (nextSeq <= ShortMaxValue) &&
5   @ (status >= 0) && (status <= 5) &&
6   @ (transaction != null) && (transaction.value > 0) &&
7   @ ((status == Epr) ==> (transaction.value <= balance)) &&
8   @ ((status == Epv) ==> (transaction.value <=
    @                       (ShortMaxValue - balance))) &&
9   @ (\forall byte i; i>=0 && i<exLog.length; exLog[i] != null);
    @*/
```

The meaning of this invariant can be understood most easily stepwise.

1. The exceptional log array is not null and its length is bigger than zero and smaller than a tenth of the APDU.buffer's size. This limit is necessary because each PayDetails object stored in the exceptional log array uses ten bytes when copied to the APDU.buffer in the readExLog operation.

2. The pointer for the next free position in the exLog array, logIdx, is bigger than or equal to zero and smaller than or equal to the exLog array length.

3. The balance is bigger than or equal to zero and smaller than or equal to the maximum value that a short variable can hold (32,767).

4. The next transaction sequence number is bigger than or equal to zero and smaller than or equal to the maximum value that a short variable can hold.

5. The status is bigger than or equal to zero and smaller than or equal to five (Idle = 0, Epr = 1, Epv = 2, Epa = 3, Endf = 4, Endt = 5).

6. The transaction is not null and that its value is bigger than zero.

7. When a purse's status is equal "Epr" (thus, a fromPurse) the transaction value is smaller than or equal to the balance. Therefore the purse has sufficient funds to perform the transfer.

8. When a purse's status is equal "Epv" (thus, a toPurse), the transaction value is smaller than or equal to the difference between the maximum value that a short variable can hold and the balance. Therefore, a purse can add the transaction value in its balance without causing overflow.

9. No position in the exceptional log is null.

History constraints, or constraints, in **JML** usually express relations between the old and actual value of a field. The old value is the value a field has before the method's execution and is expressed with the construct "\old." On the other hand, the actual value is the value of a field after the method's execution. For this case study, only two constraints were defined.

```
/*@ constraint
  @ ((\old(balance) != balance) ==>
  @                 ((\old(balance) - balance) == bookedValue()));
  @*/
```

This constraint states that whenever the purse's balance is updated, the value added or subtracted is equal to the value booked (defined in the section *Ensuring the Security Properties*5.3).

```
/*@ public constraint
  @ ((\old(logIdx) != logIdx) ==> (status == Idle);
  @*/
```

This constraint states that whenever the purse's exception log pointer is updated, the purse's status is equal to "Idle" after this operation. This invariant helps ensuring that a transaction can not be logged more than once (because a transaction can not be logged when the purse's status is equal to "Idle").

5.2 The Application Methods Contracts

A method contract is written to specify the expected behavior of a method. It is usually specified by means of its preconditions (which must be true for the method to exhibit the specified behavior), its postconditions (which must be true after the method's execution), what fields are updated within the method's body, and the exceptions that the method might throw. In **JML**, a method behavior can be specified as *normal*, *exceptional*, or just behavior. One uses normal behavior to specify the behavior of a method when it certainly terminates normally without throwing any exception; exceptional behavior to specify the behavior when it certainly terminates by throwing an exception; and simply behavior to specify the behavior of a method presents when it terminates normally or exceptionally. In this case, the exceptions that the method might throw can also be specified. This, in practice, is the most common way to specify a method's behavior (at least in **JavaCard**) because many exceptions are thrown under conditions that are hard or impossible to express (for example, I/O exceptions). For instance, in this case study, practically all methods perform operations that might thrown an exception under a condition that can not be predetermined when the method is called (like the APDU's method setIncomingAndreceive that throws an APDUException for I/O error). Thus, the simple*behavior* case was used to specify almost all contracts.

5.2.1 The Process Method

This method receives the APDU dispatched by the JCRE, checks the APDU header (to ensure that the APDU belongs to this application) and calls the correct operation to process the APDU (indicated by the APDU instruction byte). Its contract is the longest in the case study and one of the most interesting contracts of the case study.

This contract specifies the purse's behavior when processing APDU commands. It ensures that no transaction is logged during its execution (expressed via the update of the exception log pointer); that whenever the purse's status is not updated by an operation, its balance and sequence number for the next transaction fields are also not updated, ensuring that these fields can only be updated by protocol operations; and finally, that whenever the purse's status is updated, the purse's behaves according to the defined protocol (expressed in terms of the allowed updates in the purse's fields).

With this contract, since the defined protocol to be followed by each purse[1] can be derived from the called methods behavior, it is ensured that the defined protocol is correctly specified by the methods contracts. As a reminder, "\old(status)" refers to the status that the purse had before a method's execution, and "status" refers to the status presented by the purse after a method's execution.

```
/* @ public behavior
   @ requires apdu != null;
   @ assignable logIdx, balance, status, nextSeq, transaction.fromName,
   @    transaction.toName, transaction.fromSeq, transaction.toSeq,
   @    transaction.value,apdu._buffer[0..((logIdx*10) - 1)];
   @ ensures
   @ ((\old(logIdx) != logIdx) ==>
   @   ((logIdx == 0) && (status == Idle) && (\old(status) == Idle))) &&
   @ ((\old(status) == status) ==>
   @   (\old(balance) == balance) && (\old(nextSeq) == nextSeq)) &&
   @ ((\old(status) != status) ==> (
   @  (\old(apdu._buffer[ISO7816.OFFSET_INS]) == apdu._buffer[ISO7816.OFFSET_INS]) &&
   @  ((\old(status) == Idle) ==>
   @   ((((status == Epr)&&(apdu._buffer[ISO7816.OFFSET_INS] == StartFrom)) ||
```

[1]Without the relationship between the purse's statuses established by the message passing between the purses.

```
   @     ((status == Epv)&&(apdu._buffer[ISO7816.OFFSET_INS] == StartTo))) &&
   @     (\old(balance) == balance))) &&
 9 @  ((\old(status) == Epr) ==> ((status == Epa) &&
   @   (apdu._buffer[ISO7816.OFFSET_INS] == Req) && (\old(balance) > balance))) &&
10 @  ((\old(status) == Epv) ==> ((status == Endt) &&
   @   (apdu._buffer[ISO7816.OFFSET_INS] == Val) && (\old(balance) < balance))) &&
11 @  ((\old(status) == Epa) ==> ((status == Endf) &&
   @   (apdu._buffer[ISO7816.OFFSET_INS] == Ack) && (\old(balance) == balance))) &&
12 @  (status != Idle) &&
13 @  ((status == Epr) ==> (\old(status) == Idle)) &&
14 @  ((status == Epv) ==> (\old(status) == Idle)) &&
15 @  ((status == Epa) ==> (\old(status) == Epr)) &&
16 @  ((status == Endf) ==> (\old(status) == Epa)) &&
17 @  ((status == Endt) ==> (\old(status) == Epv)) &&
18 @  ((\old(balance) > balance) ==> ((status == Epa) &&
   @        ((balance - \old(balance)) == -transaction.value))) &&
19 @  ((\old(balance) < balance) ==> ((status == Endt) &&
   @        ((balance - \old(balance)) == transaction.value))) &&
20 @  ((\old(balance) == balance) ==> ((status == Idle) || (status == Epr) ||
   @        (status == Epv) || (status == Endf))) &&
21 @  (((status == Epr) || (status == Epv)) ==>
   @   ((nextSeq == (\old(nextSeq) + 1)) ||
   @   ((nextSeq == 0)  && (\old(nextSeq) >= ShortMaxValue)))) &&
22 @  (!((status == Epr) || (status == Epv)) ==>
   @   ((\old(nextSeq) == nextSeq) &&
   @    (\old(transaction.fromName) == transaction.fromName) &&
   @    (\old(transaction.toName) == transaction.toName) &&
   @    (\old(transaction.fromSeq) == transaction.fromSeq) &&
   @    (\old(transaction.toSeq) == transaction.toSeq) &&
   @    (\old(transaction.value) == transaction.value)))));
23 @ signals_only ISOException;
24 @ signals (ISOException e) (\old(balance) == balance) &&
   @  (\old(status) == status) && (\old(logIdx) == logIdx) &&
   @  (\old(nextSeq) == nextSeq);
   @*/
  public void process(APDU apdu)
```

1. In **JML**, a requires clause specifies a precondition of method. For this method the only requirement is that the received APDU is not null.

2. In **JML**, an assignable clause gives a frame axiom for a specification. It says that, from the client's point of view, only the locations named, and locations in the data groups associated with these locations, can be assigned to during the execution of the method. Since the process method is the caller of all other methods, any purse's field (excluding the exceptional log array) and the APDU buffer can be updated during its execution.

3. In **JML**, an ensures clause specifies a normal postcondition, i.e., a property that is guaranteed to hold at the end of the method (or constructor) invocation in the case that this method (or constructor) invocation returns without throwing an exception. In this specification, the method postcondition is stated from the number four to fourteenth. The meaning of each formula is described in the sequence.

4. If the exceptional log pointer logIdx is updated, then its new value is zero and the old and actual values of the purse's status are equal to zero. This means that, the exceptional log pointer withing the process method can only be updated by the clearExLog operation (and not by logging a

transaction, which is only done by the “abort_if_necessary” method, which is not called by the process method).

5. Whenever the purse’s status is not updated, neither the balance nor the transaction sequence fields is. This means that the operations that are not part of the protocol (abort, readExLog, and clearExLog) do not change the purse’s balance and sequence fields.
6. The next set of postconditions (7 to 22) must hold whenever the purse’s status is updated. They describe a purse’s local behavior when taking part in a transaction (according to the protocol).
7. The APDU’s instruction byte remains unchanged.
8. A purse’s status of “Idle” is updated either to “Epr,” when the APDU’s instruction byte is equal to “StartFrom,” or to “Epv,” when the APDU’s instruction byte is equal to “StartTo.” In both cases the purse’s balance remains unchanged. (This happens when executing the “startFrom” or “startTo” operation.)
9. A purse’s status of “Epr” is updated to “Epa” when the the APDU’s instruction byte is equal to “Req.” In this case, the purse’s balance is decreased. (This happens when executing the “req” operation.)
10. A purse’s status equal to “Epv” is updated to “Endt” when the APDU’s instruction byte is equal to “Val.” In this case, the purse’s balance is increased. (This happens when executing the “val” operation.)
11. A purse’s status equal to “Epa” is updated to “Endf” when the APDU’s instruction byte is equal to “Ack.” In this case, purse’s balance is not updated. (This happens when executing the “ack” operation.)
12. After starting a transaction, the purse’s status is not equal to “Idle.”
13. If the purse’s status is equal to “Epr,” the previous status was equal to “Idle.”
14. If the purse’s status is equal to “Epv,” the previous status was equal to “Idle.”
15. If the purse’s status is equal to “Epa,” the previous status was equal to “Epr.”
16. If the purse’s status is equal to “Endf,” the previous status was equal to “Epa.”
17. If the purse’s status is equal to “Endt,” the previous status was equal to “Epv.”
18. If the purse’s value is decreased, then the purse’s status is equal to “Epa” (the purse’s balance is only decreased by the “req” operation) and the decreased value is equal to minus the transaction’s value.
19. If the purse’s value is increased, then the purse’s status is equal to “Endt” (the purse’s balance is only increased by the “val” operation) and the increased value is equal to the transaction’s value.

20. If the purse's balance is not updated, then the purse's status are not equal to "Epa" or "Endt."

21. If the purse's status is equal to "Epr" or to "Epv," then the transaction sequence number was updated (increased by one or to 0, in case that the increased value would cause an overflow in the nextSeq field).

22. If the purse's status is equal to neither "Epr" nor "Epv," then the transaction sequence number and the transaction details are not updated. This means that the these fields are only updated by the "startFrom" and "startTo" operations.

23. In **JML**, a signals_only clause is an abbreviation for a signals-clause that specifies what exceptions may be thrown by a method, and thus, implicitly, what exceptions may not be thrown. Here, it states that only an ISOException is thrown by the "process" method (considered a good security property of a **JavaCard** applet, according to the section 14.5.2 Security Properties (of **JavaCard** applets) of the **KeY** book).

24. In **JML**, a signals clause specifies the exceptional or abnormal postcondition, i.e., the property that is guaranteed to hold at the end of a method (or constructor) invocation when the method (or constructor) invocation terminates abruptly by throwing a given exception. Here, it is stated that whenever an ISOException is thrown (the only that might occur), the purse's balance, status, logIdx, and nextSeq fields preserves their original values (i.e. are not updated).

5.2.2 The start_from_operation Method

The start_from_operation method is called to initiate a transaction in the "from" purse. Its behavior is defined as follows.

- The purse's exception log can not be full, in order to be able to store the actual transaction if necessary.
- The purse's status has to be equal to "Idle," i.e. the purse can not be taking part on any other transaction.
- The counter part details have to be consistent (method readCounterPartDetails): the counter part purse's name has to be different from the name of this purse and bigger than zero; and the counter part purse's sequence number for this transaction has to be bigger than or equal to zero.
- The value of the transaction has to be less than or equal to the purse's balance.
- The purse atomically increases the sequence number for the next transaction and sets the status to "Epr." If the sequence number can not be increased without causing an overflow, it is reset to zero.

These behavior is ensured by the method's postcondition, except the transaction value limit, which is already stated as a class invariant and thus has to be preserved by this method.

