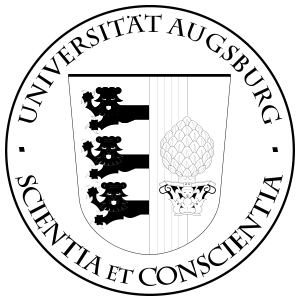
Universität Augsburg



A Refinement Method for Java Programs

Holger Grandy, Kurt Stenzel, Wolfgang Reif

Report 29 2006

Institut für Informatik
D-86135 Augsburg

Copyright © Holger Grandy, Kurt Stenzel, Wolfgang Reif
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

A Refinement Method for Java Programs

Holger Grandy, Kurt Stenzel, Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Augsburg
86135 Augsburg Germany

E-Mail: {grandy, stenzel, reif}@informatik.uni-augsburg.de

Abstract. We present a refinement method for Java programs which is motivated by the challenge of verifying security protocol implementations. The method can be used for stepwise refinement of abstract specifications down to the level of code running in the real application. The approach is based on a calculus for the verification of Java programs for the concrete level and Abstract State Machines for the abstract level. In this paper we illustrate our approach with the verification of a M-Commerce application for buying movie tickets using a mobile phone written in J2ME. The approach uses the interactive theorem prover KIV [1].

1 Introduction

Refinement is an established method for proving algorithms correct. A concrete specification is a refinement of a more abstract specification if every state change that can be performed on the concrete level is also possible on the abstract level. State based refinement methods (e.g. [8] [26] [3]) have been used in numerous case studies for the verification of algorithmic correctness. The underlying theory and the methods for applying those approaches, also on the level of tool support, are elaborated and widely used.

Much less work has been done on refinement methods for the verification of Java implementations. Although there are many examples of Java [14] program verification, e.g. [13] [5] [6] [18] [12], the authors are not aware of a larger case study of interactive verification using a refinement framework for proving functional correctness of a Java program respecting an abstract specification.

In the field of security protocol implementations the past has shown that implementation flaws are very common and can be very subtle. In this paper, we present a general refinement method for Java programs inspired by the challenge of verifying security protocol implementations. The method is illustrated with the verification of a Java M-Commerce application, the Cindy[1] case study. The refinement approach is not limited to the field of security protocols. Using the mechanisms described below we can prove functional correctness for all kinds of programs with input, output and state change.

[1] Cinema Handy (Handy is the German word for mobile phone)

The paper is organized as follows: Section 2 presents the case study, Section 3 illustrates the refinement method and proof obligations. Section 4 describes the mapping of abstract data types to Java classes. Section 5 presents some difficulties the refinement method has to solve stemming from this representation and Section 6 gives some details on the verification of the case study. Finally Section 7 compares the approach to related work and Section 8 concludes.

2 The Cindy Case Study

With Cindy users can buy cinema tickets using mobile phones. A user can order a ticket using a Java application running on the device. Payment can be done using the usual phone bill. After having ordered a ticket it is sent to the mobile phone as a MMS (Multimedia Messaging Service) message. The ticket contains the movie data and an additional unique identifier for the ticket. It can be displayed on the phone using a two-dimensional data matrix barcode and is scanned at the entrance to the cinema directly from the display using a barcode scanner. This kind of application exists e.g. in the Netherlands [2]. Additionally the German railway company, Deutsche Bahn, has recently implemented a similar service for buying train tickets using a mobile phone.

One important question for the cinema is, of course, how to avoid fraud. The idea is simple: Every ticket contains a nonce, a unique random number that is too long to guess. Therefore it is virtually impossible to 'forge' a ticket.

Full details on the abstract model of Cindy as well as the details on the verification of security properties on this abstract level can be found in [10]. The next section describes the approach for verifying an implementation of Cindy running on a mobile phone written in J2ME.

3 The Refinement Method

We assume the reader is roughly familiar with data refinement theory, which in this section we will adopt to Java programs using the notation based on [9].

The abstract level is given as a data type $\mathsf{ADT} = (\mathsf{GS}, \mathsf{AS}, \mathsf{AINIT}, \{\mathsf{AOP_i}\}_{\mathsf{i}\in\mathsf{I}}, \mathsf{AFIN})$ consisting of a set of global states GS and a set of (local) states AS. Total relations $\mathsf{AINIT} \subseteq \mathsf{GS} \times \mathsf{AS}$ and $\mathsf{AFIN} \subseteq \mathsf{AS} \times \mathsf{GS}$ initialize and finalize the data type. $\mathsf{AOP_i} \subseteq \mathsf{AS} \times \mathsf{AS}$ (using an index $\mathsf{i} \in \mathsf{I}$) are the operations possible on the data type. The concrete level is given similarly as $\mathsf{CDT} = (\mathsf{GS}, \mathsf{CS}, \mathsf{CINIT}, \{\mathsf{COP_i}\}_{\mathsf{i}\in\mathsf{I}}, \mathsf{CFIN})$.

