Int J Software Informatics, Volume 5, Issue 1-2 (2011), Part I, pp. 159–184
International Journal of Software and Informatics, ISSN 1673-7288

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Mondex: Engineering a Provable Secure Electronic Purse

Dominik Haneberg, Nina Moebius, Wolfgang Reif, Gerhard Schellhorn,

and Kurt Stenzel

(Lehrstuhl für Softwaretechnik und Programmiersprachen, Augsburg University, Germany)

Abstract Mondex electronic purses are used to replace cash payment by electronic payment with a smart card. Tool-based proofs of an abstract specification of the security protocol used by Mondex cards has been proposed as a challenge for mechanized verification. This paper extends the challenge and describes a comprehensive formal approach for developing Mondex that starts with the abstract specification that was used to certify Mondex, and ends with correct Java code. The development was verified with the theorem prover KIV using several refinements. Mondex cards can also be viewed as a challenge for software engineering a security-critical application. We describe a model-driven tool-supported approach, that uses UML models enhanced with security aspects, and generates correct and secure Java Card code. The UML models can also be used to generate the formal specifications used in KIV.

Key words: electronic purse; security; Mondex; refinement; abstract state machines

Haneberg D, Moebius N, Reif W, Schellhorn G, Stenzel K. Mondex: Engineering a provable secure electronic purse. *Int J Software Informatics*, **Vol.5, No.1-2 (2011), Part I: 159–184. http://www.ijsi.org/1673-7288/5/i80.htm**

1 Introduction: KORSO — Twenty Years Later

Some of the authors of this article know Manfred Broy for more than twenty years. In 1990, in a large national project, KORSO[9], funded by the German government, there were lively discussions about the potential of formal methods in software engineering. Manfred Broy was one of the coordinators of the KORSO project. The discussion about the right semantics for algebraic specifications (be it initial, final or loose) was left behind. The 'loose approach' had gained a lot of supporters, particularly in practical applications. A number of theorem proving and verification tools had emerged (ACL2[27], Coq[4], HOL[58], Isabelle[41], KIV[48], PVS[44], ...), and one of the questions was how much load these could really carry. At that time many researchers agreed that the use of formal methods could be greatly supported by achieving (among others) the following goals:

1. It would be helpful to devise one or two standardized expressive specification languages, that would cover a broad range of applications. These could serve as reference languages the community could rely on and other efforts could be based upon.

Corresponding author: Dominik Haneberg, Email: haneberg@informatik.uni-augsburg.de
Received 2010-11-03; Accepted 2011-01-03; Final revised version 2011-01-17.

2. The use of formal methods in software engineering demands for powerful modeling and verification tools. The major question was, how far these systems could reach, and how easy they would be to apply.

3. At that time only few attempts had been made to integrate stable formal methods into the software engineering process, such that combined tools could be used to certify e.g. commercial products. Roughly at the same time certification bodies designed commonly accepted criteria for safe and secure IT products (ITSEC [23], CC [12]).

20 years later there are a number of industrial strength verification tools around, and there are numerous success stories applying formal verification to large and complex applications, and even more important to commercial products. There are a number of research groups worldwide able to apply formal methods to complex applications on a routine basis. Some examples are:

- formal analysis of programming languages, providing formal semantics for different languages or verifying the type-safety of type systems [63],
- compilation verification for different languages [53,47,40,31]
- verification of hardware processor models, micro kernels, and hypervisors [5,29,30]
- formal security proofs for payment protocols [3,25]

This article highlights such a verification challenge illustrating what can be achieved with formal methods today. The challenge is about the security of the commercial product Mondex, an electronic purse distributed by Mastercard. The challenge has received considerable attention because it is part of the current Grand Challenge No. 6 of the British Computer Society. In this article we present a comprehensive solution to the Mondex challenge ranging from the protocol level to verified Java Card code. However, the message of this article is not just the results of a verification challenge. It also shows, how this project can be turned into a general framework for verifying security protocols for electronic commerce. Furthermore, we show how it can be combined with model-driven software development to form a framework for software and security co-design for e-business applications.

The formalisms and the reasoning in the Mondex study heavily rely on abstract state machines and algebraic specifications, the latter being one of the research areas where Manfred Broy made outstanding contributions. The algebraic specification part of the verification system KIV was heavily influenced by his paper[64], and Manfred Broy is a charismatic teacher bringing together formal methods with software engineering in the elite graduate program Software Engineering in the Bavarian Elite Network. Manfred Broy had not only a strong influence on many recent developments in Computer Science but also had a strong personal influence on some of the authors, possibly more than he is aware of. Therefore, we dedicate this paper to Manfred Broy in honour of his outstanding career as a person and as a scientist.

2 The Mondex Challenge

Mondex smart cards implement an electronic purse. Although they were developed already in the 90's, they are still in use and are now distributed by Mastercard[33]. The typical use of Mondex cards is to replace cash payment in shops by electronic payment. To pay, two cards are inserted into a terminal with two card slots. The amount of money is typed into the terminal and transferred from the customer card to the one of the shop owner. To make payment as simple (or even simpler) than cash payment neither giving credit card information nor a password is necessary. An alternative usage scenario is payment over the internet (or over phone lines) by using two card readers.

Mondex smart cards have become famous for having been the target of one of the first ITSEC evaluations of the highest level E6[13], (now superseded by the highest level EAL7 of the ISO Common Criteria standard[12]) which requires formal specification and verification.

The formal specification and proofs were done in Ref.[61] using the Z specification language[59]. Two models of electronic purses connected by a suitable data refinement theory were defined in Ref.[14]: an abstract one (called level 1 in the following) which models the transfer of money between purses as elementary transactions. Level 1 is used as a simple specification of the main security goal of electronic purses: money should never be created nor destroyed. The second level specifies a communication protocol that can cope with lost messages using a suitable logging of failed transfers such that the main security goal is preserved. The models successfully solve the problem of abstractly specifying a security protocol without specifying the (company-secret) cryptographic primitives used by the real protocol. Instead assumptions about unforgeability of certain messages are used.

To mechanize the security and refinement proofs in Ref.[61] has been proposed as a theorem proving challenge (see Ref.[65] for more information on the challenge and its relation to 'Grand Challenge 6'). We and several other groups have solved this challenge. We describe the Mondex protocol and summarize our main results in Section 3.

The two abstract specifications are still far away from code that is used on an actual smart card. This code has to implement a cryptographic protocol that has to satisfy the assumptions of the abstract model. It also has to cope with the low level communication model used by smart cards. And, of course, the implementation should still satisfy the security goals.

Therefore we have extended the challenge to the systematic formal development and verification of executable code. Our development needs two more refinements, Figure 1 shows an overview. The first refinement described in Section 4 from level 2 to level 3 adds abstract cryptography and a proper attacker model. It results in a proper cryptographic protocol that satisfies the assumptions made on level 2. Level 3 has steps of purses, steps of the terminal as well as steps of a potential attacker, that can use faked cards or can intercept messages sent from one card to another.

The last refinement develops working Java Card code for the steps of the purse. An important aspect of this refinement is the use of a library that encodes data structures as byte arrays. This allows to avoid to clutter the main Mondex code with low-level handling of byte arrays. Section 5 describes the relevant details. The

verification is based on a calculus[60] to verify Java programs in KIV[48].

The Mondex verification effort has been a rather complex project that spanned several person years. It was part of a larger, still ongoing research agenda which develops a software engineering methodology that integrates object-oriented modeling with UML, and formal verification of security properties for various application scenarios (smart cards, mobile devices like smart phones, web services etc.).

In the context of this research agenda, and from experience with several other case studies that had different security goals[18,17,39,20] we have developed a model-driven approach. The approach is based on UML models that are enhanced with security aspects like the definition of an attacker model. Section 6 shows the relevant models for Mondex. The models are given a formal semantics by translating them (automatically) to a formal specification of the cryptographic protocol. The resulting formal specification is that of level 3. The UML models are also used to automatically generate the Java code of level 4. The methodology of this model-driven approach and some of the problems that had to be solved are given in Section 6.2. Another aspect of the model-driven approach is that it opens an alternative to the verification of refinements to code. Instead of proving the refinement from level 3 to 4 individually, it becomes possible to check that the model-to-model transformations used to generate the formal model and the Java code are semantics preserving. Section 6.4 gives an outlook on current work that explores this possibility. Finally, Section 7 concludes.

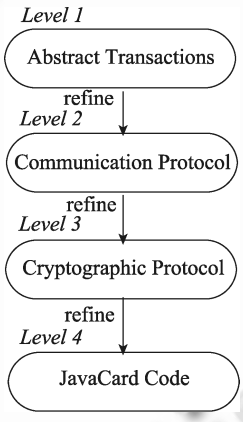

Figure 1. Overview of the Mondex case study

3 The Original Verification Challenge

In this section we describe the Mondex verification challenge.