```
/*@ public behavior
1 @ requires apdu != null;
2 @ assignable transaction.fromName, transaction.toName,
  @     transaction.fromSeq, transaction.toSeq, transaction.value,
  @     nextSeq, status;
3 @ ensures
4 @ (\old(status)==Idle) && (status==Epr) && (logIdx < exLog.length) &&
5 @ ((nextSeq == \old(nextSeq) + 1) ||
  @      ((nextSeq == 0)  && (\old(nextSeq) >= ShortMaxValue))) &&
6 @ (transaction.fromSeq == \old(nextSeq)) &&
7 @ (transaction.fromName == name) && (transaction.toName != name) &&
8 @ (transaction.toName > 0) && (transaction.toSeq >= 0);
  @ signals_only ISOException;
  @ signals (ISOException e)
9 @ (\old(status) == status) && (\old(nextSeq) == nextSeq);
  @*/
  private void start_from_operation(APDU apdu) throws ISOException
```

1. The method only requires reception of a non null APDU as parameter.
2. It updates the actual transaction fields, and the purse's nextSeq and status fields.
3. The postconditions are stated from four to seven.
4. The status before the method's execution was equal to "Idle" and became "Epr." The exceptional log pointer is smaller than the exceptional log's length (and therefore the actual transaction can be logged if necessary. The method checks whether this is true before accepting the transaction and throws an ISOException indicating that the exceptional log if full and therefore the purse can not participate in any transaction before its exceptional log be cleared).
5. The next sequence number field, nextSeq, is increased by one or, in the case that its values has reached the maximum value it can store, is set to zero.
6. The sequence number for this purse in this transaction is equal to the purse's sequence number before the method execution.
7. This purse is the "from" purse, and the "from" and "to" purses of the transaction are not the same purse (they have different names).
8. The counter part purse's name is bigger than zero and the counter part purse's sequence number for this transaction bigger than or equal to zero;
9. If an exception is thrown, the transaction is not initiated: the purse's status and sequence number remain unaltered.

5.2.3 The start_to_operation Method

The start_to_operation method is called to initiate a transaction in the "to" purse. Its behavior is the same as the previous method with two differences.

- The value of the transaction may not cause an overflow when added to the purse's balance.
- The purse's status is set to "Epv" (instead of "Epr").

As before, the limit for the transaction value is not stated in the method's contract because it is part of the class invariant and has to be preserved by this method.

```
/*@ public behavior
1 @ requires apdu != null;
2 @ assignable transaction.fromName, transaction.toName,
  @     transaction.fromSeq, transaction.toSeq, transaction.value,
  @     nextSeq, status;
3 @ ensures
4 @ (\old(status)==Idle) && (status==Epv) && (logIdx < exLog.length) &&
5 @ ((nextSeq == \old(nextSeq) + 1) ||
  @      ((nextSeq == 0)  && (\old(nextSeq) >= ShortMaxValue))) &&
6 @ (transaction.toSeq == \old(nextSeq)) &&
7 @ (transaction.toName == name) && (transaction.fromName != name) &&
8 @ (transaction.fromName > 0) && (transaction.fromSeq >= 0);
  @ signals_only ISOException;
  @ signals (ISOException e)
9 @ (\old(status) == status) && (\old(nextSeq) == nextSeq;
  @*/
  private void start_to_operation(APDU apdu) throws ISOException
```

1. The method only requires reception of a non null APDU as parameter.
2. It updates the actual transaction fields, and the purse's nextSeq and status fields.
3. The postcondition are stated from four to seven.
4. The status before the method's execution was equal to "Idle" and became "Epv." The exceptional log pointer is smaller than the exceptional log's length (and therefore the actual transaction can be logged if necessary. The method checks whether this is true before accepting the transaction and throws an ISOException indicating that the exceptional log if full and therefore the purse can not participate in any transaction before its exceptional log be cleared).
5. The next sequence number field, nextSeq, is increased by one or, in the case that its values has reached the maximum value it can store, is set to zero.
6. The sequence number for this purse in this transaction is equal to the purse's sequence number before the method execution.
7. This purse is the "to" purse, and the "from" and "to" purses of the transaction are not the same purse (they have different names).
8. The counter part purse's name is bigger than zero and the counter part purse's sequence number for this transaction bigger than or equal to zero;
9. If an exception is thrown, the transaction is not initiated: the purse's status and sequence number remain unaltered.

5.2.4 The req_operation Method

The req_operation method implements the process that happens in the "from" purse when receiving a REQ message from the "to" purse.

- The purse's status has to be equal to "Epr."
- The transaction details sent with the APDU (counter part purse's name and sequence number, and transaction value) must match with the purse's ongoing transaction (method checkTransaction).
- The transaction's value is decreased from the purse's balance and its status is set to "Epa" atomically.

```
/*@ public behavior
1 @ requires apdu != null;
2 @ assignable balance, status;
  @ ensures
3 @ (balance == \old(balance) - transaction.value) &&
  @ (\old(status) == Epr) && (status == Epa);
  @ signals_only ISOException;
  @ signals (ISOException e)
4 @ ((balance == \old(balance)) && (status == \old(status)));
  @*/
  private void req_operation(APDU apdu) throws ISOException
```

1. The method only requires reception of a non null APDU as parameter.
2. It updates the purse's balance and status.
3. As postcondition, the actual balance is equal to its value before the method's execution minus the transaction value, and the status before the method's execution was equal to "Epr" and became "Epa."
4. If an exception is thrown, the purse's balance and status field remain unaltered.

5.2.5 The val_operation Method

The val_operation method implements the process that happens in the "to" purse when receiving a VAL message from the "from" purse.

- The purse's status has to be equal to "Epv."
- The transaction details sent with the APDU (counter part purse's name and sequence number, and transaction value) must match with the purse's ongoing transaction (method checkTransaction).
- The transaction's value is increased in the purse's balance and its status is set to "Endt" atomically.

```
/*@ public behavior
1 @ requires apdu != null;
2 @ assignable balance, status;
  @ ensures
```

```
3 @ (balance == \old(balance) + transaction.value) &&
  @ (\old(status) == Epv) && (status == Endt);
  @ signals_only ISOException;
  @ signals (ISOException e)
4 @ ((balance == \old(balance)) && (status == \old(status)));
  @*/
  private void val_operation(APDU apdu) throws ISOException
```

1. The method only requires reception of a non null APDU as parameter.
2. It updates the purse's balance and status.
3. As postcondition, the actual balance is equal to its value before the method's execution plus the transaction value, and the status before the method's execution was equal to "Epv" and became "Endt."
4. If an exception is thrown, the purse's balance and status field remain unaltered.

5.2.6 The ack_operation Method

The ack_operation method implements the process which happens in the "from" purse when receiving a ACK message from the "to" purse.

- The purse's status has to be equal to "Epa."
- The transaction details sent with the APDU (counter part purse's name and sequence number, and transaction value) must match with the purse's ongoing transaction (method checkTransaction).
- The purse's status is set to "Endf."

```
/*@ public behavior
1 @ requires apdu != null;
2 @ assignable status;
  @ ensures
3 @ (\old(status) == Epa) && (status == Endf);
  @ signals_only ISOException;
4 @ signals (ISOException e) (status == \old(status));
  @*/
  private void ack_operation(APDU apdu) throws ISOException
```

1. The method only requires reception of a non null APDU as parameter.
2. It only updates the purse's status.
3. As postcondition, the status before the method's execution was equal to "Epa" and became "Endf."
4. If an exception is thrown, the purse's status field remains unaltered.

5.2.7 The read_ex_log_operation Method

The read_ex_operation method is responsible for sending the purse's exceptional log to the archive. It can be called at any time, even during a transaction.

- The APDU's LC byte equal zero indicates that the number of transactions logged must be sent via the status word SW_RETURN_VALUE[2]. Here, this mechanism is required since the host application can not, a priory, determine the size of the expected data in the response APDU.
- When not zero, the APDU's LC byte must be the necessary size to store all logged transactions.
- The logged transactions (from the beginning of the exception log until the position immediately before the log's pointer) are copied to the APDU's buffer and send to the host application through a response APDU.

```
/*@ public behavior
1 @ requires apdu != null;
2 @ assignable apdu._buffer[0..((logIdx*10) - 1)];
3 @ ensures true;
  @ signals_only ISOException;
4 @ signals (ISOException e) true;
  @*/
  private void read_ex_log_operation(APDU apdu) throws ISOException
```

1. The method only requires reception of a non null APDU as parameter.
2. It updates the APDU's buffer, from its first position (0) until its $((logIdx * 10) - 1)$th position.
3. No postcondition is defined.
4. No postcondition is defined for the case when an exception is thrown.

In addition in order to very more easily this method, the loop that iterates on the exceptional log was also specified.

```
1 /*@ loop_invariant (logIdx >= i) && (i >= 0);
2   @ assignable i, buffer[0..(((logIdx - 1) * 10) -1)];
3   @ decreases (logIdx - i);
4   @*/
    while (i < logIdx)
```

1. A loop invariant is in practice a formula that describes the loop's behavior and must be true before, during and after the loop's execution. When proving with **KeY**, it is the only information that can be seen from outside the loop's body. Therefore, it has to be as complex as needed by the method's post condition. In this case, since the method's postcondition is trivial, the implicit loop bound defined in the the paper *Provably Correct Loops Bounds for Realtime Java Programs* ([11]) is sufficient. The implicit bound is defined as follows.

[2]An easy way to send a small value to the host application is to embed it in a status word and send it in an exception. This mechanism is explained in the paper *Javacard and Opencard Framework: a Tutorial*[7].

> For the case where the decrease clause contains only one loop dependent variable (a variable declared in the assignable clause of the loop), the implicit loop invariant seems to be sufficient for proving the method termination. The implicit loop bound describes the decrease clause definition and states that the decreases term is always positive and in every loop iteration less than the evaluation of the decreases term with the loop variable instantiated to its initial value.

In this case, the implicit loop bound is expressed by the formula $(logIdx - i) >= 0$ and $(logIdx - 0) >= (logIdx - i)$ which after simplification becomes $(logIdx >= i)$ and $(i >= 0)$.

2. The assignable clause of a loop is not part of the **JML**. It is an extension used by the **KeY** tool in order to simplify a loop verification. It contains the locations that might be assigned during the loop's execution.

3. The decreases term is the variant part of the loop specification. It specifies an expression that must not be less than 0 when the loop is executing, and must decrease by at least one (1) on each iteration of the loop (as stated in the last item).

5.2.8 The clear_ex_log_operation Method

The clear_ex_log_operation method is responsible for clearing the purse's exceptional log.

- The purse's status must be equal to "Idle."
- The received code must match the image calculated for the exceptional log (see image method).
- Since the relevant logged transactions are stored before the log's pointer (see method's read_ex_log_operation), for clearing the log it is sufficient to move its pointer to its initial position.

```
/*@ public behavior
1 @ requires apdu != null;
2 @ assignable logIdx;
3 @ ensures (logIdx == 0) && (status == Idle);
  @ signals_only ISOException;
4 @ signals (ISOException e) (logIdx == \old(logIdx));
  @*/
  private void clear_ex_log_operation(APDU apdu) throws ISOException
```

1. The method only requires reception of a non null APDU as parameter.
2. It only updates the purse's exceptional log pointer logIdx.
3. After the method's execution, the purse's logIdx field is equal to zero and the status is equal to "Idle" (and since the status is not modified by this method, its value before the method's execution is the same, i.e. (\old(status) == Idle) is also true).
4. If an exception is thrown, the purse's logIdx field remains unaltered.

5.2.9 The abort_if_necessary method

The abort_if_necessary method is called by the *select* method to ensure that the previous transaction, if interrupted, will be logged, and that the status of a purse is set to "Idle" at each time an applet is selected (consequently, for this case study, at the beginning of each transaction.

First, the method checks whether or not the previous transaction needs to be logged. This is the case when the purse's actual status is equal to "Epv" or "Epa." In this case, the previous transaction (still in the purse's *persistent* transaction field) has to be logged. Before logging the transaction, the method checks whether or not the exception log can store a new transaction or not. If not[3], an exception is thrown and the method's execution is terminated. Otherwise, the transaction is copied to the exception log at the log's pointer position, and the log's pointer is increased.

At the end, the purse's status is set to "Idle." The purse is then ready for a new transaction.