Our operations are total so we use the approach of [11] and a forward simulation $\mathsf{R} \subseteq \mathsf{AS} \times \mathsf{CS}$ leading to the following proof obligations for refinement correctness:

- $\mathsf{CINIT} \subseteq \mathsf{AINIT} \fatsemi \mathsf{R}$ ("initialization")
- $\forall\, \mathsf{i} \in \mathsf{I}.\ \mathsf{R} \fatsemi \mathsf{COP_i} \subseteq \mathsf{AOP_i} \fatsemi \mathsf{R}$ ("correctness")
- $\mathsf{R} \fatsemi \mathsf{CFIN} \subseteq \mathsf{AFIN}$ ("finalization")

In the specification of the Cindy example different agents are involved modelling the different protocol participants. Every agent has a type type(agent) (the type can be cellphone, cinema, user or attacker). The index set I of $\mathsf{AOP}_{\mathsf{i}}$ now consists of the different agents, where e.g. $\mathsf{AOP}_{\mathrm{cellphone(n)}}$ denotes the protocol steps of the cellphone agent with number n.

The state as : AS consists of a function astate : agent $\rightarrow \mathsf{A}_{\mathsf{type(agent)}}$ that maps each agent to its internal state in $\mathsf{A}_{\mathsf{type(agent)}}$ (this is e.g. the list of current tickets stored on a phone). Additionally, as contains the current context actxt : context of the communication infrastructure (connections and inputs for every agent that represent the messages that are currently in transit). Together as = astate $\times$ actxt. The global state GS is ignored in AINIT, in AFIN extracts the list of tickets sold so far.

We now refine the cellphone agent type to Java. The approach works by stepwise replacement of an agent type by an implementation for that kind of agent, preserving every other part of the specification.

A concrete state cs : CS is defined as cs = cstate $\times$ cctxt with cctxt : context and cstate : agent $\rightarrow \mathsf{B}_{\mathsf{type(agent)}}$. The context needs to be preserved like in the abstract level because the communication infrastructure is not implementable (it represents the messages currently in transit). The state of a Java program is stored in an algebraic data type called store in KIV. A store can be seen as the equivalent of the heap of a Java virtual machine (in our case the JVM running on a mobile phone). All the runtime information about pointer structures is contained inside the store. Full details on the store and on the Java Calculus implemented in KIV can be found in [22] [21]. On the concrete level the state of a refined agent is now replaced with a store st : store. The state of non-refined agents remains the same as on the abstract level. This means that $\mathsf{B}_{\mathsf{cellphone}} = \mathsf{store}$ and $\mathsf{B}_{\mathsf{agenttype}} = \mathsf{A}_{\mathsf{agenttype}}$ for agenttype $\neq$ cellphone.

The abstract specification of the functionality of the protocol in Cindy is given as an Abstract State Machine [4] consisting of models for all the different agents in the scenario. Abstract State Machines are modeled in KIV using Dynamic Logic (DL). In DL, the formula $\langle\alpha\rangle\ \varphi$ states, that φ holds after the execution of program α. The operation $\mathsf{AOP}_{\mathsf{agent}}$ of the agent agent is given as a DL procedure $\mathsf{APROG}_{\mathsf{agent}}$. Those procedures are all total. So $\mathsf{AOP}_{\mathsf{agent}}$ can be defined as follows:

$$\mathsf{AOP}_{\mathsf{agent}}(\mathsf{astate}, \mathsf{actxt}, \mathsf{astate}', \mathsf{actxt}') \leftrightarrow$$
$$\langle \mathsf{APROG}_{\mathsf{agent}}(\mathsf{astate}, \mathsf{actxt})\rangle\ (\mathsf{astate} = \mathsf{astate}' \wedge \mathsf{actxt} = \mathsf{actxt}')$$