Rather than using the original Z notation of Ref.[61], we give the relevant operations as rules of an abstract state machine (ASM[19,7]). Such rules are imperative pseudo code over algebraic data types that should be easy to read. We prefer the imperative notation over purely relational notation (an alternative would be e.g. algebraic state machines[10], which have also been used to model security-critical applications[26]), since the control structure of ASM rules can be exploited in our theorem prover KIV[48] to automate proofs by symbolically executing code (by computing strongest postconditions). Otherwise the choice of a formalism to describe the atomic step is largely a matter of taste and tool support.

Informally an ASM is a state machine, that chooses a rule nondeterministically and executes it (atomically) to get to the next state. This informal understanding is sufficient for the following, for a formal semantics see Ref.[7].

The following two subsections give a (slightly simplified) overview over the first two specification levels. Full details on the protocol specification can be found in Ref.[55]. The third subsection discusses the main difficulties of the verification. The final subsection gives and overview over our verification results, the most important one being the discovery of a weakness of the protocol.

3.1 Abstract transactions (Level 1)

The top-level specification of Mondex is as abstract as possible: it just specifies transfer of money between purses. A function balance : Purse $\rightarrow \mathbb{N}$ models the amount of money each purse stores (its "balance"). Transfer from a purse from to another purse to of an amount value just moves the money (which of course must not be larger than balance(from)):

TRANSFEROK
choose from $\neq$ to $\in$ Purse, value $\leqslant$ balance(from) **in**
balance(from) := balance(from) $-$ value
balance(to) := balance(to) + value

Since the transfer of money from one to another purse may fail (due to the card being pulled abruptly from the card reader, due to an attacker sending wrong messages or for internal reasons like lack of memory) the abstract specification has to cope with the fact, that the message transferring money may be lost. One of the key design issues for the protocol defined in the next section will be to ensure, that in this case enough information is saved on the cards such that money lost in such failed transactions is *not truly lost*, but can be recovered.

In the abstract specification money, that is lost but can be recovered is simply modeled by a function lost : Purse $\rightarrow \mathbb{N}$. TRANSFERFAIL moves money to the lost component on the from purse:

TRANSFERFAIL
choose from $\neq$ to $\in$ Purse, value $\leqslant$ balance(from) **in**
balance(from) := balance(from) $-$ value
lost(from) := lost(from) + value

For this simple abstract specification it is trivial to prove that no money is ever lost or generated. The security goal is valid, since the sum of all balance and lost values stays the same in both TRANSFEROK and TRANSFERFAIL.

3.2 The communication protocol (Level 2)

On the second specification level transferring money is done using a protocol with 5 steps. Each step sends a message to one of the two purses involved. Figure 2 shows a sequence diagram with the messages of a successful protocol run. The initial message StartFrom is sent by the terminal to the from purse. The from purse computes its answer in the STARTFROM rule (more details on the rules will be given

below). The answer message StartTo and the following Req(uest) message (computed by the STARTTO rule) authenticate purses, since it is assumed that only authentic Mondex purses can produce these and the following messages. The two messages also fix the amount of money to be transferred. Money is sent from the from to the to purse with the Val(ue) message. Before sending this message the from purse subtracts value from its balance in the REQ rule as indicated in the sequence diagram. When the Val message arrives at the to purse, value is added to its balance in the VAL rule, and a final Ack(nowledge) message is sent to the from purse to indicate a successful transfer of money.

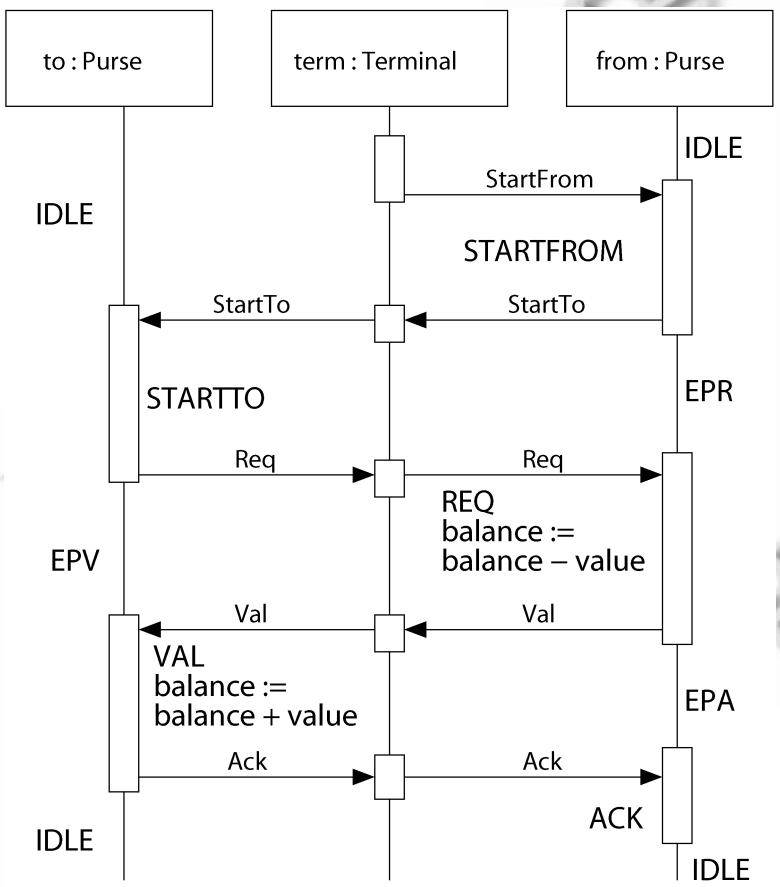

Figure 2. An overview of the Mondex protocol

To execute the protocol, each purse needs a status that indicates how far it has progressed executing the protocol. The possible values are IDLE, EPR, EPV, EPA, (*exp*ecting *r*equest, *v*alue, *a*cknowledge) and the sequence diagram indicates which state a purse is in: both purses start in IDLE state, executing STARTFROM puts the from purse in state EPR etc. Formally, a function state : Purse → status is used to model purse states.

The formal model of the protocol does not model sending and receiving explicitly. Instead all previously sent messages together with all StartFrom messages are collected in a set of messages, called the ether. A purse just picks one of the messages from ether

to receive a message, and adds a new message to ether to send a message. Picking the wrong message from ether models the possibilities of an attacker. It is assumed that an attacker can intercept communication between the two purses (either by using a faked communication terminal, or by intercepting internet communication), so it is possible that the attacker replaces the correct message with any message that was previously sent, or by a publicly available StartFrom message. The other messages StartTo, Req, Val, Ack are assumed to be unforgeable, since they are sent by a Mondex purse that may suitably encrypt them.

To distinguish between a message of an old protocol run and a message of the current run all purses store a sequence number, modeled as a function nextSeqNo : Purse $\rightarrow \mathbb{N}$. This sequence number is incremented in STARTFROM and STARTTO to indicate a new protocol run.

To distinguish messages of the current protocol run from other messages, all encrypted messages store *payment details*. The type PayDetails is a tuple of 5 elements: the names and sequence numbers of the two involved purses and the amount value of money that is transferred. When a purse receives a message, it checks the payment details against payment details stored on the purse. These stored payment details are modeled as a function pdAuth : Purse $\rightarrow$ PayDetails. They are initialized when a new protocol run starts with STARTFROM and STARTTO. The resulting REQ rule is shown in Fig.3 and the other rules are similar (see Ref.[55] for a full listing of all rules).

It remains to explain how a purse handles an unexpected message. As indicated in Fig.3, the protocol run is aborted by calling ABORT. This (sub-)rule takes care to keep enough information to avoid really losing money. The critical message in the protocol is of course the Val message. If this message is lost, then the from purse has subtracted money from its balance, but the to purse has not added it.

```
REQ
    choose from ∈ Purse, msg ∈ ether in
      if state(from) = EPR ∧ msg = Req(pdAuth(from))
      then balance(from) := balance(from) − pdAuth(from).value
           state(from) := EPA
           ether := ether ∪ {Val(pdAuth(from))}
      else ABORT(from)
```

Figure 3. REQ step on level 2

Now, when the from purse aborts, it can detect that it has sent the Val message but that it has not received an Ack, by simply checking its current state to be EPA. Dually the to purse can detect (in state EPV), that it has sent a Req message, but not received a Val.

The solution to the problem therefore is to let both purses write an exception log with the payment details of the current run, if they have to abort in state EPA resp. EPV. Then the Val message has been lost if and only if both purses have written an exception log with the same payment details. In reality, this means that on a failed protocol run, both parties must show their smart card at the bank to check for matching exception logs. Then they could get their money back (this would be

handled by a separate protocol not modeled in the case study). Since aborted protocol runs are rare, this is assumed not to be too inconvenient.