```
/*@ public behavior
1 @ requires true;
2 @ assignable exLog[\old(logIdx)], logIdx, status;
3 @ ensures
4 @ (status == Idle) &&
5 @ (((\old(status) == Epv) || (\old(status) == Epa)) ==>
  @  ((\old(logIdx) < exLog.length) && (logIdx == (\old(logIdx) + 1)) &&
  @   (exLog[\old(logIdx)] == transaction))) &&
6 @ (((\old(status) != Epv) && (\old(status) != Epa)) ==>
  @  ((logIdx == \old(logIdx)) && (\old(exLog[logIdx]) == exLog[logIdx])));
  @ signals_only ISOException;
7 @ signals (ISOException e)
  @ (\old(logIdx) == logIdx) &&(\old(status) == status);
  @*/
  private void abort_if_necessary() throws ISOException
```

1. There is no precondition defined for this method.

2. During the method's execution, the position pointed by the exceptional log pointer logIdx at the beginning of the method and logIdx itself might be assigned, and the purse's status is certainly assigned.

3. The method's postcondition is stated on lines four to six of the specification above.

4. The purse's status is equal to "Idle."

[3]Considering that (i)a transaction to be logged has to have been initiated, (ii)the exception log my not be full at the beginning of a transaction (because there is an explicit test at the beginning of the "StartFrom" and "startTo" operations), and (iii)the exception log is not updated by any operation between the beginning of a transaction and its logging, then one could guarantee that the exception log is not full whenever a transaction has to be logged. However, this property can not be easily stated using **JML**. One could think about including the statement (logIdx ¡ exLog.length) in the *process* method's postcondition and prove that it is true from the beginning to the end of a transaction. But it still would need to be stated that the abort transaction happens after that, and this can not be done using **JML** since the abort operation is performed out of the *process* method's scope.

5. If the purse's status at the beginning of the method's execution is equal to "Epv" or to "Epa" then (i) the exceptional log pointer logIdx at the beginning of the method's execution points to a position smaller than the length of this log (and the transaction can be logged without causing an overflow), (ii) the logIdx is incremented by one, and (iii) the value of the transaction field is copied to the exceptional log at the position pointed by logIdx at the beginning of the method's execution.

6. If the purse's status at the beginning of the method's execution is neither equal to "Epv" nor to "Epa" then neither logIdx nor the position of the exceptional log pointed by it are updated.

7. If an exception is thrown, the exception log pointer and the status remain unaltered. The exceptional log is not listed here because it might be updated in the case where the exception was caused by a transaction error. However, since the exceptional log pointer remains unaltered, this update has side effect in the application (because the updated position will be reused by the next logged transaction).

5.2.10 The image Method

The image method models a method that calculates the clear code for the exceptional log. Since it is not described in the original definition, it could not be fully implemented. Its contract was used in the verification of the *clear_ex_log_operation* method behavior, and its body in the *clear_ex_log_operation* method strong invariants preservation proof.

```
1 /*@ public normal_behavior
2   @ requires true;
3   @ assignable \nothing;
4   @ ensures true;
    @*/
    private short image();
```

1. In **JML**, normal_behavior means that the behavior specified terminates normally without throwing any exception. A normal_behavior specification case is just syntactic sugar for a behavior specification case with an implicit signals clause "signals (java.lang.Exception) false;."

2. No precondition is defined.

3. No field is updated by this method.

4. No postcondition is defined.

5.3 Ensuring the Security Properties

The contracts presented in the last section establish the expected behavior of each single method of the application. Nothing is said about the general *Security Properties* to be demonstrated. These properties are especially difficult to express because

- they are general to the application, therefore they do not belong to any particular method (described in subsection *Expressing General Properties*);
- they result from the protocol execution, hence, a sequence of steps, and this is particularly hard to express in **JML** (described in subsection *Expressing Sequence of Events*);
- *SP1* and a*SP2* (see section 3.2) are stated on the basis of all possible purses, while the the transaction protocol is defined for two purses (described in subsection *The World of Purses*);
- for their specification it is necessary to reference the interaction of the local purse with another purse, an object of same class being specified (described in subsection *Referencing the Counter Part Purse*); and
- it is not well defined "when" they should hold (described in subsection *Specifying when the Properties Hold*).

5.3.1 Expressing General Properties

To solve this problem one has to take into account that the security properties defined for the case study can be derived by the class invariants (plus the relationship between the statuses of the two purses during a transaction, as explained in the section 5.3.4). In **KeY**, the implication of these properties by the invariants could be stated as a Java DL formula. However, for this case study, the intention was to fully specify the application using **JML**, and therefore a solution to establish these properties had to be found.

The solution was to define a (dummy) method (*showProperties*) without a body whose only purpose was to enable writing the security properties as its postcondition. By proving the correctness of this contract, one ensures that, as long as the class invariants and the method's precondition are true, the security properties are also true.

5.3.2 Expressing Sequence of Events

The second problem is related with a more problematic issue to solve: causality, not easily expressible in **JML** [4]. The solution adopted here uses the **JML** *history constraints* mechanism. Even though it is not a full solution for expressing causality, one can express *one* change of state (the *old* and the *actual*) and therefore one can express one step of the protocol with it.

Since the *successor state* is a transitive function, it was possible to express in **JML** that the application implements the defined protocol for each purse: it was expressed as a post condition of the *process* method's (responsible for calling all the other methods). In this way, one can prove that the application implements the protocol to be followed by each purse correctly.

[4]There is work being done in this direction such as the **JML** extension proposed in the paper *Specifying and checking method call sequences of Java programs*[5] and the tool called *JAG*, a **JML** annotation generator for verifying temporal properties on Java classes(http://lifc.univ-fcomte.fr/~jag/). However, this work was not considered in this case study.)

Nevertheless, this does not completely solve the causal problem: one must establish that a purse having a status "Endf" already performed the *val* operation and hence already decreased the transaction value from its balance. One can easily see by the protocol definition that this is true. One can also see that following the specified "status" and "old status" in the *process* method's postcondition, this is certainly true. But one can not express something like "this happened in method A and, since method B is executed after method A ..." in **JML**. The solution adopted in this case study was to include helper methods to determine this information based on the purse's status. For this case, the method *bookedValue* was defined (see section 5.4).

5.3.3 The World of Purses

The third point brings the problem of how to express a property suppose to be hold by all objects of a class in the class definition, i.e., how to reference the effect of any transaction on all existent purses. In the case study's definition, the security properties *SP1* and *SP2* are establish for the *world of purses*, i.e., for all existent (authentic) purses. This brings immediately the idea of modeling the world of purses, which would make the specification more complicated. However, the approach used in the case study defines that

- a purse's name uniquely identifies a purse,
- a transaction number and name uniquely identifies a transaction,
- a transaction can happen only between two purses (guaranteed by the transaction details shared by both purses), and
- a purse can only take part in one transaction at the time.

In addition for this case study, the assumption was made that (according to the case study definition)

- the change in the sum of all individual purse's balance, intended to be verified is the change resulting from the transactions specified in the case study (not from a purse in isolation, or from some other possible transaction);
- all transactions present the same behavior (defined by the protocol to be verified);
- each transaction happens in isolation, and therefore its contribution to the change on the *world of purse's balance*, if any, is independent from any other transaction, and
- therefore the contribution of each transaction to the change of the *world of purse's balance* is equal to the change happened in the sum of the balance of the two purses in the transaction.

Therefore one can deduce that, by demonstrating that one transaction preserves the security properties related to changes on the *world of purse's balance*, one demonstrates that all transaction do as well. This deduction is assumed true in this case study[5] and therefore the security properties *SP1* and *SP2* are demonstrated to be hold by a transaction.

[5]There is ongoing work in the **KeY** group to prove this deduction.

5.3.4 Referencing the Counter Part Purse

Referencing the other purse is a non trivial issue. The implementation and the specification, as already stated, refer to the local purse. However, in order to specify the security properties *SP1* and *SP2* one has to express the interaction of the local purse with another purse, an object of the same class.

Hence, one has first to decide how to refer to the other purse. For this case study for referencing the counter part purse, the following assertion was done: either a purse is not taking part of any transaction (thus its status is equal to "Idle") or it is taking part in a transaction (therefore there does exist a counter part purse taking part of the same transaction). This was stated in **JML** as follows.

```
@ (status != Idle) ==>
@  (\exists ConPurseJC x;
@  x!= null && x.transaction == transaction && x.name != name;
```

In addition, one has to be aware of the other purse when writing the specification in order to make it sufficient to express the properties of both purses without being too verbose. This issue had to be carefully taken into account, for instance, when deciding how the protocol to be followed by a purse should be specified. If it has to be available to prove the security properties, it has to be expressed as a general class information, in this case as class constraints. Since this is not the case, it was expressed as a *process* method's postcondition. However, the specification of the relationship between the purse's statuses while performing the transaction is necessary for proving the security properties (because when considering the end of a transaction in one purse, it is necessary to know what are the possible status presented by the counter part purse). This relationship could have been expressed as a class invariant but, since it is not necessary for any other proof, it was stated as a precondition of the method *showProperties*. Considering "x" the counter part purse, this relationship was expressed in **JML** as follows.

```
1 @  ((status == Endf) ==> (x.status == Endt)) &&
2 @  ((status == Endt) ==> ((x.status == Epa) || (x.status == Endf))) &&
3 @  ((status == Epa) ==> ((x.status == Epv) || (x.status == Endt))) &&
4 @  ((status == Epv) ==> ((x.status == Idle) || (x.status == Epr) ||
  @                        (x.status == Epa))) &&
5 @  ((status == Epr) ==> ((x.status == Idle) || (x.status == Epv)))
```

This specification can be read as whenever the purse's status is equal to "Endf," the counter part purse's status has to be equal to "Endt;" and whenever the purse's status is equal to "Endt," the counter part purse's status has to be equal to "Epa" or "Endf;" and so on. These assumptions are not explicitly proven in this case study but they can be easily verified by the protocol definition in figure 4.1 and derived from the knowledge of the message passing protocol, as explained below.

1. When the purse's status is equal to "Endf," its previous status was equal to "Epa" (proven in the process method's postcondition). Considering its previous status, the purse received from the host application a command APDU containing the instruction byte equal to "Ack" (proven in the process method's postcondition). Since the host application sent this APDU,

it certainly received an "Ack" message from the counter part purse host application (according to the assumed behavior of the host application and considering the assumption that a message can not be forged). Since the counter part purse host application sent the "Ack" message, it received from its purse (the counter part purse) a response APDU indicating the successful end of the "Val" operation (according to the assumption on the host application behavior). Therefore, since the counter part purse executed the "Val" operation, its own status is equal to "Endt" (proven in the val_operation method's contract).

2. When the purse's status is equal to "Endt," its previous status was equal to "Epv" (proven in the process method's postcondition). Considering its previous status, the purse received from the host application a command APDU containing the instruction byte equal to "Val" (proven in the process method's postcondition). Since the host application sent this APDU, it certainly received an "Val" message from the counter part purse host application (according to the assumed behavior of the host application and considering the assumption that a message can not be forged). Since the counter part purse host application sent a "Val" message, it received from its purse a response APDU indicating the successful end of the "Req" operation (according to the assumption on the host application behavior). Therefore, since the counter part purse executed the "Req" operation, its own status is equal to "Epa" (proven in the val_operation method's contract).

   In addition, when the purse status is equal to "Endt" it successfully performed the "val_operation" method (one already knows that the purse received a command APDU containing the instruction byte equal to "Val." By code inspection, one can ensure that this operation is performed when the purse receives this APDU. Finally, the purse's final status is proven in the "val_operation" method's contract). Thus a response APDU indicating the successful end of this operation was sent to the host application, and therefore an "Ack" message was sent to the counter part purse's host application. Thus, it is possible, if the counter part purse had already received and processed this message, that the counter part purse's status is not "Epa" anymore rather "Endf."

3. When the purse's status is equal to "Epa," its previous status was equal to "Epr" (proven in the process method's postcondition). Considering its previous status, the purse received from the host application a command APDU containing the instruction byte equal to "Req" (proven in the process method's postcondition). Since the host application sent this APDU, it certainly received an "Req" message from the counter part purse host application (according to the assumed behavior of the host application and considering the assumption that a message can not be forged). Since the counter part purse host application sent a "Req" message, it received from its purse a response APDU indicating the successful end of the "StartTo" operation (according to the assumption on the host application behavior). Therefore, since the counter part purse executed the "StartTo" operation, its own status is equal to "Epv" (proven in the Start_to_operation method's contract).

In addition, if the purse status is equal to “Epa” it successfully performed the “req_operation” method (one already knows that the purse has received a command APDU containing the instruction byte equal to “Req.” By code inspection, one can ensure that this operation is performed when the purse receives this APDU. Finally, the purse’s final status is proven in the “req_operation” method’s contract). Thus a response APDU indicating the successful end of this operation was sent to the host application, and therefore an “Val” message was sent to the counter part purse’s host application. Thus, it is possible, if the counter part purse had already received and processed this message, that the counter part purse’s status is not “Epv” anymore rather “Endt.”

4. When the purse’s status is equal to “Epv,” its previous status was equal to “Idle” (proven in the process method’s postcondition). Considering its statuses, the purse received from the host application a command APDU with the instruction byte equal to “StartTo” (proven in the process method’s postcondition). Considering that it is the beginning of a new transaction, one can assume that the counter part purse’s host application sent a command APDU containing a instruction code equal to “StartFrom” to its purse (according to the assumed behavior of the host application). Since the counter part purse’s host application sent this APDU, either the counter part purse already processed it or not. If not, its status is still equal to “Idle;” and if it has processed, the “start_from_operation” method was executed (one already knows that the purse has received the a command APDU with the instruction byte equal to “StartFrom.” By code inspection, one can ensure that this operation is performed when the purse receives this APDU) an thus the counter part purse’s status is equal to “Epr.”

   In addition, since the purse status is equal to “Epv” it successfully performed the “start_to_operation” method (one already knows that the purse received a command APDU with the instruction byte equal to “StartTo.” By code inspection, one can ensure that this operation is performed when the purse receives this APDU. Finally, the purse’s final status is proven in the “start_to_operation” method’s contract). Thus a response APDU indicating the successful end of this operation was sent to the host application, and therefore an “Req” message was sent to the counter part purse’s host application. Thus, it is possible, if the counter part purse had already received and processed this message, that the counter part purse’s status is not “Epr” anymore rather “Epa.”

5. When the purse’s status is equal to “Epr,” its previous status was equal to “Idle” (proven in the process method’s postcondition). Considering its statuses, the purse received from the host application a command APDU with the instruction byte equal to “StartFrom” (proven in the process method’s postcondition). Considering that it is the beginning of a new transaction, one can assume that the counter part purse’s host application sent a command APDU containing a instruction code equal to “StartTo” to its purse (according to the assumed behavior of the host application). Since the counter part purse’s host application sent this APDU, either the counter part purse already processed it or not. If not, its status is still

equal to "Idle;" and if it has processed, the "start_to_operation" method was executed (one already knows that the purse has received a command APDU with the instruction byte equal to "StartTo." By code inspection, one can ensure that this operation is performed when the purse receives this APDU) an thus the counter part purse's status is equal to "Epv."

5.3.5 Specifying when the Properties Hold

Finally, the last point to consider is specifying when the security properties should hold. They are properties of the protocol and therefore should be preserved by the transaction execution. In principle, all security properties stated for this case study, except one, hold at all times during the transaction's execution (like strong invariant).

There is however, a problem defining when the *Security Property 2.1 All value accounted* should hold. There is a time during a normal execution of a transaction that the property does not hold: when the *fromPurse* already sent the value message (and subtracted the transaction value from its balance) but the *toPurse* did not yet receive it (and therefore did not yet add this value to its own balance). At this point, the sum of the *world of purse's balance* is smaller than at the beginning of the transaction. This problem is naturally solved within the protocol: either the *toPurse* receives the message and adds the value in its own balance, or it does not receive the message, because the transaction was interrupted ,and the transaction is logged when the purse is reinitialized (in this case the *fromPurse*'s transaction is also logged when it is reinitialized, and the lost element ensures that the property holds).

Certainly, the moment when the property does not hold has to be taken into account when formalizing the property. Perhaps the proposal for enabling a full protocol specification in **JML**, presented in the paper *Specifying and checking method call sequences of Java programs*[5][6], or something in a similar direction would solve this problem. However in this case study, a solution is proposed using standard **JML**.

The solution proposed here is to ensure that the property holds at the beginning and at the end of the transaction, i.e. assuming that the property holds before a transaction execution, the property holds when this execution is over. Considering the implementation, a transaction is processed from its beginning until its end (normal or interrupted). Then, sometime after, when the application is selected again, the *abort_if_necessary* method is executed in order to initialize a card (performing any necessary clean-up operation to ensure that the card is in a "Idle" state ready to perform any further action). In this operation depending on the purse's status, it is possible to know whether the previous transaction had to be logged or not. The security property holds when the last step referred to a transaction happens: after they have been logged or not, depending on the purse's status, at the initialization step.

There is another problem as well: the initialization step is local to a purse and is performed by each purse at different times. Nevertheless, one can determine that the security properties concerning a transaction "t" between two purses "p1" and "p2" hold when "p1" and "p2" have been initialize after executing the transaction "t."

[6]This proposal was not considered in this case study and this is the reason for the doubt whether it is a solution or not.

For that, the purse's statuses referred in the *showProperties* method's postcondition (for calculating the lost component) are the status that a purse had when initialization was performed (at the beginning of the *abort_if_necessary* method). One might find this definition confusing because, since the initialization happens at different time, a purse might be taking part in another transaction when the properties are verified. In fact, this does not make any difference because the properties are based on the status a purse had at initialization time (and not currently).

There is however another practically equivalent time during the transaction's execution when the security properties hold: at the end of a transaction's execution, where the *end of a transaction's execution* here means the point in the protocol where the transaction's execution terminates (normally or abruptly), i.e. when the last APDU referred to this transaction was received and processed and no further message referring to this transaction is sent to any of the purses. Thus, the purse's statuses referred in the *showProperties* method's postcondition (for calculating the lost component) are the last status reached by a purse within this transaction's execution. For instance, if a purse's status at the end of a transaction is equal to "Epa," it means that this purse stopped processing this transaction when its status was equal "Epa" (i.e., the transaction was abruptly terminated). If the purse's status is equal to "Endt," it means that the transaction was executed until the end of the protocol and then its execution was normally terminated. The problem with this definition is that the transaction is effectively terminated (and logged if necessary) only when the card is reinitialized. Nevertheless, one can consider that if at the end of a transaction's execution, the purse's status is equal to "Epv" or "Epa" the transaction value "will certainly" be logged (guaranteed by the *abort_operation* method's contract) and therefore can assume that the security properties hold at this point.

There is no decision here about what is the best definition. Any one can be used as an interpretation of the purse's statuses mentioned in the *showProperties* method's postcondition for calculating the lost component.

5.4 Auxiliary Methods

In order to make clearer the security properties specification two extra functions were defined. In principle, they could have been implemented as **JML** model methods but since the translation of model methods into **Java** DL is not fully implemented in **KeY**, they were declared as **Java** methods. In addition, they have to be side-effect-free methods (in **JML** stated trough the modifier "pure") in order to be used within an specification.

The *bookedValue* method determines the value already booked in the purse's balance in the actual transaction. It is determined based on the purse's status (according to the defined protocol), thus (i) if the (from) purse's status is equal to "Epa" or "Endf," the transaction value was certainly already decreased from the fromPurse's balance, so minus this value is the booked value; (ii) if the (to) purse's status is "Endt," the transaction value was certainly already increased from the fromPurse's balance, so this value is the booked value; otherwise no value was already booked in the purse's balance. The steps in the protocol were the transaction value has already been booked have to be explicitly defined because in **JML** the behavior specification is restricted to a method, and one can not refer to an event which happened before a method's execution (which is the case of the method *ack_operation* where one can not state in its contract that the value had been already booked in the purse's balance). The value returned by this method is ensured by the class constraints that establishes that whenever the purse's balance changes, the booked value is equal to the difference of the purse's actual balance and the previous balance.

5.5 Expressing the Security Properties

The *showProperties* method's specification is described in the sequence.