On the concrete level, the steps of a refined agent need a data transformation step from the abstract state into the store. The inputs of the Java agent (given by actxt) need to be refined to Java data types representing the input. The reverse transformation has to be done for the output. This is done by the operations TOSTORE and FROMSTORE. More details on this transformation can be found in section 4. Java method calls are written in the Java calculus in KIV as $\langle st;\ step()\rangle\ \varphi$, which states that formula φ holds after the execution of method

step() in the context of store *st*. Together with TOSTORE and FROMSTORE, we define $\mathsf{COP}_{\mathsf{agent}}$ as:

$$
\begin{array}{l}
\mathsf{COP}_{\mathsf{agent}}(\mathsf{cstate}, \mathsf{cctxt}, \mathsf{cstate}', \mathsf{cctxt}') \leftrightarrow \\
\quad \textbf{if } \neg\, \mathsf{is\_refined}(\mathsf{agent}) \textbf{ then} \\
\qquad \mathsf{AOP}_{\mathsf{agent}}(\mathsf{cstate}, \mathsf{cctxt}, \mathsf{cstate}', \mathsf{cctxt}') \\
\quad \textbf{else } (\exists\, \mathsf{st}, \mathsf{st}'.\ \mathsf{st} = \mathsf{TOSTORE}(\mathsf{cctxt}, \mathsf{cstate}(\mathsf{agent})) \wedge \\
\qquad\qquad\qquad \langle \mathsf{st};\ \mathsf{step}() \rangle\ (\mathsf{st} = \mathsf{st}') \wedge \\
\qquad\qquad\qquad \mathsf{cstate}' = \mathsf{cstate}[\mathsf{agent} \mapsto \mathsf{st}'] \wedge \\
\qquad\qquad\qquad \mathsf{cctxt}' = \mathsf{FROMSTORE}(\mathsf{st}', \mathsf{cctxt}))
\end{array}
$$

An operation of a non-refined agent ($\neg\, \mathsf{is\_refined}(\mathsf{agent})$) is the same as on the abstract level. For the refined agents, the inputs are transformed into Java objects in the store ($\mathsf{TOSTORE}(\mathsf{cctxt}, \mathsf{cstate}(\mathsf{agent}))$). Then a Java method call step() implementing the protocol and starting in this store st must result in a store st′, which is given by cstate′ (cstate′ = cstate[agent ↦ st′]).

Figure 1 gives an overview of the refinement structure. The constructor call of the Java class implementing the protocol is new Protocol(), step() is the Java method call of the protocol implementation.

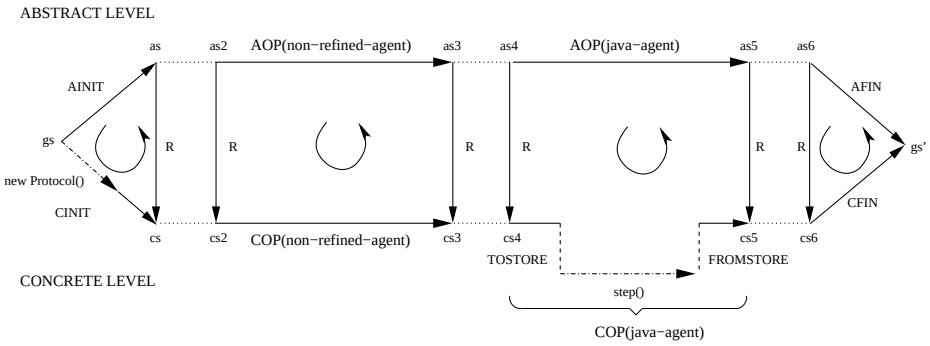


Fig. 1. Refinement diagram

All together the main proof obligation for the refinement of the Java agent schematically looks like this:

$$
\begin{array}{l}
\quad \mathsf{R}(\mathsf{astate}, \mathsf{actxt}, \mathsf{cstate}, \mathsf{cctxt}) \\
\wedge\ \ \mathsf{st} = \mathsf{TOSTORE}(\mathsf{cctxt}, \mathsf{cstate}(\mathsf{agent})) \\
\wedge\ \ \langle \mathsf{st};\ \mathsf{step}() \rangle\ (\mathsf{st} = \mathsf{st}') \\
\wedge\ \ \mathsf{cstate}' = \mathsf{cstate}[\mathsf{agent} \mapsto \mathsf{st}'] \\
\wedge\ \ \mathsf{cctxt}' = \mathsf{FROMSTORE}(\mathsf{st}', \mathsf{cctxt}) \rightarrow \\
\exists\, \mathsf{astate}', \mathsf{actxt}'.\ \mathsf{AOP}_{\mathsf{agent}}(\mathsf{astate}, \mathsf{actxt}, \mathsf{astate}', \mathsf{actxt}') \\
\qquad\qquad\qquad \wedge\ \mathsf{R}(\mathsf{astate}', \mathsf{actxt}', \mathsf{cstate}', \mathsf{cctxt}')
\end{array}
$$

If the retrieve relation holds for two states and the concrete level performs a sequence of TOSTORE, the actual protocol step step() and FROMSTORE, resulting in state $\mathsf{cstate}' \times \mathsf{cctxt}'$, then there must be the possibility to perform a similar step on the abstract level (AOP) which leads to a state $\mathsf{astate}' \times \mathsf{actxt}'$ in which the retrieve relation holds again.