In the formal model, a function exLog : Purse → set(PayDetails) is used for the exception logs and ABORT is specified as

ABORT(purse)
if state(purse) ∈ {EPV, EPA}
then exLog(purse) := exLog(purse) ∪ {pdAuth(purse)}
state(purse) := IDLE
nextSeqNo(purse) := nextSeqNo(purse) +1

The sequence number is incremented to ensure a new protocol run.

3.3 The verification problem

The Mondex case study was originally intended as a challenge of mechanizing and automating proofs. Any correctness proof has to solve the following problems:

- Relating level 1 to level 2 is an instance of *nonatomic refinement*. One abstract transaction is split into several protocol steps (5 for successful execution, $\leqslant 5$ for failed transactions). This is not easily handled by most refinement definitions, as they typically do not allow simulations that verify "1:n"-diagrams directly. Data refinement has simulations with 1:1-diagrams only, where one concrete operation implements one abstract. Other simulations (e.g. refinement in TLA or refinement of IO automata) allow n:1 diagrams, where one concrete operation implements a sequence of n abstract steps, where all but one must be internal (also called "skip","τ" or "stuttering") steps.

- The nonatomic refinement is complicated by the fact that the full scenario allows protocol runs in parallel. This is reflected in the formal model by the possibility of having arbitrary interleaved sequences of protocol steps done by different purses. Even though one purse can only be active in one protocol run, note that intermediate states are not guaranteed to have pairs of communicating purses. A purse p that is in the middle of a protocol run with q can abort the run, and start a new one with another purse r, while q still expects an answer from p.

- As discussed in the previous section, the lost component is implemented as the sum of all values which have matching exception logs. But this simple characterization is true only for quiet states, where all purses are idle. A more complex formula is necessary to characterize lost money in the middle of a protocol run.

Reference [61] already gives proofs on paper for a particular solution to the two problems. The approach uses Z[59] and data refinement[14]. The first problem is solved by adding an explicit SKIP operation (called AbIgnore in Ref.[61]) to the abstract specification. All operations but one implement skip. To have a unique operation that does not implement SKIP, backward simulation is used. This allows to reason backwards in a run of a protocol: its outcome (successful or unsuccessful) can be

used to fix REQ as the one operation that implements both TRANSFEROK as well as TRANSFERFAIL. For this approach proofs are given on 240 pages which are elaborated to the detail of almost calculus level.

3.4 Verification results

The verification challenge was tackled by several groups, the results are published in Ref.[24]. Some groups used the original approach using backward simulation, others used alternative forward simulations, or incremental development of the concrete level using several refinements.

Our verification effort was the first to succeed[56]. We first tried to model the specifications as faithfully as possible in KIV. Z operations were encoded as binary relations. The data refinement theory of Ref.[14] was formalized in KIV, so we could follow the original proof strategy using a backward simulation.

As expected, the proofs on paper still contained a number of small problems in the specification (some properties of the invariant were slightly too weak and some preconditions were missing; see Ref.[21] for full details).

Getting familiar with the case study and doing the proofs required around a month of work. Similar efforts were reported by the other groups, who also found similar bugs.

Our first solution tried to mimic the original as best as possible. This solution was not optimal for KIV, since it did not exploit the support of KIV for ASMs (using Dynamic Logic to express weakest preconditions of imperative code), and for ASM refinement[49,6,51] in particular. ASM refinement was developed with compiler verification in mind, where often m source code instructions are implemented by n assembler instructions. Therefore ASM refinement allows to directly verify arbitrary m:n-diagrams, with data refinement as a special case[52]. We worked out a second solution using the variant of ASM refinement described in Ref.[50]. This solution shows an alternative way to deal with the problem of characterizing lost money. Instead of giving a complex predicate logic formula for intermediate protocol states like the original solution, it gives a simple implicit characterization that says:

In any state: if all purses execute ABORT then the lost money is characterized as the sum of all money in matching exception logs.

This formula can be directly expressed in Dynamic Logic (see Ref.[55] for details). The new approach leads to simpler and more systematic proofs than the original.

For the new approach we had to work out invariants and simulation relations ourselves, so we had to think conceptually about the case study, instead of just imitating proofs. This lead to an important discovery: the original Mondex protocol (as given in Ref.[61]) is susceptible to a denial of service attack. To avoid this attack we modified the protocol slightly. The sequence diagram in Fig.2 of the previous section already shows the improved version, where the StartTo message is an answer to the StartFrom message. In the original protocol both StartFrom and StartTo are sent by the terminal and therefore assumed to be forgeable. The StartFrom message has no answer in the original protocol, it just sets pdAuth(from) and changes the state to EPR. The original protocol allows an attacker with a faked card to first start a protocol with to. Sending the StartTo the attacker can collect the Req answer. Pulling out the fake card would abort the protocol and create an exception log on the to purse. In

a second step the attacker then connects to the from purse and sends both StartFrom and the collected Req. The protocol is then aborted a second time, and a matching exception log is created on the from purse. Running back and forth between the two purses, the attacker is therefore able to fill the purses with matching exception logs. Since both purses can store only a finite number of exception logs in reality, before they refuse to work anymore (one must bring them to the bank), a denial of service attack is possible. Note that the exception logs are created without the two purse owners knowing of each other. It is debatable, how realistic the attack is (the purse owners would probably get suspicious about the failed transfers). Anyway, the attack can be avoided by the modified protocol. For this protocol, the attacker can still connect to the from purse, send StartFrom and collect the StartTo message. Connecting to the to purse, sending the StartTo and collecting the Req is possible too. But sending the Req to the from purse is no longer possible, since the first disconnect from the from purse will reset the protocol (with ABORT). No lost money (as two matching exception logs) will be created, since the from purse aborts in state EPR.

The case study also lead to interesting follow up work: inspired by the incremental developments in Event-B and RAISE/PVS (see again Ref.[24]) we also gave an incremental development of the Mondex protocol[54] that introduces the concepts used one by one. An analysis of the different variations, which protocol step could be chosen as the one that does not implement SKIP is given in Ref.[1].

Summarizing, the Mondex protocol still is an interesting challenge for trying out different refinement and verification strategies.

4 Adding Cryptography

After finishing the original Mondex challenge, we realized that on the way to a secure and error-free Mondex purse, we were only halfway done. In fact even the concrete Mondex level (level 2) assumes security more than proving it. This is due to the fact that the protocol requires that the unforgeability of the Req, Val and Ack messages is somehow ensured by cryptographic means but it is left open how this is to be done.

One simple solution is the introduction of symmetric encryption: We assume that all authentic cards share a secret symmetric key, and encrypt all messages with this key. This is fairly realistic (all smart card protocols from the early 90's use symmetric encryption, but usually with key derivation so that each card has an individual key). It is also possible to use asymmetric encryption. This requires certificates and a key exchange in advance, but the essential part of the protocol remains the same.

Formulating a model of Mondex with the concepts relevant for the verification of cryptographic protocols, e.g. abstract cryptographic operations and an attacker, the third level adds a lot of parts to the formal model compared to the level 2 (Section 3.2).

The additional model elements originate from three sources:

1. An explicit attacker model

2. Relevant concepts of the application domain (e.g. terminals that communicate with smart cards; terminals are implicit in the model of level 2)

3. Abstract cryptography as it is used in the current approaches to verify cryptographic protocols

The Attacker Model The level 3 model follows the general ideas of Paulson's Inductive Approach[46] and other current approaches to modeling and verifying cryptographic protocols by using abstract messages, abstract cryptography and an explicit attacker model. The attacker model, which is not present in level 2, explicitly formalizes the abilities of the attacker with respect to the communication channels (application specific) as well as the cryptography, e.g. decryption only if the appropriate key is available and no reversing of the hash-function (identical for all applications). The formal model of communication on level 3 supports multiple communication channels with different properties for each agent. For example the Mondex terminal has two communication channels for communicating with the to and the from purse. The model of communication on level 3 is more elaborate than the model of communication in other approaches for verifying cryptographic protocols (e.g. Ref.[46]). It uses so called connections to represent communication links between the participants. Each connection link has two ports which represent communication interfaces of the agents. Linked to each port is a queue of messages that were sent to this port via the connection but not yet processed by the agent. Sending a message to a port of an agent corresponds to appending the message to the corresponding list.

Using this explicit model of the possible communication links allows a detailed view on the attacker's access to the communication. Instead of a simple Dolev-Yao model[15] which grants the attacker unlimited access to all communication, we can model communication links that are unavailable or readable only for the attacker. This is more adequate for e.g. smart card applications. Furthermore it is modeled which communication links exist at a given time. E.g. a smart card can communicate with the terminal it is plugged into but with no other agent. The communication links are established (plugging in a smart card) or removed (taking the card out of the reader) by dedicated operations of the ASM which represent the corresponding aspects of the real world problem domain.