```
/* @ public behavior
 1 @ requires
   @ (status != Idle) ==>
   @  (\exists ConPurseJC x;
   @  x!= null && x.transaction == transaction && x.name != name;
   @  ((status == Endf) ==> (x.status == Endt)) &&
   @  ((status == Endt) ==> ((x.status == Epa) || (x.status == Endf))) &&
   @  ((status == Epa) ==> ((x.status == Epv) || (x.status == Endt))) &&
   @  ((status == Epv) ==> ((x.status == Idle) || (x.status == Epr) ||
   @                        (x.status == Epa))) &&
   @  ((status == Epr) ==> ((x.status == Idle) || (x.status == Epv))));
   @ assignable \nothing;
 2 @ ensures
 3 @ ((status == Idle) ||
 4 @  (\exists ConPurseJC x;
 5 @  x!= null && x.transaction == transaction && x.name != name;
 6 @  ((bookedValue() > 0) ==> (x.bookedValue() < 0)) &&
 7 @  ((x.bookedValue() > 0) ==> (bookedValue() < 0)) &&
 8 @  (bookedValue() + x.bookedValue() +
   @   ((((status == Epa) || (status == Epv)) &&
   @     ((x.status == Epa) || (x.status == Epv))) ? transaction.value : 0)
   @    == 0)));
   @*/
    void showProperties(){}
```

1. The relationship between the purses statuses is established as precondition as already explained in the section 5.3.4. The precondition can be considered as follows. Considering that "*if the purse's status is not Idle then it is taking part in transaction with another purse behaving as defined by this relationship*" is true, then the postcondition is also true.

2. The postcondition is stated from the item three to eight:

    (a) A purse's status is equal to "Idle" (and therefore it is not part of any ongoing transaction), or

    (b) there exists another purse (the counter part purse),

    (c) which is not null, its stored current transaction is the same as this purse's stored current transaction, and its not the same purse as this (its name is different from this purse); and

(d) whenever the booked value of this purse is bigger than zero, the booked value of the counter part purse is smaller than zero; and

(e) whenever the booked value of the counter part purse is bigger than zero, the booked value of this purse is smaller than zero; and

(f) the actual transaction does not change the sum of the purse's balances: the sum of the booked value of each purse and the lost[7] is zero, and therefore this transaction can not alter the sum of the purse's balances.

5.6 Checking the Invariants Consistency

```
 /*@ public behavior
   @ requires true;
   @ ensures false;
   @*/
 void checkConsistency () {}
```

The (dummy) method *checkConsisteny*[8] can be used for checking the class invariants for consistency. Its specification is obviously wrong, but it can be proven if the class invariants are inconsistent.

5.7 Equivalency Among Specifications

The specification presented in this chapter represents one way of specifying the application's behavior. Although they were correct and sufficient to ensure the desired security properties for this application, the application could have been specified in a very different way. Different invariants and contracts could have been written. In addition, there are equivalent ways of writing the same specification. For instance, the contract of the *ack_operation* shown in the sequence

```
/*@ public behavior
  @ requires apdu != null;
  @ assignable status;
  @ ensures (\old(status) == Epa) && (status == Endf);
  @ signals_only ISOException;
  @ signals (ISOException e) (status == \old(status));
  @*/
```

is equivalent to the contract shown in the sequence.

```
/*@ public behavior
  @ requires (apdu != null) && (status == Epa);
  @ assignable status;
  @ ensures (status == Endf);
  @ signals_only ISOException;
  @ signals (ISOException e) (status == \old(status));
```

[7]The lost is defined as follows: if both purses logged the transaction (when their statuses are either equal to "Epa" or "Epv") then the lost is equal to the transaction value, otherwise it is zero.

[8]Method provided by Erik Poll.