Additionally we have to prove that the constructor call of the Java implementation performs the same initialization steps as AINIT for the refined agent type. This proof obligation is omitted here because it is very similar to the main proof obligation above (excepting TOSTORE and FROMSTORE because there is no input or output for the constructor).

The retrieve relation R between the abstract and the concrete level has to express how the state of the Java program and the abstract state of the protocol ASM relate to each other. The general form of this relation is:

$$
\begin{aligned}
&\mathsf{R}(\mathsf{astate}, \mathsf{actxt}, \mathsf{cstate}, \mathsf{cctxt}) \leftrightarrow \\
&\mathsf{actxt} = \mathsf{cctxt} \wedge \mathsf{AINV}(\mathsf{astate}, \mathsf{actxt}) \wedge \mathsf{CINV}(\mathsf{cstate}, \mathsf{cctxt}) \wedge \\
&(\forall\ \mathsf{agent}.\mathbf{if}\ \mathsf{is\_refined}(\mathsf{agent})\ \mathbf{then}\ \mathsf{extract}(\mathsf{cstate}(\mathsf{agent})) = \mathsf{astate}(\mathsf{agent}) \\
&\qquad \mathbf{else}\ \mathsf{cstate}(\mathsf{agent}) = \mathsf{astate}(\mathsf{agent}))
\end{aligned}
$$

The extract function gets the state of the agent from the store (more precisely it looks at the fields of the classes implementing the protocol and converts those fields back into an abstract state). Then for every refined agent the state on the abstract level (astate(agent)) must be equal to the corresponding value in the store (extract(cstate(agent))). For every agent that is not refined the state must be equal. The context (like the inputs of the agents) must be equal in every case. Additionally we need an invariant on the abstract state (AINV) and an invariant on the concrete state (CINV) that is preserved by every step. The invariants basically state that everything is well-formed and reasonable for our application, e.g. the list of tickets contains only tickets, not other entries.

4 The Mapping to the Concrete Level

Java programs and Abstract State Machines use different internal types. On the one hand we have the Java class hierarchy (consisting of interfaces and classes) and primitive types, on the other hand we have algebraically specified abstract data types and state functions for the state of our ASMs.

For our M-Commerce example same external behavior means sending of the same output messages in reply to the same input messages. On the abstract level input and output are specified using an abstract data type called document. This data type is quite similar to the messages used in [19] or [7]. It is specified algebraically as follows:

$$
\begin{aligned}
\mathsf{document} = {} & \mathsf{intdoc}(.\mathsf{int} : \mathsf{int}) \\
& \mid \mathsf{keydoc}(.\mathsf{key} : \mathsf{key}) \\
& \mid \mathsf{noncedoc}(.\mathsf{nonce} : \mathsf{nonce})
\end{aligned}
$$

| secretdoc(.secret : secret)
| hashdoc(.doc : document)
| encdoc(.key : key; .doc : document)
| sigdoc(.key : key; .doc : document)
| doclist(.list : documentlist)

A document can contain an arbitrary large integer (intdoc). The intdoc type is also used to model arbitrary data since every data can be represented as an integer. Documents can also contain a key (keydoc), a nonce (noncedoc) or a secret (secretdoc). Furthermore a document can be the result of a cryptographic hashing operation (hashdoc) or can be an encrypted document with a certain key (encdoc) or a signature of a document with a certain key (sigdoc). To model composition of messages our document type also contains a type doclist containing a list of other documents. In our ASM model the inputs of all agents are represented as an ASM state function inputs : agent → documentlist (which is a part of the context described in section 3).

On the concrete level a natural representation of the abstract document data type is a class hierarchy which is directly implementing our abstract data type. The Cindy application relies on the security of GSM communication which already supports encryption of all sent messages. Therefore the protocol of Cindy only uses *IntDocs* for modelling the ticket data or concepts like phone numbers, *Noncedocs* for modelling the unique identifier of the ticket and *Doclists* for composing those basic documents to MMS messages.

