UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

A modeling framework for the development of provably secure e-commerce applications

Nina Moebius, Dominik Haneberg, Wolfgang Reif, Gerhard Schellhorn

Angaben zur Veröffentlichung / Publication details:

Moebius, Nina, Dominik Haneberg, Wolfgang Reif, and Gerhard Schellhorn. 2007. "A modeling framework for the development of provably secure e-commerce applications." In *International Conference on Software Engineering Advances (ICSEA 2007), 25-31 August 2007, Cap Esterel, France*, edited by Sergiu Dascalu, Petre Dini, Sandro Morasca, Tadashi Ohta, and Andre Oboler, 8. Piscataway, NJ: IEEE. https://doi.org/10.1109/icsea.2007.7.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

A Modeling Framework for the Development of Provably Secure E-Commerce Applications

Nina Moebius, Dominik Haneberg, Wolfgang Reif, Gerhard Schellhorn
Institute of Computer Science, University Augsburg
Department of Software Engineering and Programming Languages
86135 Augsburg, Germany
{moebius, haneberg, reif, schellhorn}@informatik.uni-augsburg.de

***Abstract*—Developing security-critical applications is very difficult and the past has shown that many applications turned out to be erroneous after years of usage. For this reason it is desirable to have a sound methodology for developing security-critical E-Commerce applications. We present an approach to model these applications with the unified modeling language (UML) [1] extended by a UML profile to tailor our models to security applications. Our intent is to (semi-) automatically generate a formal specification suitable for verification as well as an implementation from the model. Therefore we offer a development method seamlessly integrating semi-formal and formal methods as well as the implementation. This is a significant advantage compared to other approaches not dealing with all aspects from abstract models down to code. Based on this approach we can prove security properties on the abstract protocol level as well as the correctness of the protocol implementation in Java with respect to the formal model using the refinement approach. In this paper we concentrate on the modeling with UML and some details regarding the transformation of this model into the formal specification. We illustrate our approach on an electronic payment system called Mondex [10]. Mondex has become famous for being the target of the first ITSEC evaluation of the highest level E6 which requires formal specification and verification.**

I. Introduction

We focus on secure E-Commerce applications such as electronic ticketing or electronic payment systems. In this paper we concentrate on smart card applications. To guarantee the security of these (usually) distributed applications security protocols based on cryptographic primitives are used. Since it is very hard to design such protocols correctly and without errors, we propose to use formal methods for verification.

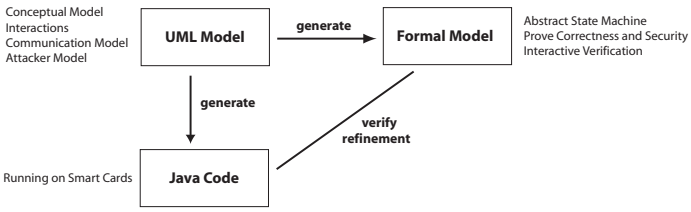

Fig. 1. Overview of the Development Process

Our approach consists of three parts (see Fig. 1):

- The modeling of a security-critical application with UML and the transformation process that generates the formal specification (this paper) as well as code from the UML models
- The formulation and verification of security properties based on the formal model ([2] [3])
- The running code and the proof that the implementation is a refinement of the formal model ([4])

UML describes different views on various parts of a system. There exist several kinds of diagrams emphasizing different aspects of an application. In our approach we use class diagrams to model the static view, i.e. the conceptual model, of an application. To design the protocols resp. to define the interaction steps between the components of the system we use sequence diagrams. To define the processing of messages and internal behavior of components we additionally use activity diagrams. The communication structure of the system and the abilities of the attacker are modeled by deployment diagrams. We only want to model functional behavior, security properties are added on the formal level.

A disadvantage of UML is the lack of a comprehensive semantics directly usable in a verification system. This leads to difficulties for verification of models as well as for generation of code. This is solved by defining a mapping from the semi-formal to a formal presentation using Abstract State Machines (ASM). These have a well-defined and relatively simple semantics [5] [6]. Our formal specification is a combination of algebraic specifications and ASMs. Algebraic specifications are used for the description of the used data types as well as the attacker model. ASMs are used for the protocol dynamics. For verification we use the interactive theorem prover KIV [7]. Our aim is to generate Java resp. JavaCard code for smart card applications. Our group proposes a method to prove that an implementation is a refinement of the abstract formal model [4] by using the Java Calculus [8] [9] implemented in KIV.