```
@ also
@ public behavior
@ requires (apdu != null) && (status != Epa);
@ ensures false;
@ assignable \nothing;
@ signals_only ISOException;
@ signals (ISOException e) true;
@*/
```

With this second form, it is easier to identify that an exception is certainly thrown when the method is called with a status other than "Epa" and that, even when the status equal to "Epa" an exception might be thrown by the method. On the other hand, the other form of the contract is more compact and easier to prove using the **KeY** tool (because it has only one behavior specification). However, one should take care when simplifying or compacting a specification: many times it is preferable to have a longer an clearer specification than a more compact and harder to understand one.

Chapter 6

The Verification

In this case study, some properties could be proven before practically any of the implementation was written. The verification of the method *showProperties* for instance, is independent of any other method, since the properties can be derived from the invariants defined for the application. Thus, the only implementation necessary to prove the security properties is the method *showProperties* itself, which has no body. It should be highlighted that these properties rely on the class invariants, and that these class invariants have to be preserved[1] by the application methods, and therefore one should not incorrectly deduce that the security properties are totally independent from the application.

In addition, when symbolically executing a method call during a proof in **KeY**, it is possible to use the method contract instead of its implementation. For that, one has to ensure that the method's precondition are true in order to continue the proof using the method's postcondition. This feature enables one to validate a system specification before having implemented the full application, reinforcing that for some applications (deductive) verifications is best done top-down, as stated in the paper *A Case Study of Specification and Verification using **JML** in an Avionics Application* ([18]). In this case study, the correctness of the *process* method's contract can be proven using the called methods contracts. Thus, one can ensure that the called methods' contracts correctly represent the defined protocol before implementing the methods. However, one should be aware that this verification does not prove the correctness of the contracts themselves (which would not make sense), rather it validates the specification, showing that it "fits together" (i.e. that the specified contracts and invariants are sufficient, and not contradictory, for the performed proofs). By validating a specification, one shows that the specification correctly represents the desired system definition, that it is the right specification, as it guarantees the desired security properties.

On the other hand, some properties could only be verified after implementing all methods. For each method, its functional behavior was verified (i.e. it was proven that the method correctly implement is contract), and that it preserves the class invariant. These verification tasks were performed separately and, for each method, the **KeY** tool was configured differently in order to distribute the verification effort, resulting in a manageable and still complete verification

[1]In this case study, the methods "strongly" preserve the invariants.

process.

The proofs for this case study can be performed mostly automatically following the **KeY**'s embedded proof strategy. Only quantifiers had to be instantiated by the user. However, the tool's strategy did not always performs its job in most efficiently and the proofs might quickly become too long (and even too heavy for the system to handle). Therefore, the proof was often performed in a kind of mixed way: some initial steps were performed interactively in order to "lead" the tool to a good path, and the rest of the proof was performed automatically.

6.1 The KeY Options Configuration

In **KeY**, one can configure some of the checks performed during the proof[2]. For this case study, the relevant options to be configured are listed below.

- *transactions* tells the tool whether the code uses the **JavaCard** transaction mechanism or not.
- *transactionsAbort* tells the tool whether the code uses the **JavaCard** abort transaction command or not. *When "on" the tool deals with the **JavaCard** transactions assuming "full" transactions semantics, i.e. account for a possible abort. When "off" the tool assumes that the program does not have an explicit (or an implicit) " abort transaction," which simplifies the proof. However, the problem is that there can also be implicit aborts, especially when one has (i) badly written program that does not close the transaction at all, or (ii) one use the while invariant rule inside a transaction. In any case, if one uses "abortOff" the proof is much simpler, but if an abort happens to occur (explicit or implicit) the proof can not be closed, i.e., with the abortOff the calculus is simply incomplete.*[3]
- *throughout* makes the throughout rules available for a proof. It has to be "on" for proving invariant preservation using the *strong invariant* semantics[4].
- *intRules* tells the tool the semantics to be used for **Java** integer arithmetic. The "Java semantics" corresponds exactly to the semantics defined in the **Java** language specification. In particular, this means that arithmetical operations may cause over-/underflow. The "Arithmetic semantics ignoring overflow" treats the primitive finite **Java** types as if they had the same semantics as mathematical integers with infinite range. The "Arithmetic semantics prohibiting overflow" is the same as the last but having the same finite range of the **Java** types. The result of arithmetical operations is not allowed to exceed the range of the **Java** type as defined in the language specification, i.e. one has to prove that this result does not cause an over-/underflow.

[2]This is done through the *Option* bottom followed by *Taclets Options Default.*

[3]Explanation provided by Dr. Wojciech Mostowsky.

[4]When using the *strong invariant* semantics for proving invariant preservation with **KeY**, after each single assignment performed during the proof, the system has to prove that the invariants are preserved. This semantics is usually not available for the **JML** front end, and therefore the strong invariants POs had to be adapted in order to be used for this case study.

- *nullPointerPolicy* tells the tool whether null pointer checks should be done during the proof or not.

In addition, when selecting a method to prove for the generated Proof Obligation (PO), one can select which invariants should be included and what kind of PO should be generated. For this case study, the proofs were performed including all invariants in the PO (i.e. the ConpurseJC class invariants and the invariants from the imported classes). Two different kinds of PO were used for this case study.

- *Behavior* generates the PO to verify the functional correctness of a method according to its contract. The generated PO has the form

  $$\{\mathit{all\ invariants}\}, \{\mathit{preconditions}\} \rightarrow \langle \mathit{method} \rangle \{\mathit{postconditions}\}$$

- *Class specification* generates the PO to verify the invariants preservation by a method. The generated PO has the form

  $$\{\mathit{all\ invariants}\}, \{\mathit{preconditions}\} \rightarrow \langle \mathit{method} \rangle \{\mathit{class\ invariant}\}$$

6.2 Verifying Invariants Consistency

For verifying the consistency of the invariants defined for this case study, the method *checkConsistency* was used. For that, the method's *behavior* PO was verified. Since the method has no body, the options presented in the last section were all set to "off" (no transaction, no abort, no throughout, arithmetic semantics ignoring overflow, and no null pointer check). All invariants were used for this verification. As one would expect, this PO could not be proven, ensuring thus that the invariants are consistent.

6.3 Verifying Behavior

The verification of the *showProperties* and *process* methods' behavior (implementation independent) was performed using the *behavior* PO. The options presented in the last section were all set to "off" (no transaction, no abort, no throughout, arithmetic semantics ignoring overflow, and no null pointer check) since these options refer to code verification and either are not relevant for the *showProperties* methods, or they were configured "on" on each method called by the *process* method and therefore the checks do not need to be performed repeatedly.

The behavior of the implementation dependent methods, however, was verified differently. The *Taclets Options Default* were configured as follows: *transaction* on, *transactionsAbort* abortOn, *throughout* "off," *intRules* "Arithmetic semantics prohibiting overflow," and *nullPointerPolicy* "nullCheck."

6.4 Verifying Invariant Preservation

The class invariant and constraint preservation verification was also performed in two different way: using the strong invariant semantics for the methods that update relevant fields (occurring in the invariants) and (ii) using normal visible

Method	**Nodes**	**Branches**	**Time (min)**
Implementation Independent			
process	4,731	54	10
showProperties	6,565	50	10
Implementation Dependent			
startFrom	3,818	102	5
startTo	3,975	105	5
req	3,482	95	5
val	3,525	91	5
ack	2,370	69	5
clear_ex_log	1,352	37	5
read_ex_log	28,292	490	35
abort_if_necessary	2,427	57	5

Table 6.1: Methods' Behavior Verification

state semantics for the methods that do not update any relevant field (i.e. the ones that trivially preserve the invariants).

The invariant preservation of the method *process* was not proven because up to the point of its execution were the other methods are called none of the fields that occur in the invariants is updated, and each called methods was proven to strongly preserve the invariants. In addition, the invariants preservation by the method *read_ex_log* was proven using the visible state semantics because it does not update any relevant field (see its assignable clause).

The invariant preservation for all the other methods was proven using the strong invariant semantics. The verification using this semantics can easily become a heavy task to perform: after each assignment performed in the code symbolic execution, an invariant preservation check is included in the PO, making the proof very long, and sometimes requiring too much memory or time to be performed. In order to avoid this problem for this verification, the **KeY** tool had the following configuration.

- Due to the size of the PO generated during the proofs, the PO generated by **KeY** was slightly changed. Originally, **KeY** requires that all invariants included in the left side of a PO's arrow are preserved by the method (including the imported API invariants and the automatically generated inReachableState invariant). For this case study, however, only the application invariants (from the class ConPurseJC) are proven to be preserved[5].
- Since **KeY** ignores the *throughout* configuration when using the **JML** front-end[6], the generated PO was adapted for this case study, handling strong invariants even though **JML** was used to specify the code.
- In order to decrease the verification effort and considering that the absence of overflow and null pointers was already proven in the methods'

[5]In fact, it is not clear that a method should preserve all invariants from the imported classes.

[6]In **JML** only the *visible state* semantics is defined.

Method	**Nodes**	**Branches**	**Time (min)**
VISIBLE STATE			
read_ex_log	7,001	83	20
STRONG INVARIANT			
startFrom	19,084	44	10
startTo	19,015	40	10
req	23,165	64	15
val	18,689	51	15
ack	14,199	32	10
clear_ex_log	7,588	18	5
abort_if_necessary	8,761	33	5

Table 6.2: Invariant Preservation Verification

behavior verification, these extra checks were not included in this PO (*intRules*: "Arithmetic semantics ignoring overflow," and *nullPointerPolicy*: "noNullCheck").

6.5 Verification Results

The details of the performed verification can be seen in tables 6.1 and 6.2. The number of nodes and branches refer to the proof tree, and the time to the time a proof can be reproduced from scratch. Related to the verification performed in this case study, some points should be highlighted.

- The time for performing the proofs—it took a considerable amount of time to specify and code correctly, but after that the proofs can be reproduced in a quite short order. Of course knowing the application helps driving the proof process, and redoing a proof seems to always be easier then proving something for the first time. The amount of user interaction was also quite minimal, being restricted mostly to quantifiers instantiation and defining the scope for the proof to proceed automatically.

- The methods' behavior—proving the correctness of each method's implementation according to its contract, shows that by just receiving an APDU the methods correctly implement their specification (the only precondition for the methods is that the APDU received as a parameter may not be null). Any erroneous APDU is ignored, i.e. the method terminates without changing its state. Thus, a method can receive any number of erroneous APDU without affecting its state. In addition, an APDU to be accepted needs to match with the purse's status. Therefore, when processing any protocol APDU (excluding the exception log related operations) the purse changes its status, no protocol APDU can be processed twice, because the second time it no longer matches with the purse's status. An APDU sent more than once for reading the exception log is not a problem because none of the purse's fields is updated. An APDU for clearing the exception log sent more than once is also not a problem because from the second time that it would be processed, the clear code would no longer match with the already cleared exception log.

- The advantage of proving invariant and constraint preservation using strong invariants semantics—proving that the application methods strongly preserve the invariants means that *the application is tear-safe* according to its invariants (see next item). It is interesting to observe that the tear-safe property can only be fully verified at implementation level because it can check low level failures not visible in the method's contract. For instance, consider the same implementation of the *abort_if_necessary* method presented in the appendix B without the begin and commit transaction commands.

```
 private void abort_if_necessary() throws ISOException
  {
    if (!((status == Epv) || (status == Epa)))
      status = Idle;
    else if (logIdx >= exLog.length)
           ISOException.throwIt(SW_LOG_FULL);
    else
    {
      exLog[logIdx] = transaction;
      logIdx++;
-----------------------------------------------------
      status = Idle;
    }
  }
```

  It implements its contract correctly and preserves the class invariant and constraints using the visible state semantics. However it is not tear-safe: if the method's execution is interrupted at the point marked by the dashed line, the class constraint

```
((\old(logIdx) != logIdx) ==> (status == Idle))
```

  is not preserved. In this situation, when the purse is again selected, according to its status (certainly still equal to "Epv" or "Epa") the same transaction is logged again. In this case, making the exception log pointer and the status be updated atomically solves this problem. In addition, verifying the strong preservation of invariants for a method might help identifying missing transactions in the method's body.

- The tear-safe property scope—the tear-safe property is related to the defined invariants. Like a method, which is not correct by itself but rather according to its specification, an implementation is not tear-safe in general, but according to the invariants and constraints proved to be preserved. For instance, if the constraint on the exception log pointer was not specified, the proof of invariants and constraints preservation for the code presented previously using strong invariants would succeed and the code would be considered tear-safe! This example also shows how important an invariant definition that covers the security aspects of an application is.

- The protocol definition derivation—it is interesting to observe that the defined protocol to be followed by each purse (the sequence of statuses) did not have to be establish as constraints on the purses statuses. Its definition could be derived from the method's contract definition (see *process* method's definition). This offers an extra check on the correctness of the method's contract specification (the protocol derivation ensures that the methods' specification respects the protocol definition).

6.6 The Security Properties

Up until now, it was shown that the protocol is correctly implemented, that each method correctly implements its contract, that the invariants are preserved at any time of the application execution, that no message can be repeatedly processed, etc. It is time to show that the implementation also respects the security properties stated in the section 3.2. As explained above, these properties can be specialized to one single transaction because the result is equivalent to evaluating all possible transactions happening between all pairs of purses existing in the world of purses (see subsection 5.3.3).

6.6.1 Security Property 1: No Value Creation

No value may be created in the system. The sum of all purse's balance does not increase.

To prove this property, it is sufficient to show that (i) the transaction value is the same for both purses, thus the value added in a purse's balance is the same as the value decreased in the counter part purse's balance; (ii) whenever the purse's balance is updated, the *bookedValue()* function correctly represents the value increased or decreased in the balance; and (iii) whenever a purse's *bookedValue()* is positive, the counter part purse's *bookedValue()* is negative, i.e. whenever a purse has added a value in its balance, the counter part purse has already decreased it from its own balance. The proof of these lemmas is shown in the sequence.

- *Lemma i*: the transaction value is received by both purses with the "StartFrom" and the "StartTo" messages at the beginning of the transaction. This value is by definition the same and it is not updated in the application.
- *Lemma ii*: the transaction's value booked in each purse's balance is defined by the method *bookedValue()*. It is guaranteed by the class constraint

  ```
  ((\old(balance) != balance) ==>
        ((\old(balance) - balance) == bookedValue()));
  ```

  that whenever the purse's balance is updated, this function correctly represents the value increased or decreased in the balance.
- *Lemma iii*: by definition of the *bookedValue()* method, when a purse's status is equal to "Idle," its booked value is equal to zero (see section 5.4), and therefore no value was possibly created. Otherwise, the *showProperties* method's postcondition ensures that whenever a purse's booked value

is bigger than zero, the counter part purse's booked value is smaller than zero, and since the modulo value of both booked values is the same their sum is zero. Finally, since the booked values correctly represents the value updated in the purse's balance, the sum of these balances is also zero and no value was possibly created during this transaction execution.

Since this proof relies on a constraint strongly preserved by all application methods, and the *showProperties* method's precondition assumed to be valid at any time, it is ensured that this property holds during the whole transaction execution.

6.6.2 Security Property 2.1: All value accounted

All values must be accounted in the system. The sum of all purse's balance and lost components does not change. A lost component is the transaction value when the transaction is logged by both purses.

Considering initial and final the balances presented by the purses at the beginning and at the end[7] of a transaction respectively, this property can be stated as

$$fromPurse's\ initial\ balance + toPurse's\ initial\ balance =$$
$$fromPurse's\ final\ balance + toPurse's\ final\ balance + lost$$

and that, according to a class constraint, whenever the purse's balance is updated, the value updated is equal to the purse's value logged, i.e.

$$purse's\ final\ balance - purse's\ initial\ balance = purse's\ bookedValue()$$

This property can be stated as

$$fromPurse's\ bookedValue() + toPurse's\ bookedValue() + lost = 0$$

depending still on the definition of the lost component. The lost component can be represented by the **JML** formula

```
((fromPurseLogs && toPurseLogs) ? transaction.value : 0)
```

and considering "fromPurseLogs" and "toPurseLogs" two statements indicating that a purse logs a transaction when its execution terminates while the purse's status is equal to "Epa" or "Epv," which can be stated as

```
((status == Epa) || (status == Epv))
```

the formula becomes as stated in the *showProperties* method's postcondition (adapting the different way of referencing the purses)