The class hierarchy we use in the implementation of Cindy is shown in Figure 2. We implement every constructor of the abstract data type document by a separate Java class type for exactly that type of document. For our general refinement approach to security protocols the other document types are implemented as well but omitted here. In addition to input/output behavior we furthermore have to prove that the same state changes are performed on both levels. In the Cindy example the state of the mobile phone consists of a list of documents representing tickets which are currently stored on the phone. This list is specified using the doclist abstract type on the abstract level, respectively implemented by the *Doclist* class for the concrete state. The state function tickets : agent → documentlist specifies this for the abstract level (part of astate(cellphone) as explained in Section 3). In addition the state function inputs : agent → documentlist is relevant for the refinement because it contains the input messages of each agent. Those two functions have to be taken into account for the refinement and have to be transformed to Java data types. Using the abstract data types and the store we define mapping functions for

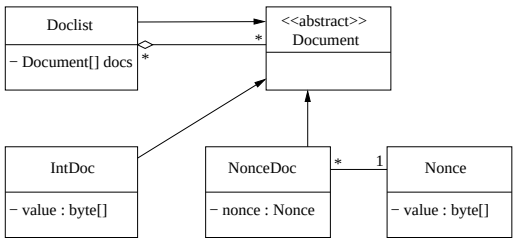


Fig. 2. Document Classes

the transformation of the abstract data type into the concrete pointer structure inside the store and vice versa. The store defines a mapping of keys to values. Store keys are a combination of a reference (a memory address) and a class field or a array index. Getting the value for the field f of the instance at reference r is written as st[r.f]. The lookup for static fields can be written as st[.f]. The value can be a primitive value or a reference to another class instance or an array. The operations for the transformation of documents are called addDoc : document × store → reference × store and getDoc : reference ×store → document (all operations below are specified algebraically). addDoc for e.g. the *IntDoc* class type works as follows:

addDoc-intdoc:
$[r_1, r_2] = \mathsf{newrefs}(2, st) \rightarrow$
  $\mathsf{addDoc}(\mathsf{intdoc}(i), st) =$
    $r_1 \times \mathsf{addobj}(r_1, \mathsf{IntDoc}, .\mathsf{value} \times r_2,$
      $\mathsf{addarray}(r_2, \mathsf{byte\_type}, \mathsf{int2bytes}(i), st))$

Adding an *Intdoc* with value i to the store works by adding an object of class *IntDoc* via the operation addobj : reference × type × fieldvalues× store → store. The reference r_1 of this new object must not be already contained in the store ($[r_1, r_2] = \mathsf{newrefs}(2, st)$). The actual value i of the *Intdoc* is encoded as an array of bytes. This array must also be added to the store via the operation addarray : reference × type × arrayvalues × store → store. The reference r_2 of this array must also be a new reference in the store ($\ldots = \mathsf{newrefs}(2, st)$). The array values are obtained by transforming the integer i to a sequence of bytes (int2bytes(i)). The function addDoc additionally returns the reference r_1 of the *IntDoc* instance as well as the store because we have to know where the new instance is placed inside the store.

The getDoc function for the IntDoc type works the other way:

getDoc-intdoc:
$r \neq \mathsf{null} \wedge st[r.\mathsf{type}] = \mathsf{IntDoc} \rightarrow$
  $\mathsf{getDoc}(r, st) = \mathsf{intdoc}(\mathsf{bytes2int}(\mathsf{getbytearray}(st[r.\mathsf{value}], st)))$

Getting the document of type *IntDoc* ($st[r.\mathsf{type}] = \mathsf{IntDoc}$, where .type is a special field containing the type information of a reference) back from the store is done by first getting the byte array representing the value from the store (getbytearray(st[r.value]). The resulting byte sequence is transformed to an integer using the operation bytes2int and the resulting integer value is used to construct the *Intdoc*.

The operations TOSTORE and FROMSTORE basically use addDoc and getDoc to transform the input messages of the agents into the Java store. Additionally getDoc implements the extract function described in Section 3 in the retrieve relation of the refinement for the list of tickets of an agent. This works because in Cindy both input/output messages and the state are specified using documents.

5 Additional Attacks on the Concrete Level

An interesting observation is the fact that when implementing the data types by pointer structures there are more possible values on the concrete level than on the abstract level. The reason is that on the concrete level there can be pointer structures that do not have any abstract counterpart.

One example for this fact are instances of class *IntDoc* which contain a null pointer in their value field. Since the value field is the counterpart of the abstract value of the integer contained in the *IntDoc* and since null does not represent a number this document has no counterpart. In the following we will call those additional inputs invalid.

A refinement respecting only valid inputs would not be correct because in the real world other inputs than the abstract ones may be given to the program by an attacker and may cause implementation errors or even security leaks.