The ether of level 2 is replaced by the so called attacker knowledge. This is the set of all messages that the attacker acquired by eavesdropping on communication channels and by decrypting all messages which he can decrypt. The attacker is modeled as another agent besides Mondex cards and terminals and is represented in the ASM by certain rules that formalize his actions. One example is the operation that allows the attacker to send a message that he can produce from his knowledge (attacker-known $\vdash$ msg) to a channel that he can inject messages into (can-send(agent, port, connections)) given the current set of connections. The corresponding rule is:

```
ATTACKER-SEND
  choose msg with attacker-known ⊢ msg in
    choose agent, port with can-send(agent, port, connections) in
      inputs(agent)(port) := inputs(agent)(port) + msg
```

The last line of the rule appends the attacker-generated message (msg) to the queue of unprocessed messages (inputs) of the port that the attacker chose to send something to.

Concepts of the Application Domain In contrast to level 2, a protocol run of the real Mondex application does not start spontaneously between cards. Instead it is triggered by a terminal that has communication channels to two Mondex cards. Also the messages necessary to initiate a protocol run are not all available globally. They have to be created by the terminal by querying the Mondex cards for their current sequence numbers and their name. The third level therefore has additional agents to model these aspects of the real application as well as additional protocol steps to prepare for a new protocol run. Furthermore the internal state of the Mondex cards is made finite e.g. by introducing a maximal length for the exception logs.

Abstract Cryptography Replacing the special Req, Val, and Ack messages by suitably encrypted messages is the most important change compared to level 2. For Mondex symmetric cryptography is used to fulfill the claims of level 2 concerning Req, Val, and Ack. The rule for treating a Req message on this level is shown in Fig.4.

```
REQ
  choose agent ∈ Purse, port
  with inputs(agent)(port) = Req(enc) + rest in
    inputs(agent)(port) := rest
    if state(agent) = EPR then
      if ¬ can-decrypt(key(agent), enc) then ABORT
      else let messagecontent = decrypt(key(agent), enc) in
        if messagecontent = Req(pdAuth(agent)) then
          balance(agent) := balance(agent) − pdAuth(agent).value
          state(agent) := EPA
          outmsg := Val(encrypt(key(agent), Val(pdAuth(agent))))
          choose conn with connected(agent, conn, conns) in
            let rem-agent = other-agent(conn, agent),
                rem-port = other-port(conn, agent) in
              if attacker-can-read(conn) then
                attacker-known := add-and-analyse(outmsg,
                                                  attacker-known)
              inputs(rem-agent)(rem-port) :=
                    inputs(rem-agent)(rem-port) + outmsg
        else ABORT
    else ABORT
```

Figure 4. REQ step on level 3 with cryptography

The rule starts by nondeterministically selecting a Mondex purse agent and a suitable port from which an input message is read. A well-formed Req message is of the form Req(messagecontent), messagecontent being the encrypted data for the transfer. The enclosing Req is used to determine the requested protocol step. The content of the message is encrypted and must equal Req(pdAuth(agent)) in order to be a legitimate message for the current protocol run of the purse. Including the message type identifier in the encrypted part of the messages is necessary to prevent reusing old messages.

After verifying that the purse is in the right state for processing a Req, the purse attempts to decrypt the received messagecontent using a symmetric key that is stored

in the Mondex card (key(agent)). If the payment details contained in the message fit to the current protocol run which is identified by the payment details stored in the purse's state (pdAuth(agent)), the purse subtracts money from its internal balance, changes its state to EPA (expecting acknowledge) and sends the encrypted message that transfers the money. The content of the encrypted message are the payment details of the current protocol run. The created outgoing message is then sent to the agent the purse currently talks to (by adding it to its inputs). Furthermore the attacker is also considered. If the attacker has access to the connection that the purse uses to send its response, the outgoing message is added to the attacker's knowledge and the new knowledge is analyzed if maybe any documents can be decrypted (add-and-analyse(outmsg, attacker-known)). The actions of the attacker are thus distributed over the ASM. The attacker generating and sending documents is captured by a special rule (ATTACKER–SEND), but the attacker eavesdropping is embedded into the rules of the other agents because whenever an agent sends a message, the attacker must have the chance to grab it (if he has access to the connection). Comparing the treatment of a Req message on level 3 and level 2 (cf. Fig.3) shows that dealing with the details of cryptography significantly adds to the size and complexity of the affected code.

The original specification and verification of level 3 used a library of generic messages similar to the ones of Ref.[46]. Experience from different case studies lead us to the specification as it is described in this article. The generic messages are replaced by application specific ones. Besides simplifying the specification this also reduces the verification effort because proofs about the attacker's handling of messages are easier this way. Application specific messages are also used for the Java implementation described in the next section, while we verified the code with generic ones.

Proving the Refinement We use ASM refinement to establish the connection between level 3 and level 2 of Mondex. We proved a forward simulation that guarantees that there is a corresponding run on level 2 for every run that is possible with the level 3 ASM. The required simulation relation has three important properties besides the correspondence of the states of the purses of level 3 and level 2:

- Certain messages (especially StartFrom) must always be contained in the ether because the attacker on level 3 can always generate the corresponding messages. Therefore they must be available on level 2.
- An invariant expressing the well-formedness of the internal state of the purses on the third level and certain conditions concerning the attacker are maintained by the rules of the level 3 ASM.
- The encrypted messages representing Req, Val, and Ack messages that are available to the attacker using his knowledge are exactly those whose corresponding Req, Val, and Ack messages are contained in the ether of level 2. Therefore the ether and the attacker knowledge are equally powerful concerning the critical protocol messages.

The proof of the forward simulation only uses 0:1 and 1:1 diagrams. A successful step on level 3 refines the corresponding step on level 2, processing a malformed or unexpected document refines ABORT. Steps without a counterpart on level 2,

e.g. attacker steps, refine skip as they do not change the state of the purses. Most proof effort deals with the attacker and how a new message influences the set of documents he can generate. This is the central part when proving that ether and attacker knowledge are equally powerful. Proving the simulation and the invariant of level 3 took a student worker with basic experience in verification approximately 4 months and 800 interactive proof steps. Compared to the refinement from level 1 to level 2, in our opinion the refinement to level 3 is conceptually simpler because the invariant is less complex and splitting atomicity is not relevant.

5 Implementing Mondex on Java Smart Cards

5.1 Java Card

The next step is to *implement* the Mondex protocol on a smart card, and to prove the implementation correct and secure. As a platform and programming language we choose Java Card[62]. Java Card is a version of Java tailored for resource-constrained devices such as smart cards. Since these small devices are not very powerful, only a subset of the functionality of Java is supported. Unsupported features are for example garbage collection, threads, dynamic class loading, and reflection. Even the use of large primitive types such as integers or floating point arithmetic is unsupported. The missing garbage collection means that space that is occupied by objects can never be reclaimed even if the object is no longer accessible. To avoid memory leaks best practice is to allocate all memory during the initialization phase, and never to create objects afterwards. On the other hand, all data (objects, arrays, and primitive values) is automatically persistent since the heap is stored in the EEPROM of the smart card. EEPROM sizes range from 16 K to 2 M. Java Card supports transactions so that a protocol step can be executed as an atomic step that either finishes or is not executed at all, even if the smart card is removed from the card reader during execution. A Java Card program runs from the moment it is loaded onto the smart card until it is deleted. It is never restarted.

Smart cards communicate with a terminal by receiving and answering commands using Application Protocol Data Units (APDUs)[22], a sequence of bytes (in Java Card a byte array). Typically, Java Card programs are not written using object-oriented paradigms. Instead, (parts of) byte arrays and primitive data types such as `shorts` are manipulated directly. This leads to low-level programs that are difficult to understand and to debug, even though Java Card programs are usually rather small, typically between 100 and 10000 lines of code.

5.2 Refinement

We proved the correctness of a Java Card implementation by defining a suitable refinement notion. The idea is to refine the protocol agent for the purse to Java Card[18]. This works by stepwise replacement of the level 3 protocol step OP(purse) by a Java implementation, preserving every other part of the ASM. Figure 5 shows the idea. As a result, level 4 is a mixture of steps of agents that are already replaced by a Java program and other agents (the terminal and the attacker) that are still identical to level 3.

The state of a Java program is stored in an algebraic data type store in KIV. A

store can be seen as the state of a JVM, the heap and other runtime information. On level 4 each abstract purse state is replaced by a store initialized with newPurse(). The simulation relation R in Fig.5 maps the Java store to the abstract purse state, while the rest is mapped by identity.

The level 3 ASM step for the purse is replaced by an ASM rule that maps the incoming message for the step on level 3 into the Java store (toJava), executes the Java program (process(msg)), and maps the result computed in the modified store back to the new level 3 state (fromJava). Proving the refinement correct basically means to prove that fromJava returns the same result as OP(Purse) on level 3.