```
(bookedValue() + x.bookedValue() +
((((status == Epa) || (status == Epv)) &&
  ((x.status == Epa) || (x.status == Epv))) ? transaction.value : 0)
 == 0)
```

[7]As defined in subsection 5.3.5.

This formula is based on the different ends that the transaction' execution might have: either (i) it terminates normally in both purses, and the lost component and the sum of the purse's balance update are zero, thus holding the property; or (ii) it terminates normally only in one purse, and the lost component is zero and the sum of the purse's balance has to be zero in order to hold the property; or (iii) it terminates abruptly on both purses, and the lost component might be different than zero, and the sum of the lost component and the purse's balance updates must be zero in order to hold the property.

During the proof of the *showProperties* method, this formula is analyzed considering any possible combination of booked values and lost, for any possible combination of the purses statuses. This should be sufficient to explain how the preservation of this property is verified. However, since it is not a trivial formula, it correctness is demonstrated below.

Case 1: Both Purses terminate Normally —in this case the "from" and "to" purses statuses are "Endf" and "Endt" respectively. Then, the value booked by each purse is equal to minus the transaction's value and equal to the transaction's value respectively (according to the definition of *booked-Value*, section 5.4).

Since none of the purse's statuses is equal to "Epa" or "Epv," this transaction is not logged by any of the purse's and therefore the lost component is equal to zero. Finally, the property to hold becomes

$$(-transaction's\ value) + transaction's\ value + 0 = 0$$

which is trivially true.

Case 2: Only One Purse terminates Normally —in this case, since only one purse terminates the transaction's execution abruptly, and therefore only one purse might log the transaction, the lost value is certainly zero. Thus it remains to be demonstrated that the sum of the purse's booked value is equal to zero.

Case 2.1: The fromPurse terminates normally —in this case, the fromPurse's status is equal to "Endf" and its booked value is equal to minus the transaction's value. According to the protocol and the explanation given in the subsection 5.3.4, the counter part purse's status can only be equal to "Endt," and its booked value is thus equal to the transaction's value. The property to hold also becomes

$$(-transaction's\ value) + transaction's\ value + 0 = 0$$

Case 2.2: The toPurse terminates normally —in this case, the toPurse's status is equal to "Endt" and its booked value is equal to the transaction's value. According to the protocol and the explanation given in the subsection 5.3.4, the counter part purse's status can be equal to "Epa" or "Endf," and in any case its booked value is equal to minus the transaction's value. The property to hold once more becomes

$$(-transaction's\ value) + transaction's\ value + 0 = 0$$

Case 3: Both Purses terminate Abruptly —in this case, any possible combination of the purse's statuses has to be verified.

- **Case 3.1: The fromPurse terminates in Idle** —the transaction did not even start for the from purse, thus its booked value is equal to zero and the transaction is not logged. The toPurse's status is equal to "Idle" or "Epv" and in any case its booked value is equal to zero and the transaction is not logged. The formula to hold becomes $0 + 0 + 0 = 0$.
- **Case 3.2: The fromPurse terminates in Epr** —no value was booked and the transaction is not logged. The counter part purse's status is equal to "Idle" or "Epv." If "Idle," no value was booked and the transaction is not logged. If "Epv," its booked value is equal to zero and the transaction is logged but, since only one purse logs the transaction the lost component is equal to zero. The formula to hold becomes $0 + 0 + 0 = 0$.
- **Case 3.3: The fromPurse terminates in Epa** —its booked value is equal to minus the transaction's value, and it logs the transaction. The counter part purse's status is equal to "Epv" (it can not be "Endt" because the counter part purse terminates abruptly). Its booked value is zero and it logs the transaction and, since both purses logged the transaction, the lost component is equal to the transaction's value. The formula to hold becomes

$$(-transaction's\ value) + 0 + transaction's\ value = 0$$

6.6.3 Security Property 2.2: Exception Logging

If a purse aborts a transfer at a point where value could be lost, then the purse logs the details.

As stated in the original definition, and shown in figure 3.1, money can only be lost either when the fromPurse aborts a transaction while its status is equal to "Epa" or when the toPurse aborts a transaction while its status is equal to "Epv." In both cases the transaction is logged when performing the *abort_it_necessary* method during the initialization procedure.

This property is ensured by the *abort_it_necessary* method's postcondition.

6.6.4 Security Property 3: Authentic Purses

A transfer can only occur between authentic purses.

As stated before (chapter 4, "There certainly exists some previous steps in the protocol where the host application requests the purse's details and checks the counter part purse's authenticity (accepting or not to take part in a transaction with that purse)." Thus, it is assumed that both purses taking part in a transaction are authentic. In addition, considering that

- the transaction's counter part purse id is received by the purse at the beginning of the transaction (startFrom or startTo operations),

- the transaction's counter part purse is not changed by any other operation until the end of the transaction (see the method's assignable clause), and
- the transaction details sent with the APDU is tested before each of the transaction operations ("req," "val," and "ack") by calling the *checkSame-Transaction* method,

One can ensure this property holds during the transaction's execution.

6.6.5 Security Property 4: Sufficient Funds

A transfer can occur only if there are sufficient funds in the from purse.

This property is ensured to hold by the class invariant

```
(status == Epr) ==> (transaction.value <= balance)
```

which states that whenever the purse's status is "Epv" (which can only happen in the fromPurse) the transaction's value is smaller than or equal to the purse's balance.

The fromPurse's balance is checked when starting the transaction (by the *readCounterPartDetails* method, called by the "startFrom" and "startTo" operation). If the fromPurse's balance is smaller than the transaction value, an error "Insufficient Funds" is sent to the host application and the transaction is not started.

Finally, since the property is stated as an invariant, it is strongly preserved by all methods in the application.

In addition, in the same way that the fromPurse's balance is checked to ensure that it has sufficient funds to take part in the transaction, the toPurse balance is checked to ensure that its balance can receive the transaction's value (i.e. to ensure that the transaction's value can be added in its balance without causing an overflow). This property is not defined in the case study. It is ensured by the class invariant

```
(status == Epv) ==> (transaction.value <= (ShortMaxValue - balance))
```

which is also strongly preserved by all methods in the application.

Chapter 7

Conclusion

This case study demonstrates that deductive formal verification can be used to verify the consistency of the given specification, to validate it, and proof the correctness of its **JavaCard** implementation, bridging the gap between specification and implementation and ensuring a fully verified result. This goal was achieved by fully specifying, implementing, and verifying the Mondex case study. Although this result can not yet be extended to all kind of application, the case study certainly represents the complexity encountered in not only in **JavaCard** but also many other real applications.

The work described herein was developed in a bit less than four months: one month for learning **Z** (basics), the purse's definition, and **JavaCard**; two months specifying, developing, and verifying; and three weeks writing this report. Previous knowledge consisted of the **Java** programing language, **JML**, and the use of the **KeY** tool. Writing a **JavaCard** application for the first time takes time because there are many details to be taken into account. However, the intention in this case study was to produce a representative implementation that could be used for verification purposes and not a finished product[1] The largest amount of time was spent with the specification, writing and adjusting it rather than verifying. In (approximated) numbers, the case study can be summarized as follows: 63 pages of relevant **Z** specification were taken into account, and 327 lines of **JavaCard** code and 185 lines of **JML** specification were written.

Correctly and completely specifying an application, in the sense of being sufficient to ensure the desired properties, is usually a difficult task. It is the core of any verification effort: besides describing how the application should behave, it describes what has to, or can be, verified. Therefore, it is natural to spend a significant amount of time of a software development project specifying it. This case study demonstrates that deductive software verification can be also be used to validate and verify the consistency of a specification. The **KeY** tool can be used to verify the security properties even before a full implementation is available. This verification ensures that the methods' contracts correctly represented the protocol before implementing any of the methods, which helps save implementation time. Finally, after implementing the methods, the only

[1]The code is in fact not complete. There are details about application identification, protocol, and channels, that were not implemented. In addition, it was not tested in a card simulation environment, like the one in the Sun MicroSystems' **JavaCard** development kit.

work left is to verify that each method correctly implements its contract and that it preserves the application invariants.

A general methodology could be formulated in three steps.

1. First, one specifies the application (invariants, constraints and methods contracts) and the general properties to be verified. The consistency of the invariants can thus be check. In this step, using the case study as an example, the purse can be fully specified in **JML**. In addition, the consistency of the invariants can be checked using the method *checkConsistency*.

2. Then, one would validate the specification previously developed. This step usually implies several steps of verifying and correcting the specification. Again, considering this case study, in the second step, the methods *showProperties* and *process* can be developed and verified, before developing methods body of the other methods (the operations).

    - Proving the showProperties one can ensure that the specification guarantees the security properties of the application.
    - In **KeY** having only the the process method body, one can prove its behavior by using the method's contracts instead of their bodies. Proving the process method's behavior, and the invariants preservation, one ensures that
        - the method's contracts correctly represents the expected behavior, and
        - the invariants are preserved by the whole application.

3. Finally, after validating and ensuring the consistency of the specification, one would implement the methods and use the methods' contracts to verify the implementation. Considering this case study, in the third step, the body of the remaining methods (the operations) can be developed and it correctness verified against their respective contracts (already developed in the step one).

The time spent for this case study would certainly have been smaller had this methodology been followed rather than the usual way of implementing, specifying, and then verifying. Therefore, this methodology is a significant result of this case study. There are, however, limitations on its usage. In this case study, verification was performed using **KeY**, and this tool has its own limitations. It actually can fully verify **JavaCard** code. The group is working on extending **KeY** to full **Java**, which is advancing well but still faces some theoretical issues, such as the verification of floating point numbers. Thus, only for **JavaCard** implementations can this methodology be fully used with **KeY**.

In short, not only can deductive verification with **JML** be used for proving the correctness of an implementation of a specification, but using constraints, the consistency of a protocol specification can be shown as well.

The complete material—source code, proofs and binaries of the verification system as well this report—is available at `http://www.key-project.org/case_studies/mondex.html`.

Acknowledgments

This work was supported in part by a Formal Methods Europe (FME) Small Grant. The findings and opinions in this technical report belong solely to the author and are not necessarily those of the sponsor.

The author thanks Prof. Peter H. Schmitt, head of the **KeY** group in Karlsruhe, for the opportunity to develop this case study and mainly for sharing his long experience in software specification and verification during this past four months. In addition, the author thanks the entire **KeY** group of Karlsruhe for its support, especially Christian Engel for adapting the tool when necessary. Thanks also go to two external colleagues who contributed continually to this work. First to Dr. Richard Bubbel, the best colleague one can have, who is a member of the **KeY** group at Chalmers, for his invaluable help and advise. Also to Dr. Wojciech Mostowski, a former member of the **KeY** group, now at the Radboud University Nijmegen, for his helpful assistance explaining how the **JavaCard** specific features are handled in **KeY**. Finally, the author thanks everyone who commented or reviewed this report, specially Dr. Erik Poll, Lecturer of Computer Science, Radboud University Nijmegen; and Dr. James J. Hunt, CEO of aicas GmbH, Germany.

This work would have been much harder to perform without them!

Bibliography

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041, pages 6–24. Springer-Verlag, 2001.

[3] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proc. Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.

[5] Y. Cheon and A. Perumandla. Specifying and checking method call sequences of java programs. *Software Quality Control*, 15(1):7–25, 2007.

[6] T. Clark and J. Warmer, editors. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS.* Springer, 2002.

[7] V. Fodor, O.; Hassler. Javacard and opencard framework: a tutorial. In *7th IEEE International Conference on Emerging Technologies and Factory Automation, 1999. Proceedings. ETFA '99*, volume 1, pages 13–22, 1999. Digital Object Identifier 10.1109/ETFA.1999.815333.

[8] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[10] E. Hubbers, M. Oostdijk, and E. 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-Verlag, 2004.

[11] J. J. Hunt, F. B. Siebert, P. H. Schmitt, and I. Tonin. Provably correct loops bounds for realtime java programs. In *JTRES '06: Proceedings of the*

4th international workshop on Java technologies for real-time and embedded systems, pages 162–169, New York, NY, USA, 2006. ACM Press.

[12] G. T. Levens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual. http://www.jmlspec.org/, 2004.

[13] R. Marlet and D. L. 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.

[14] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[15] Uk itsec scheme certification report no.p129, mondex purse, 1999.

[16] E. Ortiz. An introduction to java card technology - part 1, May 2003.

[17] T. Ramananandro. Mondex, an eletronic purse: specification and refinement checks with the alloy model-finding method. Technical report, École Normale Supérieure, Paris, France, September 2006.

[18] P. Schmitt, I. Tonin, C. Wonnemann, E. Jenn, S. Leriche, and J. J. Hunt. A case study of specification and verification using JML in an avionics application. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 107–116, New York, NY, USA, 2006. ACM Press.

[19] S. Stepney, D. Cooper, and J. Woodcock. An eletronic purse — specification, refinement, and proof. Technical report, Oxford University Computing Laboratory, Programing Research Group, July 2000.

[20] J. Woodcock. First steps in the verified software grand challenge. In *IEEE Computer Society*, pages 57–64. October 2006.

Appendix A

The Mondex Application APDU

Start From Operation							
Command APDU							
Cla	**Ins**	**P1**	**P2**	**Lc**	**Data**		
(0) 0xB0	(1) 1	(2) Purse's name		(4) 6	(5) Cpd Name	(7) Cpd Seq	(9) Value

Response APDU	
Status Word	**Meaning of Status Word**
0x6740	Log Full
0x6200	Ignored – Status $\neq$ Idle
ISO7816.SW_WRONG_LENGTH	LC wrong Length
0x6400	Transaction Error
0x6700	Invalid counter part details
0x6710	Invalid value
0x6730	Insufficient funds
0x6760	APDU Error
0x9000	Successful processing

Table A.1: Start From Purse Operation APDU

Start To Operation							
Command APDU							
Cla	**Ins**	**P1**	**P2**	**Lc**	**Data**		
(0) 0xB0	(1) 2	(2) Purse’s name		(4) 6	(5) Cpd Name	(7) Cpd Seq	(9) Value

Response APDU	
Status Word	**Meaning of Status Word**
0x6740	Log Full
0x6200	Ignored – Status $\neq$ Idle
ISO7816.SW_WRONG_LENGTH	LC wrong Length
0x6400	Transaction Error
0x6700	Invalid counter part details
0x6710	Invalid value
0x6720	Value Overflow
0x6760	APDU Error
0x9000	Successful processing

Table A.2: Start To Purse Operation APDU

Req Operation							
Command APDU							
Cla	**Ins**	**P1**	**P2**	**Lc**	**Data**		
(0) 0xB0	(1) 3	(2) Purse’s name		(4) 6	(5) Cpd Name	(7) Cpd Seq	(9) Value

Response APDU	
Status Word	**Meaning of Status Word**
0x6200	Ignored – Status $\neq$ Epr or wrong Cpd
ISO7816.SW_WRONG_LENGTH	LC wrong Length
0x6400	Transaction Error
0x6760	APDU Error
0x9000	Successful processing

Table A.3: Request Receipt (REQ) Operation APDU

Val Operation							
Command APDU							
Cla	**Ins**	**P1**	**P2**	**Lc**	**Data**		
(0) 0xB0	(1) 4	(2) Purse’s name		(4) 6	(5) Cpd Name	(7) Cpd Seq	(9) Value

Response APDU	
Status Word	**Meaning of Status Word**
0x6200	Ignored – Status $\neq$ Epv or wrong Cpd
ISO7816.SW_WRONG_LENGTH	LC wrong Length
0x6400	Transaction Error
0x6760	APDU Error
0x9000	Successful processing

Table A.4: Value Receipt (VAL) Operation APDU

Ack Operation							
Command APDU							
Cla	**Ins**	**P1**	**P2**	**Lc**	**Data**		
(0) 0xB0	(1) 5	(2) Purse’s name		(4) 6	(5) Cpd Name	(7) Cpd Seq	(9) Value

Response APDU	
Status Word	**Meaning of Status Word**
0x6200	Ignored – Status $\neq$ Epa or wrong Cpd
ISO7816.SW_WRONG_LENGTH	LC wrong Length
0x6760	APDU Error
0x9000	Successful processing

Table A.5: Acknowledge Receipt (ACK) Operation APDU

ReadExLog Operation					
COMMAND APDU					
Cla	**Ins**	**P1**	**P2**	**Le**	
(0) 0xB0	(1) 8	(2) Purse's name		(4) 0 or (Log length * 10)	
RESPONSE APDU					
Data (0 - (le - 1))					
1st logged transaction					2nd . . .
(0)fromName	(2)toName	(4)value	(6)fromSeq	(8)toSeq	(10) . . .

Status Word	**Meaning of Status Word**
0x63XX	XX is the Log length
ISO7816.SW_WRONG_LENGTH	LC wrong Length
0x6760	APDU Error
0x9000	Successful processing

Table A.6: Read Exception Log (ReadExLog) Operation APDU

ClearExLog Operation					
COMMAND APDU					
Cla	**Ins**	**P1**	**P2**	**Lc**	**Data**
(0) 0xB0	(1) 9	(2) Purse's name		(4) 2	(5) Clear code

RESPONSE APDU	
Status Word	**Meaning of Status Word**
0x6200	Ignored – Status $\neq$ Idle
ISO7816.SW_WRONG_LENGTH	LC wrong Length
0x6750	Invalid clear code
0x6760	APDU Error
0x9000	Successful processing

Table A.7: Clear Exception Log (ClearExLog) Operation APDU

Appendix B

The Mondex JavaCard Code

Java + JML