The major advantage of our approach with respect to other existing techniques (e.g. [20]) is that we give a method seamlessly integrating modelling, formal methods as well as an implementation.

In this paper we describe the first part of the development process, i.e. the modeling of the application with UML. Our approach is focused on an easy to learn, and intuitive way of building the required models, and abstracts from details of the formal specification or the implementation. We chose an object oriented modeling technique and use a notation that is close to an object oriented programming language.

Our approach provides an opportunity to generate a formal model without paying attention to the specifics of the formal specification which is harder to create and understand than the UML models.
Our technique is illustrated by an electronic payment application called Mondex that is introduced in Section II. In Section III we present our modeling framework. Section IV exemplifies some details of the transformation from UML into the formal model and addresses the abstract data types used in the formal model. Section V addresses related work. Section VI discusses future work and concludes.

II. Case Study Mondex

Mondex [10] cards are smart cards that are used as electronic purses with the aim of replacing coins by electronic cash. Mondex is owned by Mastercard International. The main field of application is the secure transfer of money from one smart card to a second card. To perform a transfer both cards are inserted into a smart card terminal that also acts as user interface. The security properties that have to be verified for Mondex are that no money can be created and any value must be accounted. In detail, this means that no money can be loaded onto a purse without subtracting it from another purse. Furthermore, if a transaction fails, no money should be lost. The Mondex case study recently received a lot of attention because its formal verification has been set up as a challenge for verification tools [11] that several groups [14] [12] work on. For Mondex, several approaches dealing with formal methods and verification exist. But, they are not combined with an engineering discipline for system development. This approach fills the gap between software engineering and formal method techniques.
Each purse has a unique name as well as a sequence number that increases with every transaction. The sequence number is unique concerning a transaction and purse. This avoids replay attacks. While transferring money from one purse to another the information regarding the current transaction is stored in a data type called *pay details*. A *pay detail* contains the name of the purse the money is subtracted from (*from* purse) and the name of the purse the money is added to (*to* purse). Furthermore, a *pay detail* contains the purses current sequence numbers as well as the value that is to be transferred. If a transaction fails (if e.g. the card is removed from the card reader too early) it can happen that the balance of the *from* purse is reduced by the amount currently in transit but the balance of the *to* purse has not yet been increased. In this case, the *pay detail* of the transaction is logged such that the ”lost” money of the *from* purse can be recovered if both cards are presented at the bank and their exception logs are compared. Our implementation (by hand) of this case study can be found in [15].

III. The Modeling Framework

In this Section we present our modeling framework and exemplify the use of our UML profile. UML profiles are a standard mechanism to extend the scope of UML and tailor it to specific domains. A UML profile defines a set of stereotypes, tagged values and constraints.

A. Structure of our Applications

The applications we focus on are security-critical E-Commerce applications. Those usually are distributed systems that consist of different components, which we call agents. In our approach, agents can be devices such as smart cards or mobile phones as well as humans that use the application, for example the customer of a service or any person interacting with the system. Another specifics of our domain is the existence of an attacker. He has to be considered for all analyzed applications. We implicitly assume the existence of the attacker and do not model this agent. The abilities of the attacker, e.g. reading or suppressing of messages, are explicitly modeled for every application (see III-F). Agents can communicate with certain other agents by using connections. Connections are links between two agents, the endpoints of a connection are called ports. If no connection exists between two agents, they cannot exchange messages. For example, the owner of a smart card cannot communicate with his card directly, he has to use a terminal with a smart card reader.

B. Use Case Model

Use cases are used to capture functional requirements of the system in an informal way. As in a traditional software engineering process one or more use cases are written that describe the interaction between the system and external actors or systems. They describe the application in a way that can easily be understood. The use cases are the basis for the sequence and activity diagrams that are used to build the formal model (see below).

C. Conceptual Model

The conceptual model of our application is comparable to a domain model known from object oriented software engineering [16]. It is a class diagram containing a set of conceptual classes, attributes and associations. Some UML elements are annotated with stereotypes that are needed to describe the applications we are interested in.
Every agent of the system is represented by a class that is annotated with category information. The distinction between agent categories is necessary because the generated code and formal model differs depending on the type of agent.
Each agent remembers which messages of a protocol he has already sent and received. This is done by defining a set of states. If, for example, a smart card receives a command it will, depending on its state, accept and process it or repudiate the message. Thus, for each agent the possible states (given by an enumeration) as well as the initial state have to be given. To determine that an enumeration defines the possible states of an agent the association between agent and enumeration is annotated with stereotype ≪state≫, the initial state is specified by a tag `init` which is assigned to exactly one possible state. The defined states can be directly transferred into the code as well as into the formal specification.