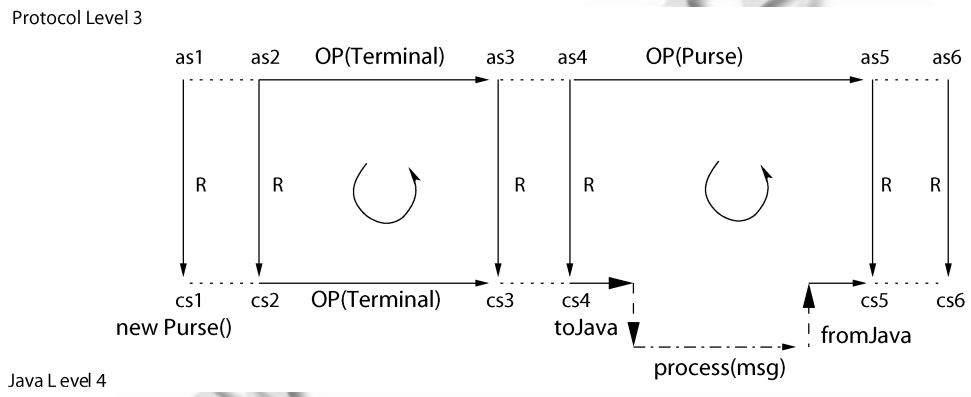

Figure 5. Refinement diagram from level 3 to 4

5.3 The Java Card implementation

Instead of working with byte arrays in the Java Card implementation we use an object-oriented programming style that uses objects for messages and (cryptographic) data types. The implementation of the REQ protocol step is shown in Fig.6. The code is quite readable and basically does the same things as the REQ step in the ASM (decrypt message in line 5, check message content in line 6, 7, reduce balance in line 8, set state to EPA in line 9, create and send Val message in lines 10 – 12). The full protocol implementation has about 600 lines of code.

Communication with the Smart Card and the cryptographic operations are based on byte arrays. The transformation to/from objects is done by a serialization that uses a lightweight ASN1 encoding for objects[16]. An encoding begins with one byte identifying the class of the object followed by the encoding of its fields in the order in which they appear in the class. Arrays require additionally two bytes (encoding a `short` value) for their length. This encoding cannot handle shared or cyclic pointer structures, but is sufficient for the messages in Mondex (and many other cryptographic protocols). The encoding and decoding has about 600 lines of code.

```
1  public void processReq(Req inmsg) {
2   EncDataSymm enc = inmsg.encmess;
3   if (state == State.EPR) {
4      Msgcontent msg = (Msgcontent)
5                        (EncDataSymm.decrypt(sesskey, enc));
6      if ((msg.msgflag == Constants.REQ &&
7           msg.pd.equals(pdAuth))) {
8         balance = Math.minus(balance, pdAuth.value);
9         state = State.EPA;
10        msg = Store.newMsgcontent(Constants.VAL, pdAuth);
11        enc = EncDataSymm.encrypt(sesskey, msg);
12        sendMsg(Store.newVal(enc));
13     } else { ABORT(); }
14  } else { ABORT(); }
15 }
```

Figure 6. Java card code for the REQ step

5.4 Additional attacks on the concrete level

An interesting observation is the fact that when implementing the data types and messages by objects (i.e. pointer structures) there are more possible values on the Java level 4 than on the protocol level 3. One example for this fact are instances of class PurseData with a `null` pointer in the name field. Since Java Card has no Strings the name field has type `byte[]`. Since the value field is the counterpart of the abstract value of the name contained in the abstract PurseData and since `null` does not represent a String (not even the empty String) this document has no counterpart. Additionally, we must take into account that the real communication with a smart card uses sequences of bytes. Given an encoding of messages into bytes there are sequences which do not represent a valid encoding. A refinement that assumes that only valid encodings of object pointer structures with an abstract counterpart occur would not be correct because in the real world an attacker can send malicously flawed messages that may cause crashes, denial of service, or security leaks.

The solution for this problem is to implement a verifier which checks whether the concrete input (as a sequence of bytes) has an abstract counterpart. This verifier is incorporated into the communication layer for decoding an incoming message, and before the actual Purse implementation is called. We have proved a theorem that this verifier never raises an exception, and accepts exactly those inputs that represent abstract messages[16]. Other inputs are treated in the same manner as unexpected abstract messages (in Mondex they are simply ignored). Similar considerations apply to decryption (the result of a decryption in Java Card is a byte array that may or may not represent an abstract value).

5.5 Proof experience

To the best of our knowledge our implementation is the only verified and usable Mondex implementation. Reference [57] implement the level 2 protocol in Java Card and prove some simple properties. However, they use no cryptography, and do not

handle the exception logs correctly.

The proof strategy for the case study is symbolic execution of the Java program[60], extended by the use of lemmata for every method that is called on the way. The proof requires an invariant on the Java store (i.e. properties about the pointer structures used by the Purse). One of the hardest parts was getting this invariant right. It is not really a matter of difficulty, more a matter of complexity, because the state of the Java program and the abstract specification is not trivial. The most important part of the proofs was the formulation of suitable lemmata, which can divide the complexity of the overall case study into smaller parts. In particular, a method called early in the program often ensured properties needed late in the program, but this was not predictable in the first iteration of the proofs. Altogether, over 1700 theorems were formulated and proved for the Mondex refinement from level 3 to the Java level. The full verification required several months.

6 Model-Driven Development of Security Protocols

Our as well as other's experience from Mondex and similar case studies[36,8,11,25] has shown that a model-driven approach is useful to develop secure protocols and implementations. The last part of this paper describes our approach with Mondex as the example and discusses benefits and ideas for future work.

6.1 Modeling security protocols with UML

We use UML to model real-world security protocols[34,37]. UML has the advantage that it provides several graphical notations and is understood by most software engineers. Security protocols are often modeled with sequence diagrams. Sequence diagrams describe the data and sequence of messages that is exchanged between agents but do not capture internal actions like checks or cryptographic operations, or the behavior in case an error occurs. For this reason we additionally use UML activity diagrams that extend the sequence diagrams and describe changes in the internal state of the agents after processing a received message. The activity diagram describes the communication as well as the sequence of actions taken as a result of receiving a message. Another approach is to use UML state machines (e.g. Ref.[25]), but we feel that activity diagrams are more appropriate because the correct handling of a security protocol is more concerned with actions than with state.

Figure 7 shows the part of the activity diagram for a REQ step. For each agent participating in the protocol one swimlane exists in the diagram. (Figure 7 shows only the swimlane for the from purse. The two edges leading to nothing at the top and bottom actually lead to another swimlane.) The from purse receives a Req message, processes it and afterwards returns a Val message. The UML acceptEventAction symbol ① indicates the receiving of a message with instruction Req and data of type EncData that is stored in a variable enc. After reception it is tested whether the agent is in state EPR ②. To model the testing of conditions we use UML decision nodes ③. If the agent was not in state EPR, ABORT() is called ④ which is defined in another activity diagram. It resets the state of a purse and, if necessary, logs the current transaction. Actions and updates are described with UML action nodes containing MEL statements[37], our own language for security protocols. The encrypted part of the Req message is decrypted with a symmetric key and assigned to a variable msg ⑤.

Then it is checked whether we really have a Req message with the correct payment details ⑥. The information that we have a Req message is included in clear text and in the encrypted part so that an attacker cannot forge a message. If this test succeeds the attributes of the from purse are updated, i.e. the balance is decreased ⑦, the state is set to EPA ⑧ and an encrypted Val message is created ⑨. Finally, the message is sent to the terminal, indicated by the UML sendSignalAction ⑩. The ASM rule for the REQ step in Fig.4 is actually generated automatically from this diagram.