```
/********************************************************
 * The Mondex Case Study - The KeY Approach
 * Purse main class
 * author: Dr. Isabel Tonin (tonin@ira.uka.de)
 * Universitaet Karlsruhe - Institut fuer Theoretische Informatik
 * http://key-project.org/
 */

import javacard.framework.APDU;
import javacard.framework.APDUException;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.Util;
import javacard.framework.TransactionException;
import javacard.framework.JCSystem;

public class ConPurseJC extends Applet
{

/********************************************************
 * Class Invariants
 */

/*@ public invariant
  @ (exLog != null) && (exLog.length > 0) &&
  @ (exLog.length < (APDU.BUFFER_LENGTH / 10)) &&
  @ (logIdx >= 0) && (logIdx <= exLog.length) &&
  @ (balance >= 0) && (balance <= ShortMaxValue) &&
  @ (nextSeq >= 0) && (nextSeq <= ShortMaxValue) &&
  @ (status >= 0) && (status <= 5) &&
  @ (transaction != null) && (transaction.value > 0) &&
  @ ((status == Epr) ==> (transaction.value <= balance)) &&
  @ ((status == Epv) ==> (transaction.value <= (ShortMaxValue - balance))) &&
  @ (\forall byte i; i>=0 && i<exLog.length; exLog[i] != null);
  @*/

/*@ public constraint
```

```
  @ ((\old(balance) != balance) ==>
  @    ((balance - \old(balance)) == bookedValue()));
  @*/

/*@ public constraint
  @ ((\old(logIdx) != logIdx) ==> (status == Idle));
  @*/

/*********************************************************
 * Method for checking the consistency of the invariants
 */

 /*@ public behavior
   @ requires true;
   @ ensures false;
   @*/
 void checkConsistency () {}

/*********************************************************
 * Method for showing the Security Properties
 */

/*@ public behavior
  @ requires
  @ (status != Idle) ==>
  @  (\exists ConPurseJC x; x!= null && x.transaction == transaction && x.name != name;
  @  ((status == Endf) ==> (x.status == Endt)) &&
  @  ((status == Endt) ==> ((x.status == Epa) || (x.status == Endf))) &&
  @  ((status == Epa) ==> ((x.status == Epv) || (x.status == Endt))) &&
  @  ((status == Epv) ==> ((x.status == Idle) || (x.status == Epr) ||
  @                        (x.status == Epa))) &&
  @  ((status == Epr) ==> ((x.status == Idle) || (x.status == Epv))));
  @ assignable \nothing;
  @ ensures
  @ ((status == Idle) ||
  @   (\exists ConPurseJC x; x!= null && x.transaction == transaction && x.name != name;
  @   ((bookedValue() > 0) ==> (x.bookedValue() < 0)) &&
  @   ((x.bookedValue() > 0) ==> (bookedValue() < 0)) &&
  @   (bookedValue() + x.bookedValue() +
  @   ((((status == Epa) || (status == Epv)) &&
  @     ((x.status == Epa) || (x.status == Epv))) ? transaction.value : 0)
  @    == 0)));
  @*/
  public void showProperties(){}

/*@ pure @*/ private short bookedValue()
   {
       if ((status == Epa) || (status == Endf))
           return (short)-transaction.value;
       else if (status == Endt)
           return transaction.value;
       else return 0;
   }

/*********************************************************
 * The Application
 *********************************************************/

/*********************************************************
 * Constants declaration
 */
```

```
// CLA byte in the command APDU header
  private final static byte Mondex_CLA = (byte)0xB0;
// INS byte in the command APDU header

// Z spec value transfer operations
  private final static byte StartFrom = 1;
  private final static byte StartTo = 2;
  private final static byte Req = 3;
  private final static byte Val = 4;
  private final static byte Ack = 5;

// Z spec exception logging operations
  private final static byte ReadExLog = 8;
  private final static byte ClearExLog = 9;

// Z spec Wallet status
  private final static byte Idle = 0; //originally Expecting Any To/From
  private final static byte Epr = 1; //Expecting Request
  private final static byte Epv = 2; //Expecting Value
  private final static byte Epa = 3; //Expecting Acknowledge

// Extra status (replaces final eaTo and eaFrom from the Z spec)
  private final static byte Endf = 4; //Transaction From ended successfully
  private final static byte Endt = 5; //Transaction To ended successfully

/********************************************************
 * SW1 and SW2 for Response APDU Command (as defined in ISO7816-4)
 * (not specified in the Z specification)
 */

// Process Completed
  private final static short SW_RETURN_VALUE = 0x6100;
  private final static short SW_IGNORED = 0x6200;

// Process Interrupted - execution error
  private final static short SW_TRANSACTION_ERROR = 0x6400;

// Process Interrupted - checking error
  private final static short SW_INVALID_CPD = 0x6700;
  private final static short SW_INVALID_VALUE = 0x6710;
  private final static short SW_VALUE_OVERFLOW = 0x6720;
  private final static short SW_INSUFFICIENT_FUNDS = 0x6730;
  private final static short SW_LOG_FULL = 0x6740;
  private final static short SW_INVALID_CLEAR_CODE = 0x6750;
  private final static short SW_APDU_ERROR = 0x6760;

/********************************************************
 * Implementation Specific Constants declaration
 */

// maximum value that a variable can hold
  private final static byte  ByteMaxValue = 127;
  private final static short ShortMaxValue = 32767;

// models any short value (necessary for implementing the image method)
  private static short aShort;

/********************************************************
 * Fields declaration (according to the Z specification)
 */

  private short balance;
  private PayDetails [] exLog;
  private short name;
  private short nextSeq;
  private byte status;
  private byte logIdx;

// actual transaction details
  private PayDetails transaction;

/*******************************************************
 * Methods inherited from Applet
 * (required by the JCRE - Javacard Runtime Environment)
 */

// Constructor: an instance of class ConPurseJC is instantiated by its
// install method.  The applet registers itself with the JCRE by calling
// the register method, which is defined in class Applet.
  private ConPurseJC ()
  {
      name = 0;
      nextSeq = 0;
      balance = 0;
      status = Idle;
// exLog length must be smaller than or equal to the 1/10 APDU buffer length
      exLog = new PayDetails[25];
      transaction = new PayDetails();
      register();
  }

// This method is invoked by the JCRE to create an applet instance and to
// register the instance with the JCRE.  The installation parameters are
// supplied in the byte array parameter, and must be in a format defined
// by the applet. They are used to initialize the applet instance.
// For this case study there is no initialization parameter.
  public static void install(byte[] bArray)
  {
      new ConPurseJC();
  }

// This method is called by the JCRE to indicate that this applet has been
// selected.  It performs the initialization required to process the
// subsequent APDU messages.  The applet can decline to be selected, for
// instance, if the pin is blocked.  In this case study, it does not
// perform any initialization and always accept the selection.
// For the case study the abort operation is performed in order put the
// card in a Idle state ready to start a new transaction, logging the old
// transaction if necessary.
  public boolean select()
  {
      abort_if_necessary();
      return true;
  }

// This method is called by the JCRE to inform the applet that it should
// perform any clean-up and bookkeeping tasks before the applet is
// deselected.
// For the case study not clean-up operation was defined.
  public void deselect()
  {
  }

/*@ public behavior
```

```
  @ requires apdu != null;
  @ assignable logIdx, balance, status, nextSeq, transaction.fromName,
  @    transaction.toName, transaction.fromSeq, transaction.toSeq,
  @    transaction.value,apdu._buffer[0..((logIdx*10) - 1)];
  @ ensures
  @ ((\old(logIdx) != logIdx) ==>
  @   ((logIdx == 0) && (status == Idle) && (\old(status) == Idle))) &&
  @ ((\old(status) == status) ==>
  @   (\old(balance) == balance) && (\old(nextSeq) == nextSeq)) &&
  @
  @ ((\old(status) != status) ==> (
  @  (\old(apdu._buffer[ISO7816.OFFSET_INS]) == apdu._buffer[ISO7816.OFFSET_INS]) &&
  @
  @  ((\old(status) == Idle) ==>
  @   ((((status == Epr)&&(apdu._buffer[ISO7816.OFFSET_INS] == StartFrom)) ||
  @     ((status == Epv)&&(apdu._buffer[ISO7816.OFFSET_INS] == StartTo))) &&
  @     (\old(balance) == balance))) &&
  @  ((\old(status) == Epr) ==> ((status == Epa) &&
  @   (apdu._buffer[ISO7816.OFFSET_INS] == Req) && (\old(balance) > balance))) &&
  @  ((\old(status) == Epv) ==> ((status == Endt) &&
  @   (apdu._buffer[ISO7816.OFFSET_INS] == Val) && (\old(balance) < balance))) &&
  @  ((\old(status) == Epa) ==> ((status == Endf) &&
  @   (apdu._buffer[ISO7816.OFFSET_INS] == Ack) && (\old(balance) == balance))) &&
  @
  @  (status != Idle) &&
  @  ((status == Epr) ==> (\old(status) == Idle)) &&
  @  ((status == Epv) ==> (\old(status) == Idle)) &&
  @  ((status == Epa) ==> (\old(status) == Epr)) &&
  @  ((status == Endf) ==> (\old(status) == Epa)) &&
  @  ((status == Endt) ==> (\old(status) == Epv)) &&
  @
  @  ((\old(balance) > balance) ==> ((status == Epa) &&
  @        ((balance - \old(balance)) == -transaction.value))) &&
  @  ((\old(balance) < balance) ==> ((status == Endt) &&
  @        ((balance - \old(balance)) == transaction.value))) &&
  @  ((\old(balance) == balance) ==> ((status == Idle) || (status == Epr) ||
  @        (status == Epv) || (status == Endf))) &&
  @
  @  (((status == Epr) || (status == Epv)) ==>
  @   ((nextSeq == (\old(nextSeq) + 1)) ||
  @   ((nextSeq == 0)  && (\old(nextSeq) >= ShortMaxValue)))) &&
  @  (!((status == Epr) || (status == Epv)) ==>
  @   ((\old(nextSeq) == nextSeq) &&
  @    (\old(transaction.fromName) == transaction.fromName) &&
  @    (\old(transaction.toName) == transaction.toName) &&
  @    (\old(transaction.fromSeq) == transaction.fromSeq) &&
  @    (\old(transaction.toSeq) == transaction.toSeq) &&
  @    (\old(transaction.value) == transaction.value)))));
  @ signals_only ISOException;
  @ signals (ISOException e) (\old(balance) == balance) &&
  @  (\old(status) == status) && (\old(logIdx) == logIdx) &&
  @  (\old(nextSeq) == nextSeq);
  @*/
  public void process(APDU apdu)
  {
// After the applet is successfully selected, the JCRE dispatches incoming
// APDUs to the process method.   At this point, only the first 5 bytes
// [CLA, INS, P1, P2, LC] are available in the APDU buffer.
      byte[] buffer = apdu.getBuffer();
// The JCRE also passes the SELECT APDU commands to the applet
// (which is ignored)
      if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
          (buffer[ISO7816.OFFSET_INS] == (short)(0xA4)) )
          ISOException.throwIt(SW_IGNORED);
// Check CLA
      if (buffer[ISO7816.OFFSET_CLA] != Mondex_CLA)
         ISOException.throwIt (ISO7816.SW_CLA_NOT_SUPPORTED);
// Ignores message not sent to this purse
      if (Util.getShort(buffer, ISO7816.OFFSET_P1) != name)
         ISOException.throwIt(SW_IGNORED);
// Calls the method indicated by the INS byte
      switch (buffer[ISO7816.OFFSET_INS])
      {
      case StartFrom: start_from_operation(apdu); break;
      case StartTo: start_to_operation(apdu); break;
      case Req: req_operation(apdu); break;
      case Val: val_operation(apdu); break;
      case Ack: ack_operation(apdu); break;
      case ReadExLog: read_ex_log_operation(apdu); break;
      case ClearExLog: clear_ex_log_operation(apdu); break;
      default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
      }
  }