The solution for this problem is to consider the invalid inputs on the concrete level by implementing a check on the input which checks whether the concrete input has an abstract counterpart. We add an additional document type $\perp$ (representing all the invalid inputs) and specify that the abstract level performs an error treatment (e.g. a reset operation on the internal state) when receiving $\perp$. Then the concrete step which receives an invalid input (and discovers this using the input check) has to be a refinement of the abstract error treatment step. With such a refinement nothing bad can happen on the concrete level when receiving invalid inputs. The TOSTORE operation now relates $\perp$ to all invalid documents. An attacker sending $\perp$ on the abstract level is therefore now able to send any invalid document on the concrete level.

Formally, the predicate $\mathsf{validDoc} : \mathsf{reference} \times \mathsf{store}$ specifies when a pointer structure is a representation of an abstract document. The result $\mathsf{r} \times \mathsf{st}$ of addDoc always satisfies $\mathsf{validDoc}(\mathsf{r}, \mathsf{st})$. The check for valid inputs is done in the receive() method in the Java implementation. Therefore the implementation of receive() must satisfy:

Receive-correct:

$$
\begin{array}{l}
\ldots \textit{// reference r is a valid communication interface in st} \\
\wedge\ \mathsf{st} = \mathsf{st}_0 \rightarrow \\
\langle \mathsf{st};\ \mathsf{r}_0 = \mathsf{r.receive}();\ \rangle \\
\qquad \mathsf{st} = \mathsf{st}_0[.\mathsf{input}, \mathsf{null}] \wedge \\
\qquad ((\mathsf{validDoc}(\mathsf{st}_0[.\mathsf{input}], \mathsf{st}_0) \rightarrow \mathsf{r}_0 = \mathsf{st}_0[.\mathsf{input}]) \wedge \\
\qquad\ (\neg\ \mathsf{validDoc}(\mathsf{st}_0[.\mathsf{input}], \mathsf{st}_0) \rightarrow \mathsf{r}_0 = \mathsf{null}))
\end{array}
$$

If the input is a valid representation (validDoc(...)) of an abstract document, the return value r_0 of receive is the reference which was added in the TOSTORE operation ($\mathsf{st}_0[.\mathsf{input}]$). Otherwise null is returned. Additionally receive sets the input to null ($\mathsf{st}[.\mathsf{input}, \mathsf{null}]$).

It is not desirable to verify the correctness of a concrete input/output checker again for every single application. E.g. all our security protocol implementations use the document class type as the input type. We have used this type for the implementation of Cindy and also e.g. for the implementation of the Mondex [23]

application. Also, a real implementation would not directly send pointer structures but do some kind of encoding (e.g. to byte arrays or XML, which is then sent by MMS). The data checker can be integrated in such a transformation function. We provide an implementation for such a transformation and data check layer which can be verified separately. This enables us to split the refinement proof into two layers. In the first layer the refinement of an abstract specification of the protocol into an implementation working on the document class type is shown using *receive-correct* as an assumption. The second refinement adds the transformation and data check layer. Then TOSTORE has to add an encoding of the input document instead of a pointer structure to the store. The receive method has to check this input and transform it into a pointer structure. Then the property of receive above can be proven using correctness properties of the check and transformation layer.

6 The Cindy Refinement

The interesting part of the abstract specification for the refinement is the step of an agent of type cellphone. An excerpt of this step which performs the protocol step to actually load a ticket on the mobile phone is shown below:

```
CELLPHONE(agent, inputs, tickets){
 let indoc = first(inputs(agent)) in
        inputs(agent) := rest(inputs(agent))
        if is_load_message(indoc) ∧
           #tickets(agent) < MAXTICKETLEN then
         tickets(agent) := tickets(agent) + getPart(2, indoc).data
        else ... // other protocol steps }
```

First a document is taken from the input (CELLPHONE is only called when the input is non-empty) and the list of input messages is shortened. If the input message has the structure of a message to load a ticket (is_load_message(indoc)) and there is space in the list of tickets of the actual agent (#tickets(agent) < MAXTICKETLEN) then the ticket contained in the input document (getPart(2, indoc).data) is added to the list of tickets. The implementation[2] of this protocol step in J2ME on the mobile phone is:

```
public class Protocol {
 private Doclist tickets; // bought tickets
 ...
 public void step(){
  if(comm.available()){
   Document inmsg = comm.receive();
```