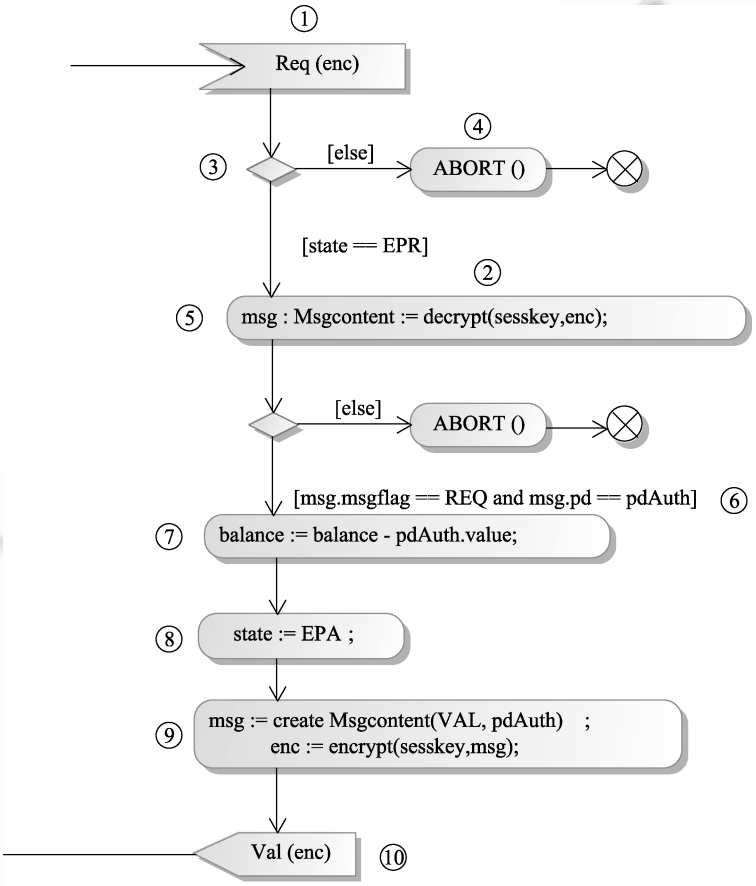

Figure 7. Segment of the Mondex activity diagram for transferring money

The precise structure of the messages, the data involved, and the attributes of the agents are defined in a UML class diagram (Fig.8) extended with stereotypes defined in a profile (this approach is similar to, e.g. Ref.[2]). The agents Purse and Terminal are annotated with stereotypes ≪Smartcard≫ and ≪Terminal≫. All messages exchanged between smart cards and terminals are subclasses of an abstract Message class. Other stereotypes indicate normal classes (≪data≫), a class containing only unique constants (≪Constant≫), or that an attribute or object must be initialized before the smart card can be used (≪initialize≫, all smart cards must have unique names). The stereotypes ≪encrypted≫ at the end of an association and ≪PlainData≫ for a class indicate cryptographic operations. For example, the Req message does not contain the Msgcontent in plain text, but only in encrypted form.

This is denoted by ≪encrypted≫ at the end of the association (in the lower right corner of Fig.8). ≪PlainData≫ indicates that objects of this class will be encrypted. While this stereotype is redundant it helps to avoid simple mistakes. Additionally a predefined library of cryptographic types is used: The purse has an attribute sesskey of the predefined type Symmkey for symmetric encryption. In the activity diagram this key is used to decrypt and encrypt the content of the messages (⑤ and ⑨ in Fig.7). Many constraints apply for a model to be valid. For example, an encrypt operation in the activity diagram may only be applied to objects of classes with stereotype ≪PlainData≫ in the class diagram. Other stereotypes and predefined types deal with asymmetric encryption, digital signatures, nonces, etc.

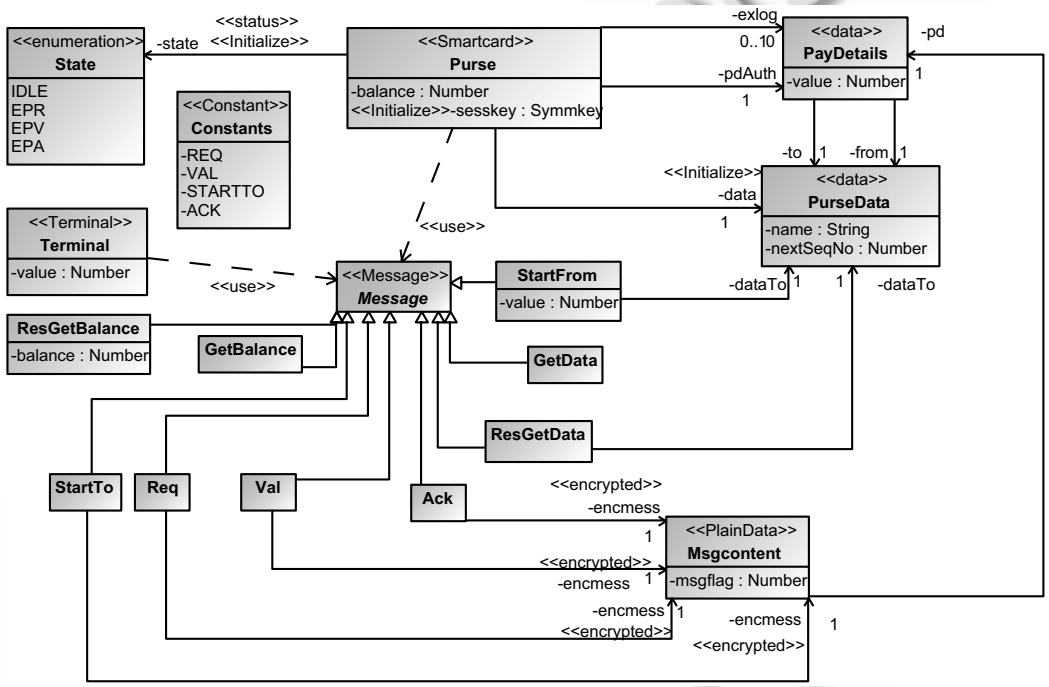

Figure 8. Class diagram for Mondex

To verify cryptographic protocols it is necessary to define the communication infrastructure as well as an attacker model. We model the existing connections with UML deployment diagrams (see Fig.9 for Mondex).

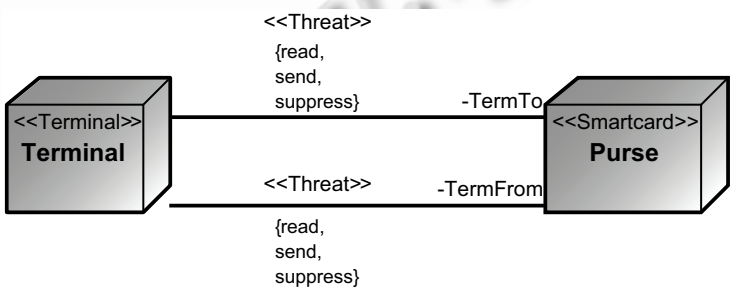

Figure 9. Deployment model for Mondex

The Terminal is needed in reality because a smart card can only communicate with a terminal. In Mondex the terminal simply forwards the messages between the from and the to purse. A ≪Threat≫ describes attacker capabilities. Since we assume that an attacker can read, send, and suppress messages we have modeled a Dolev-Yao attacker[15]. Omitting some of the threats or using other threats allows to model other kinds of attackers with different capabilities. This is an extension to other formal approaches (e.g. Refs.[45, 32]) that use Dolev-Yao exclusively.

6.2 Generating Java Card code

The UML model contains enough information to verify the security of the protocol on an abstract level. It turns out that it is also detailed enough to automatically generate a full, runnable, correct, and secure Java Card implementation using a model-driven approach[36].

The correctness and security considerations presented in the previous section also apply to the generated code. Even more: If the code generation is not flawed the generated code is correct and secure by construction – there is no need to verify the code anymore.

The UML model is transformed into Java Card source code in several steps using a model-driven approach (see Fig.10). All transformations are fully automatic.

In a first step the UML model is transformed into an MEL model (where MEL is our own language for security protocols[37]. The MEL meta model contains the same information as the UML model, but is tailored to our approach. For example, in the UML class diagram all associations must be directed, in the MEL class diagram there is no other possibility. Most of the features of UML class diagrams may not be used, and they simply do not exist in the MEL meta model. On the other hand, the actions in the UML activity diagrams simply contain text, while the text is parsed and transformed into model elements representing an annotated abstract syntax tree in the MEL model. Both models can be seen as platform independent models.

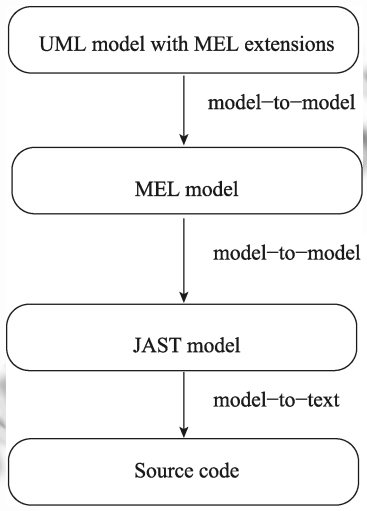

Figure 10. Model-Driven development

The most important transformation step is from the MEL model to the JAST

model. The JAST meta model represents an annotated abstract syntax tree for Java (Card). In this step everything not related to the smart card is discarded, and the full smart card code is generated, in particular:

1. The activities for the smart card are transformed into Java Card methods that handle the individual protocol steps. The code shown in Fig.6 is actually the *generated* code for the REQ protocol step.

2. The Java classes related to the smart card are generated from the MEL class diagram. This includes the message classes and data types used by the card and the messages. The classes correspond exactly to the class diagram. Additionally, two methods are generated: an equal method that (recursively) compares field values and a copy method that (recursively) copies the values from another object into the object itself. These methods mimic the equality and assignment from the UML activity diagram. They are useful because using pointer equivalence makes no sense in protocol implementations, and in Java Card we cannot allocate new objects.