/*******************************************************
 * Application Specific Methods
 */

/*@ public behavior
  @ requires true;
  @ assignable exLog[\old(logIdx)], logIdx, status;
  @ ensures
  @ (status == Idle) &&
  @ (((\old(status) == Epv) || (\old(status) == Epa)) ==>
  @  ((\old(logIdx) < exLog.length) && (logIdx == (\old(logIdx) + 1)) &&
  @   (exLog[\old(logIdx)] == transaction))) &&
  @ (((\old(status) != Epv) && (\old(status) != Epa)) ==>
  @  ((logIdx == \old(logIdx)) && (\old(exLog[logIdx]) == exLog[logIdx])));
  @ signals_only ISOException;
  @ signals (ISOException e)
  @ (\old(logIdx) == logIdx) && (\old(status) == status);
  @*/
  private void abort_if_necessary() throws ISOException
  {
      if (!((status == Epv) || (status == Epa)))
      status = Idle;
      else if (logIdx >= exLog.length)
               ISOException.throwIt(SW_LOG_FULL);
           else
           {
              try
              {
                  exLog[logIdx] = transaction;
                  JCSystem.beginTransaction();
                  logIdx++;
                  status = Idle;
                  JCSystem.commitTransaction();
              }
              catch (TransactionException e)
              {
                  ISOException.throwIt(SW_TRANSACTION_ERROR);
              }
          }
  }
```

```
/*@ public behavior
  @ requires apdu != null;
  @ assignable status, transaction.fromName, transaction.toName,
  @   transaction.fromSeq, transaction.toSeq, transaction.value, nextSeq;
  @ ensures
  @ (\old(status) == Idle) && (status == Epr) && (logIdx < exLog.length) &&
  @ ((nextSeq == \old(nextSeq) + 1) ||
  @  ((nextSeq == 0)  && (\old(nextSeq) >= ShortMaxValue))) &&
  @ (transaction.fromSeq == \old(nextSeq)) &&
  @ (transaction.fromName == name) && (transaction.toName != name) &&
  @ (transaction.toName > 0) && (transaction.toSeq >= 0);
  @ signals_only ISOException;
  @ signals (ISOException e)
  @ (\old(status) == status) && (\old(nextSeq) == nextSeq);
  @*/
  private void start_from_operation(APDU apdu) throws ISOException
    {
        if (logIdx >= exLog.length) ISOException.throwIt(SW_LOG_FULL);
        if (status == Idle)
        {
            readCounterPartDetails(apdu, StartFrom);
            try
            {
                JCSystem.beginTransaction();
                if (nextSeq < ShortMaxValue)
                    nextSeq = (short) (nextSeq + 1);
                else nextSeq = 0;
                status = Epr;
                JCSystem.commitTransaction();
            }
            catch (TransactionException e)
            {
                ISOException.throwIt(SW_TRANSACTION_ERROR);
            }
        }
        else ISOException.throwIt(SW_IGNORED);
    }

/*@ public behavior
  @ requires apdu != null;
  @ assignable status, transaction.fromName, transaction.toName,
  @   transaction.fromSeq, transaction.toSeq, transaction.value, nextSeq;
  @ ensures
  @ (\old(status) == Idle) && (status == Epv) && (logIdx < exLog.length) &&
  @ ((nextSeq == \old(nextSeq) + 1) ||
  @  ((nextSeq == 0)  && \old(nextSeq) >= ShortMaxValue)) &&
  @ (transaction.toSeq == \old(nextSeq)) &&
  @ (transaction.toName == name) && (transaction.fromName != name) &&
  @ (transaction.fromName > 0) && (transaction.fromSeq >= 0);
  @ signals_only ISOException;
  @ signals (ISOException e)
  @ (\old(status) == status) && (\old(nextSeq) == nextSeq);
  @*/
  private void start_to_operation(APDU apdu) throws ISOException
  {
      if (logIdx >= exLog.length) ISOException.throwIt(SW_LOG_FULL);
      if (status == Idle)
      {
          readCounterPartDetails(apdu, StartTo);
          try
          {
            JCSystem.beginTransaction();
            if (nextSeq < ShortMaxValue)
                nextSeq = (short) (nextSeq + 1);
            else nextSeq = 0;
            status = Epv;
            JCSystem.commitTransaction();
        }
        catch (TransactionException e)
        {
            ISOException.throwIt(SW_TRANSACTION_ERROR);
        }
    }
    else ISOException.throwIt(SW_IGNORED);
}

/*@ public behavior
  @ requires apdu != null;
  @ assignable balance, status;
  @ ensures
  @ (balance == \old(balance)-transaction.value) &&
  @ (\old(status) == Epr) && (status == Epa);
  @ signals_only ISOException;
  @ signals (ISOException e)
  @ ((balance == \old(balance)) && (status == \old(status)));
  @*/
  private void req_operation(APDU apdu) throws ISOException
  {
      if (status == Epr)
      {
          checkSameTransaction(apdu);
          try
          {
              JCSystem.beginTransaction();
              balance = (short)(balance - transaction.value);
              status = Epa;
              JCSystem.commitTransaction();
          }
          catch (TransactionException e)
          {
              ISOException.throwIt(SW_TRANSACTION_ERROR);
          }
      }
      else ISOException.throwIt(SW_IGNORED);
  }

/*@ public behavior
  @ requires apdu != null;
  @ assignable balance, status;
  @ ensures
  @ ((balance == \old(balance)+transaction.value) &&
  @ (\old(status) == Epv) && (status == Endt));
  @ signals_only ISOException;
  @ signals (ISOException e)
  @ ((balance == \old(balance)) && (status == \old(status)));
  @*/
  private void val_operation(APDU apdu) throws ISOException
  {
      if (status == Epv) {
          checkSameTransaction(apdu);
          try
          {
              JCSystem.beginTransaction();
```

```
                balance = (short)(balance + transaction.value);
                status = Endt;
                JCSystem.commitTransaction();
            }
            catch (TransactionException e)
            {
                ISOException.throwIt(SW_TRANSACTION_ERROR);
            }
        }
        else ISOException.throwIt(SW_IGNORED);
    }

  /*@ public behavior
    @ requires apdu != null;
    @ assignable status;
    @ ensures (\old(status) == Epa) && (status == Endf);
    @ signals_only ISOException;
    @ signals (ISOException e) (status == \old(status));
    @*/
    private void ack_operation(APDU apdu) throws ISOException
    {
        if (status == Epa)
        {
            checkSameTransaction(apdu);
            status = Endf;
        }
        else ISOException.throwIt(SW_IGNORED);
    }

      //          /*@ loop_invariant i <= logIdx && i >= 0;

  /*@ public behavior
    @ requires apdu != null;
    @ assignable apdu._buffer[0..((logIdx*10) - 1)];
    @ ensures true;
    @ signals_only ISOException;
    @ signals (ISOException e) true;
    @*/
    private void read_ex_log_operation(APDU apdu) throws ISOException
    {
        byte[] buffer = apdu.getBuffer();
        if (buffer[ISO7816.OFFSET_LC] == 0)
            ISOException.throwIt((short)(SW_RETURN_VALUE + logIdx));
        if (buffer[ISO7816.OFFSET_LC] != (logIdx * 10))
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        byte i = 0;
      /*@ loop_invariant (logIdx >= i) && (i >= 0);
        @ assignable i, buffer[0..(((logIdx - 1) * 10) -1)];
        @ decreases (logIdx - i);
        @*/
        while (i < logIdx)
        {
            Util.setShort(buffer, (short)(i*10), exLog[i].fromName);
            Util.setShort(buffer, (short)(i*10+2), exLog[i].toName);
            Util.setShort(buffer, (short)(i*10+4), exLog[i].value);
            Util.setShort(buffer, (short)(i*10+6), exLog[i].fromSeq);
            Util.setShort(buffer, (short)(i*10+8), exLog[i].toSeq);
            i++;
        }
        try
        {
            apdu.setOutgoingAndSend((short)0, (short)exLog.length);
        }
        catch (APDUException e)
        {
            ISOException.throwIt(SW_APDU_ERROR);
        }
    }

  /*@ public behavior
    @ requires apdu != null;
    @ assignable logIdx;
    @ ensures (logIdx == 0) && (status == Idle);
    @ signals_only ISOException;
    @ signals (ISOException e) (logIdx == \old(logIdx));
    @*/
     private void clear_ex_log_operation(APDU apdu) throws ISOException
     {
         if (status == Idle)
         {
             try
             {
                 if (apdu.setIncomingAndReceive() != 2)
                     ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
                 byte[] buffer = apdu.getBuffer();
                 if (image() != Util.getShort(buffer, (short) 5))
                     ISOException.throwIt(SW_INVALID_CLEAR_CODE);
                 logIdx = 0;
             }
             catch (APDUException e)
             {
                 ISOException.throwIt(SW_APDU_ERROR);
             }
         }
         else ISOException.throwIt(SW_IGNORED);
     }

  /*@  public normal_behavior
    @  requires true;
    @  assignable \nothing;
    @  ensures true;
    @*/
    private short image()
    {
        return aShort;
    }

    private void readCounterPartDetails(APDU apdu, byte ins)
        throws ISOException
    {
        try
        {

            if (apdu.setIncomingAndReceive() != 6)
                ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
            byte[] buffer = apdu.getBuffer();
            short cpdName = Util.getShort(buffer, (short) 5);
            short cpdSeq = Util.getShort(buffer, (short) 7);
            short value = Util.getShort(buffer, (short) 9);
            if ((cpdName == name) || !(cpdName > 0) || (cpdSeq < 0))
                ISOException.throwIt(SW_INVALID_CPD);
            if (value <= 0) ISOException.throwIt(SW_INVALID_VALUE);
            if (ins == StartFrom)
                if (value > balance)
                ISOException.throwIt(SW_INSUFFICIENT_FUNDS);
            else
            {
                transaction.fromName = name;
                transaction.toName = cpdName;
                transaction.fromSeq = nextSeq;
                transaction.toSeq = cpdSeq;
                transaction.value = value;
            }
        if (ins == StartTo)
            if (value > (short) (ShortMaxValue - balance))
                ISOException.throwIt(SW_VALUE_OVERFLOW);
            else
            {
                transaction.fromName = cpdName;
                transaction.toName = name;
                transaction.fromSeq = cpdSeq;
                transaction.toSeq = nextSeq;
                transaction.value = value;
            }
    }
    catch (APDUException e)
    {
        ISOException.throwIt(SW_APDU_ERROR);
    }
  }

  private void checkSameTransaction(APDU apdu) throws ISOException
  {
    if ((status != Epr) && (status != Epa) && (status != Epv))
        ISOException.throwIt(SW_IGNORED);
    try
    {
        if (apdu.setIncomingAndReceive() != 6)
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    }
    catch (APDUException e)
    {
        ISOException.throwIt(SW_APDU_ERROR);
    }
    byte[] buffer = apdu.getBuffer();
    short cpdName = Util.getShort(buffer,(short)5);
    short cpdSeq = Util.getShort(buffer,(short)7);
    short value = Util.getShort(buffer,(short)9);
    if ((value != transaction.value) ||
        (((status == Epr) || (status == Epa)) &&
         ((transaction.toName != cpdName) ||
          (transaction.toSeq != cpdSeq))) ||
        ((status == Epv) && ((transaction.fromName != cpdName) ||
                             (transaction.fromSeq != cpdSeq))))
        ISOException.throwIt(SW_IGNORED);
  }
}

/**********************************************************
 * The Mondex Case Study - The KeY Approach
 * Payment Details class
 * author: Dr. Isabel Tonin (tonin@ira.uka.de)
 * Universitaet Karlsruhe - Institut fuer Theoretische Informatik
 * http://key-project.org/
 */
```

```
public class PayDetails
{
  /* Z spec TransferDetails fields */
  short fromName = 0;
  short toName = 0;
  short value = 0;
  /* Z spec PayDetails extra fields */
  short fromSeq = 0;
  short toSeq = 0;

  protected PayDetails () {}

  public boolean equals(PayDetails x)
  {
    return (x.fromName == fromName &&
            x.toName   == toName &&
            x.value    == value &&
            x.fromSeq  == fromSeq &&
            x.toSeq    == toSeq);
  }
}
```

Java + JML