[2] This source code is running on any J2ME mobile phone. We have tested it on Nokia 3250 and Sony Ericsson W550i. The receive operation uses the J2ME API to access the MMS messages of the mobile phone.

```
  phoneStep(inmsg);}}

private void phoneStep(Document inmsg) {
 Document originator = inmsg.getPart(1);
 inmsg = inmsg.getPart(2);
 Doclist ticket = getTicket(inmsg);
 if(ticket != null && tickets.len() < MAXTICKETLEN){
  tickets = tickets.attach(ticket);}
 ... //other protocol steps}

private Doclist getTicket(Document indoc) {
 if(indoc != null && indoc.is_comdoc()){
  byte[] ins = indoc.getPart(1).getValue();
  if(ins.length == 1 && ins[0] == LOADTICKET){
   indoc = indoc.getPart(2);
   if(indoc != null && indoc.len() == 2){
    Document indoc1 = indoc.getPart(1);
    Document indoc2 = indoc.getPart(2);
    if(indoc1 != null && indoc1.is_intdoc() &&
       indoc2 != null && indoc2.is_noncedoc()){
     return indoc;}}}}
 return null;}}
```

The method step() is the top-level method for executing a protocol step. First of all the method tests whether input is available. If there is an input the receive method is called and a step is performed via the method phonestep(). This method now tests the structure of the input message with the getTicket() method. getTicket() does some checks regarding the structure of the input document and returns the data part of the input document if it was a valid representation of a ticket and null otherwise. phonestep() then adds the returned data to the list of actual tickets if the input was valid.

Starting with the general proof obligation given by the refinement theory we first symbolically execute the two abstract state machines. The cases for the non-refined agents (such as the attacker) are trivial because they are the same in both specifications. For the refined agent it makes sense to formulate theorems for each Java method which relate the behavior of the method to the abstract counterpart of its input. The corresponding theorem for the load-ticket protocol step is:

$$
\begin{aligned}
&\mathsf{is\_load\_message}(\mathsf{first}(\mathsf{inputs}(\mathsf{agent}))) \wedge \mathsf{st}_1 = \mathsf{store}(\mathsf{agent}) \wedge \\
&\mathsf{st} = \mathsf{TOSTORE}(\mathsf{inputs}, \mathsf{st}_1) \wedge \mathsf{INV}(\mathsf{st}_1) \wedge \ \ldots \\
&\rightarrow \langle \mathsf{st};\ \mathsf{Protocol.step}();\ \rangle \\
&\quad (\mathsf{getDoc}(\mathsf{st}[\mathsf{Protocol.tickets}], \mathsf{st}) = \\
&\qquad \mathsf{tickets}(\mathsf{agent}) + \mathsf{first}(\mathsf{inputs}(\mathsf{agent})) \\
&\quad \wedge\ \mathsf{st}[.\mathsf{input}] = \mathsf{null} \wedge \mathsf{INV}(\mathsf{st}))
\end{aligned}
$$

If the actual input document (first(inputs(agent))) is a correct load message (is_load_message) on the abstract level and if this document is added to the

store via TOSTORE then the step method performs the correct state change: It computes the correct ticket list (the new ticket attached to the old tickets). Also the input was deleted (st[.input] = null). Additionally an invariant that holds before the execution of the method (INV(st)) holds again afterwards.

With such theorems the refinement proof obligation is divisible in different proof obligations for every protocol step. This makes the whole proof feasible. The case study as a whole consists of around 1000 lines of code. The implementation of Cindy itself consists of around 350 lines of code. The rest is the implementation of the document classes and some utility classes (e.g. for handling byte arrays). The verification of the refinement starting with the creation of the concrete and abstract specification of the protocol and ending with the refinement proof took around one and a half man months with KIV. The case study consists of 329 theorems which took 11408 proof steps. 4655 of those steps were done by the user. The degree of automation thereby is nearly 60 %. We expect a much higher degree of automation for upcoming case studies because of the high reusability of the Document implementation and the corresponding library.

7 Related Work

Related work concerning the verification of Java programs was already mentioned in Section 1. Here we focus on related work concerning refinement approaches for security protocols:

[16] describes a similar approach for Java Smart Cards. The authors specify protocols using a high level specification language for proving security properties and a more concrete one which works on the level of byte arrays. They specify lengths and contents of messages using byte arrays and then use static program analysis on the JavaCard implementation to decide whether the implementation is correct. This approach is limited to the very specific class of protocols the specification language allows while our approach allows any abstract specification using all the possibilities of algebraic specifications on KIV [15]. Additionally, because of the automated analysis and the fact that implementation correctness is undecidable this approach cannot give reliable answers in every case.