3. An application specific encoding and decoding (de-/serialization) into byte arrays is generated for communication as described in Sect. 5.3. (Java Card does not support reflection, so a generic implementation is not possible.) This is one of the major advantages of the model-driven approach: Programming this conversion by hand for every new application is tedious and error prone. The generator does not care how many classes have to be encoded.

4. A simple memory management (described in more detail below). Since Java Card has no garbage collection (and no possibility to free memory) objects must be reused.

5. The predefined security data types for the cryptographic operations, and a wrapper class for communication.

This model is the platform specific model. In the last step the JAST model is converted into source code. This step is a simple pretty printing of the abstract syntax tree, and completely independent of our approach to model security protocols. Actually, it would be possible to generate byte code directly from the JAST model, but usually one wants to look at the source code.

6.3 Memory management

The absence of garbage collection requires some sort of memory (or better object) management, the Store. An analysis of the activity diagrams yields the number and classes of objects that must be managed by the store. Fields of the smart card class Purse are allocated when the Purse object is created and will be reused or updated. The store manages temporary objects that are needed only during one protocol step. Data that is needed across protocol steps must be stored in fields of the Purse object, and when data from a message is to be stored in a Purse field it must be copied. This is also automatically analyzed and generated, in the activity diagram a simple assignment can be used. Figure 11 shows the part of the store that handles Msgcontent objects. The method initMsgcontent (line 6 – 9) is called once during initialization.

If MC_MAX objects are needed they are stored in an array, i.e. an array of length MC_MAX is created (line 7), and filled with 'empty' objects (line 8). MC_MAX is indeed 2 because at most two Msgcontent objects are used in a single protocol step. A counter mcCount points to the next 'free' object. Two methods newMsgcontent behave more or less like constructors with no arguments and with arguments for all fields, resp. However, the method with arguments (line 12) copies recursively its arguments into the 'free' object (line 15). Before another protocol step is executed all counters are reset.

```
1  public class Store {
2    public final static byte MC_MAX = 2;
3    private static Msgcontent[] mcs;
4    private static byte mcCount = 0;
5
6    private static void initMsgcontent() {
7      mcs = new Msgcontent[MC_MAX];
8      for(short s=0;s<MC_MAX;s++) mcs[s] = new Msgcontent();
9    }
10   static newMsgcontent() { return mcs[mcCount++]; }
11
12   static newMsgcontent(short msgflag, PayDetails pd) {
13     Msgcontent _mc = newMsgcontent();
14     _mc.msgflag = msgflag;
15     _mc.pd.copy(pd);
16     return _mc;
17 }
```

Figure 11. Part of the generated store for Msgcontent

6.4 Future work: Correctness of the Code generation

The model-to-model transformations are implemented in QVT[43], and the model-to-text transformation in XPand[42]. Both are part of the modeling framework of Eclipse, and it is possible to generate the code by clicking on a UML model[35]. Additionally, code for the terminal can be generated. The ASM model in KIV can also be generated by a simple click from the MEL model[38].

The notion of correctness and security of the Java Card code was described in the previous section. To prove that the code generation produces correct and secure code, the following steps are necessary (this is future work):

1. Map a JAST model onto a formal Java semantics specified in KIV. (Actually, this has already been done.)
2. Map the result of the MEL to ASM transformation to KIV's ASMs.
3. Develop a calculus for the verification of QVT programs.
4. Prove that the QVT transformation from MEL to JAST creates Java Card code that is a refinement of the ASM rules created by the MEL to ASM transformation (by arguing on the semantics level).

7 Conclusion

The Mondex case study has been a challenge from various points of view, since it contains a variety of aspects from formal methods and software engineering.

Mondex started out as a challenge to demonstrate that today's theorem provers are powerful enough to handle the proofs of the refinement from level 1 to level 2. For our group as well as several others this turned out to be a moderate effort.

However, the different solutions given by various groups showed a second interesting aspect of the case study: the development of refinement theories that allow an elegant development of the Mondex protocol. The development of an improved refinement theory made us reconsider the security issues of the Mondex protocol, which lead to the discovery of a weakness of the protocol. We could resolve the problem with a small modification.

A third aspect of Mondex was, that its specifications abstracted from cryptography. We solved the question, how to integrate Mondex with abstract cryptography and a proper attacker model by giving a second refinement from level 2 to level 3.

A fourth aspect of Mondex is that one would like not only to prove security for a high-level specification, but for the real code, which has to deal with many additional difficulties compared to the abstract model. We now see Mondex as an important challenge to demonstrate that formal methods can deal with full software development starting with abstract specifications and ending with executable code. As with all large case studies, our theorem prover and in particular our Java calculus have benefited a lot from the proofs of the refinement to Java code. The resulting proofs of the case study as well as the Java Card code can be found online at Ref.[28].

Mondex is a challenge for the application of formal methods. But it is also a challenge for tool-supported software engineering of security-critical applications. We have contributed a model-driven approach that integrates security aspects with object-oriented modeling using UML.

The models can be used to automatically generate readable Java Card code. The Mondex case study has been one important instance to test the solutions we found for the difficulties inherent in this software engineering process.

The UML models can also be used to allow proofs for security properties by translating them to a formal model for our theorem prover KIV. For Mondex this allows to generate the models of level 3 and level 4 automatically. An important question for future work is still, whether it is possible to verify the programmed transformations once and for all, so that verifying refinements from level 3 to 4 becomes unnecessary. Finally, it would also be interesting to see more case studies where an abstract transaction layer like that of Mondex is relevant. In this case it would be interesting to extend the modeling approach to such an abstract layer.

Acknowledgement

Many people contributed to our work on Mondex. We wish to thank Holger Grandy, Bogdan Tofan, Markus Bischof, Robert Bertossi, Marian Borek, and Kuzman Katkalov for work on different aspects of the case study, Jim Woodcock for proposing and promoting the challenge, Egon Börger for pointing us to it and many interesting exchanges on the ASM formalism, many colleagues who also worked on the Mondex

challenge for fruitful discussions, and many anonymous reviewers of the papers for valuable feedback.

References

[1] Banach R, Schellhorn G. Atomic actions, and their refinements to isolated protocols. FAC, 2010, 22(1): 33–61.

[2] Basin D, Doser J, Lodderstedt T. Model Driven Security: From UML Models to Access Control Infrastructures. ACM Trans. on Software Engineering and Methodology, 2006. 39–91.

[3] Bella G, Massacci F, Paulson LC. Verifying the SET purchase protocols. Journal of Automated Reasoning, 2006, 36(1-2): 5–37.

[4] Bertot Y, Casteran P. Interactive Theorem Proving and Program Development. EATCS. Springer, 2004.

[5] Beyer S, Jacobi C, Krning D, Leinenbach D, Paul WJ. Putting it all together - formal verification of the VAMP. STTT Journal, Special Issue on Recent Advances in Hardware Verification, 2005.

[6] Börger E. The ASM Refinement Method. Formal Aspects of Computing, November 2003, 15(1-2): 237–257.

[7] Börger E, Stärk RF. Abstract State Machines—A Method for High-Level System Design and Analysis. Springer-Verlag, 2003.

[8] Breu R, Popp G, Alam M. Model Based Development of Access Policies. International Journal on Software Tools for Technology Transfer, 2007, 9(5): 457–470.

[9] Broy M, Jähnichen S, eds. KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report. LNCS 1009, Springer, Heidelberg, Nov. 1995.

[10] Broy M, Wirsing M. Algebraic State Machines AMAST. LNCS 1816, Springer, 2000. 89–118.

[11] Brucker A, Doser J, Wolff B. A model transformation semantics and analysis methodology for SecureUML. MoDELS 2006: Model Driven Engineering Languages and Systems. LNCS 4199, Springer, 2006.

[12] CCIB. Common Criteria for Information Technology Security Evaluation, Version 3.1 (ISO 15408). Common Criteria Implementation Board, November 2007. Available at http://csrc.nist.gov/cc.

[13] UK ITSEC Certification Body. UK ITSEC SCHEME CERTIFICATION REPORT No. P129 MONDEX Purse. Technical report, UK IT Security Evaluation and Certification Scheme, 1999. http://www.cesg.gov.uk/site/iacs/itsec/media/certreps/CRP129.pdf.

[14] Cooper D, Stepney S, Woodcock J. Derivation of Z Refinement Proof Rules: forwards and backwards rules incorporating input/output refinement. Technical Report YCS-2002-347, University of York, 2002. http://www-users.cs.york.ac.uk/susan/bib/ss/z/zrules.htm.

[15] Dolev D, Yao AC. On the Security of Public Key Protocols. Proc. of the 22th IEEE Symposium on Foundations of Computer Science. IEEE, 1981.

[16] Grandy H, Bertossi R, Stenzel K, Reif W. ASN1-light: A Verified Message Encoding for Security Protocols. Software Engineering and Formal Methods, SEFM. IEEE Press, 2007.

[17] Grandy H, Haneberg D, Reif W, Stenzel K. Developing provably secure M-Commerce applications. In: Müller G, ed. Emerging Trends in Information and Communication Security (ETRICS). LNCS 3995, Springer, 2006. 115–129.

[18] Grandy H, Stenzel K, Reif W. A refinement method for Java programs. Formal Methods for Open Object-Based Distributed Systems (FMOODS). LNCS 4468, Springer, 2007.

[19] Gurevich Y. Evolving algebras 1993: Lipari guide. In Börger E, ed. Specification and Validation Methods. Oxford Univ. Press, 1995. 9–36.

[20] Haneberg D,Reif W,Stenzel K. Electronic-Onboard-Ticketing: software challenges of an State-of-the-Art M-Commerce application. In: Pousttchi K,Turowski K, eds. Mobile Economy - Transaktionen, Prozesse, Anwendungen und Dienste. Proc. zum 4. Workshop Mobile Commerce. Volume P-42 of Lecture Notes in Informatics, 2004.

[21] Haneberg D,Schellhorn G,Grandy H, Reif W. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. Formal Aspects of Computing, January 2008, 20(1).

[22] International Standards Organization. ISO 7816-4 – Identification Cards – Integrated cicuit(s)

cards with contacts – Part 4: Organization, security and commands for interchange, 1995.
[23] ITSEC. Information Technology Security Evaluation Criteria, Version 1.2. Office for Official Publications of the European Communities, June 1991. (available at http://www.bsi.bund.de/zertifiz/itkrit/itsec.htm).
[24] Jones C, Woodcock J. Formal Aspects of Computing. volume 20 (1). Springer, January 2008.
[25] Jürjens J. Secure Systems Development with UML. Springer, 2005.
[26] Jürjens J. Algebraic state machines: Concepts and applications to security. PSI. LNCS 1816, Springer, 2003. 81–92.
[27] Kaufmann M,Manolios P. Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, 2000.
[28] Web presentation of the Mondex case study in KIV.
URL: http://www.informatik.uni-augsburg.de/swt/projects/mondex.html.
[29] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, N orrish M, Sewell T, Tuch H, Winwood S. seL4: formal verification of an OS kernel. Proc. of the 22nd ACM SIGOPS. ACM, 2009. 207–220.
[30] Leinenbach D,Santen T. Verifying the Microsoft Hyper-V Hypervisor with VCC. Proc. of the 2nd World Congress on Formal Methods, FM'09. Springer, 2009. 806–809.
[31] Leroy X. Formal verification of a realistic compiler. Communications of the ACM, 2009, 52(7): 107–115.
[32] Lowe G. Breaking and fixing the Needham-Schroeder Public-Key protocol using FDR. Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS). LNCS 1055, Springer, 1996. 147–166.
[33] MasterCard International Inc. Mondex. http://www.mondex.com.
[34] Moebius N, Haneberg D, Schellhorn G, Reif W. A modeling framework for the development of provably secure E-Commerce applications. International Conference on Software Engineering Advances (ICSEA) 2007. IEEE Press, 2007.
[35] Moebius N, Stenzel K, Grandy H, Reif W. Model-Driven Code Generation for Secure Smart Card Applications. 20th Australian Software Engineering Conference. IEEE Press, 2009.
[36] Moebius N, Stenzel K, Grandy H, Reif W. SecureMDD: a model-driven development method for secure smart card applications. Third International Workshop on Secure Software Engineering, SecSE, at ARES 2009. IEEE Press, 2009.
[37] Moebius N, Stenzel K, Reif W. Modeling Security-Critical Applications with UML in the SecureMDD Approach. International Journal On Advances in Software, 2008, 1(1).
[38] Moebius N, Stenzel K, Reif W. Generating Formal Specifications for Security-Critical Applications - A Model-Driven Approach. ICSE 2009 Workshop: International Workshop on Software Engineering for Secure Systems (SESS'09). IEEE/ACM Digital Libary, 2009.
[39] Moebius N, Stenzel K, Reif W. Formal Verification of Application-Specific Security Properties in a Model-Driven Approach. Proc. of ESSoS 2010 - International Symposium on Engineering Secure Software and Systems. Springer LNCS 5965, 2010.
[40] Moore J. Proving Theorems about Java-like Byte Code. In: Olderog ER, Steffen B, eds. Correct System Design – Recent Insights and Advances. LNCS 1710, Springer, 1999.
[41] Nipkow T, Paulson LC, Wenzel M. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS 2283, Springer, 2002.
[42] Open Architecture Ware. URL: http://www.openarchitectureware.org/.
[43] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2008.
[44] Owre S, Shankar N. A brief overview of PVS. TPHOL 2008. LNCS 5170, Springer, 2008. 22–27.
[45] Lawrence C. Paulson. Inductive analysis of the internet protocol TLS. Technical Report 440, Computer Laboratory, University of Cambridge, December 1997.
[46] Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. Journal of Computer Security, 1998, 6: 85–128.
[47] Pusch C. Proving the Soundness of a Java Bytecode Verifier in Isabelle/HOL. Formal Underpinnings of Java – an OOPSLA'98 Workshop, Vancouver, October 1998. URL: http://www-dse.doc.ic.ac.uk/~sue/oopsla/cfp.html.

[48] Reif W,Schellhorn G,Stenzel K,Balser M. Structured specifications and interactive proofs with KIV. In: Bibel W, Schmitt P, eds. Automated Deduction—A Basis for Applications, volume II: Systems and Implementation Techniques, chapter 1: Interactive Theorem Proving. Kluwer Academic Publishers, Dordrecht 1998. 13–39.

[49] Schellhorn G. Verification of ASM refinements using generalized forward simulation. Journal of Universal Computer Science, 2001, 7(11): 952–979. http://www.jucs.org.

[50] Schellhorn G. ASM refinement preserving invariants. Journal of Universal Computer Science, 2008, 14(12): 1929–1948.

[51] Schellhorn G. Completeness of Fair ASM Refinement. Science of Computer Programming. Elsevier, 2009. (available online).

[52] Schellhorn G. ASM Refinement and Generalizations of Forward Simulation in Data Refinement: A Comparison. Journal of Theoretical Computer Science, May 2005, 336(2–3): 403–435.

[53] Schellhorn G, Ahrendt W. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In: InBibel W, Schmitt P, eds. Automated Deduction — A Basis for Applications. Volume III: Applications, chapter 3: Automated Theorem Proving in Software Engineering. Kluwer Academic Publishers, 1998. 165–194.

[54] Schellhorn G.Banach R. A Concept-Driven Construction of the Mondex Protocol using Three Refinements. Proc. of ABZ 2008. LNCS 5238, Springer, 2008. 57–70.

[55] Schellhorn G, Grandy H, Haneberg D, Moebius N, Reif W. A systematic verification approach for Mondex electronic purses using ASMs. In: Glässer U,Abrial JR, eds. Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis. LNCS 5115, Springer, 2009. 93–110.

[56] Schellhorn G,Grandy H,Haneberg D, Reif W. The Mondex challenge: Machine checked proofs for an electronic purse. In: Misra J, Nipkow T, Sekerinski E, eds. Formal Methods 2006, Proceedings. LNCS 4085, Springer, 2006. 16–31.

[57] Schmitt PH, Tonin I. Verifying the Mondex case study. Software Engineering and Formal Methods, SEFM. IEEE Press, 2007.

[58] Slind K, Norrish M. A brief overview of HOL4. TPHOL. LNCS 5170, Springer, 2008. 28–32.

[59] Spivey JM. The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science. 2nd edition, 1992.

[60] Stenzel K. A formally verified calculus for full Java Card. In: Rattray C, Maharaj S, Shankland C, eds. Algebraic Methodology and Software Technology (AMAST) 2004, Proc. LNCS 3116, Springer, 2004.

[61] Stepney S,Cooper D,Woodcock J. AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000. http://www-users.cs.york.ac.uk/susan/bib/ss/z/monog.htm.

[62] Sun Microsystems Inc. Application Programming Interface Java Card Platform, Version 2.2.1. http://java.sun.com/products/javacard/.

[63] von Oheimb D, Nipkow T. Machine-checking the Java Specification: Proving Type-Safety. In: Alves-Foss J, ed. Formal Syntax and Semantics of Java. LNCS 1523, Springer, 1999.

[64] Wirsing M, Pepper P, Partsch H, Dosch W, Broy M. On hierarchies of abstract data types. Acta Informatica, 1983, 20: 1–33.

[65] Woodcock J. First Steps in the Verified Software Grand Challenge. IEEE Computer, 2006, 39(10): 57–64.