[24] uses the Spi Calculus for specifying security protocols and a code generation engine to transform this specification to an implementation. They also map messages of the protocol to Java objects. Code generation yields very large implementations that are much less readable than our code and cannot be optimized without losing correctness guarantess. Their mapping to concrete data types is not formally verified and does not address the problem of invalid inputs on the concrete level.

The Mondex [17] case study has recently received a lot of attention because its tool supported verification has been set up as a challenge for today's verification tools [25]. The original refinement proofs using Z have been done on a very detailed level by hand [23]. [20] shows that the same verification can be done with good tool support and in a short period of time using KIV. The

Mondex refinement basically splits a world view of an application into components implementing a protocol. But even the lowest level of the Mondex case study is a only an abstract specification of the communication protocol of the involved parties that does not contain cryptographic operations. The approach presented here can be used to do an additional refinement for Mondex adding a real implementation. This is current work in progress.

8 Conclusion

We presented a refinement method for Java programs instantiating data refinement. The method is based on a calculus for Java verification and Abstract State Machines using the interactive theorem prover KIV. The approach can transfer a security proof for an abstract specification down to running Java code. We have shown how to handle invalid inputs that only exist on the concrete level of Java pointer structures. With Cindy, we have demonstrated that the method is suitable for handling case studies of relevant size. Further work includes the full verification of an implementation of the described input check layer and the application of our approach to other case studies like Mondex.

References

1. Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.
2. *Tickets on your Mobile.* URL: `http://www.beep.nl` [last seen 2006-03-16].
3. C. Bolton, J. Davies, and J.C.P. Woodcock. On the refinement and simulation of data types and processes. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the International conference of Integrated Formal Methods (IFM)*, pages 273–292. Springer, 1999.
4. Egon Börger and Robert F. Stärk. *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
5. C. Breunesse, B. Jacobs, and J. van den Berg. Specifying and verifying a decimal representation in Java for smart cards. In *Proceedings AMAST 2002*, Reunion Island, France, 2002. Springer LNCS 2422.
6. L. Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*. Volume 80 of Electronic Notes in Theoretical Computer Science, Elsevier, 2003.
7. Michael Burrows, Martín Abadi, and Roger M. Needham. A Logic of Authentication. Technical report, SRC Research Report 39, 1989.
8. W. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
9. J. Derrick and E. Boiten. *Refinement in Z and in Object-Z : Foundations and Advanced Applications*. FACIT. Springer, 2001.

10. Holger Grandy, Dominik Haneberg, Wolfgang Reif, and Kurt Stenzel. Developing Provably Secure M-Commerce Applications. In Günter Müller, editor, *Emerging Trends in Information and Communication Security*, volume 3995 of *LNCS*, pages 115–129. Springer, 2006.
11. He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag, 1986.
12. M. Huisman. Verification of java's abstractcollection class: a case study. In *MPC'02: Mathematics of Program Construction*. Springer LNCS 2386, 2002.
13. Bart Jacobs, Claude Marche, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.
14. Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java (tm) Language Specification, Second Edition*. Addison-Wesley, 2000.
15. KIV homepage. `http://www.informatik.uni-augsburg.de/swt/kiv`.
16. R. Marlet and D. Le Metayer. Verification of Cryptographic Protocols Implemented in JavaCard. In *Proceedings of the e-Smart conference (e-Smart 2003)*, Sophia Antipolis, 2003.
17. MasterCard International Inc. *Mondex*. URL: `http://www.mondex.com`.
18. W. Mostowski. Rigorous development of java card applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proceedings, Fourth Workshop on Rigorous Object-Oriented Methods*, London, U.K., 2002.
19. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6, 1998.
20. Gerhard Schellhorn, Holger Grandy, Dominik Haneberg, and Wolfgang Reif. The Mondex Challenge: Machine Checked Proofs for an Electronic Purse. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Formal Methods 2006, Proceedings*, volume 4085 of *LNCS*, pages 16–31. Springer, 2006.
21. Kurt Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.
22. Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, URL: `http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/`, 2005.
23. S. Stepney, D. Cooper, and J. Woodcock. AN ELECTRONIC PURSE Specification, Refinement, and Proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000. URL: `http://www-users.cs.york.ac.uk/$\sim$susan/bib/ss/z/monog.htm`.
24. B. Tobler and A. Hutchison. Generating Network Security Protocol Implementations from Formal Specifications. In *CSES 2004 2nd International Workshop on Certification and Security in Inter-Organizational E-Services at IFIPWorldComputerCongress*, Toulouse, France, 2004.
25. J. Woodcock. Mondex case study, 2006. URL: `http://qpq.csl.sri.com/vsr/\\shared/MondexCaseStudy/`.
26. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.