It remains to model the attributes and remaining conceptual classes. Since we want to use the conceptual model to generate the formal model as well as the code from it, the question arises which primitive resp. predefined (application independent) data types we use in our models. Typically, in the formal model arbitrary data is represented as a number whereas an implementation uses the Java(Card) primitive data types such as `byte`, `short` and so on. These considerations motivate the following approach: We define a data type `Number` (that represents numbers such as the value of a transaction) as well as a type `Data` that represents application data. Besides Data and Numbers we define some more data types such as `sets`, `lists` as well as certain operations on them. To specify the used data types for the implementation we use a class diagram to define the used Java(Card) types. The data type transformation into the formal model is generated automatically. Our formal model (as well as most formal models that can be found in the literature) uses an abstract application independent data type to specify arbitrary data (see Section IV). This is due to the fact that the formal model emphasizes on a uniform attacker model which can handle messages of all types. The generic data type has the disadvantage that the representation of complex types is hard to read and to create. By giving the possibility to define conceptual classes instead of application independent data types the designer of the protocol does not have to care about the formal representation of data.

In the following we illustrate the conceptual model of Mondex which is given in Fig. 2. In the Mondex application we

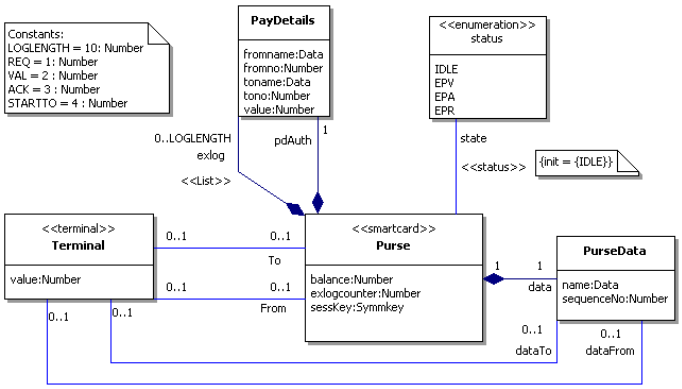

Fig. 2. Conceptual Model of Mondex

have smart cards (that are used as *to* and *from* purse) as well as a smart card terminal used for interaction with the card owner and to forward smart card commands from one card to another. We introduce stereotypes ≪smartcard≫ and ≪terminal≫ for these agent types. A purse may be in state `IDLE`, `EPR` (expecting request), `EPV` (expecting value) or `EPA` (expecting acknowledge), the initial state is `IDLE`. Since the terminal simply forwards messages to the cards and accepts all kinds of messages, it needs no state.

The data of a purse represented by primitive types as e.g. the `balance` are given as attributes of the class. The complex types such as the data of the purse are modeled as concepts. Besides the `PurseData` we define a concept `PayDetails` that records the details of the current transaction. Furthermore, a purse stores its exception log, i.e. a list of *pay details*. The stereotype annotating this composition defines that this list is mapped to the predefined type `List` in the formal model.

D. The Protocol Description

We use sequence diagrams to specify the protocol steps and the flow of information. Besides the information flow the composition and generation of messages that are sent are an essential part of the protocol. In our approach we model these steps in the diagrams by adding comments to the corresponding execution lines of the agents.

A sequence diagram contains one lifeline for each agent participating in the protocol and additionally one lifeline for the ”user”. The user represents the customer of the service and usually initiates a protocol, i.e. sends the first message.

The Mondex protocol is given in Fig. 3. When receiving the *purse data* (method call `getData` in the diagram), the terminal starts the transfer process and sends a `startFrom` message with the data of the *to* purse and the value to be transferred to the *from* purse. The *from* purse stores the received information in the variable `pdAuth` of type `PayDetails` (see Fig. 3, ①). This is modeled by the comment added to the lifeline. Here it is defined how the *pay details* are generated. In a second comment (②) it is specified that the *from* purse generates an encrypted message `encmess` that contains the constant STARTTO together with the generated *pay detail* and is encrypted with the key *sesskey* of the purse. This symmetric key is the same for all purses, see Fig. 2. The protocol is a

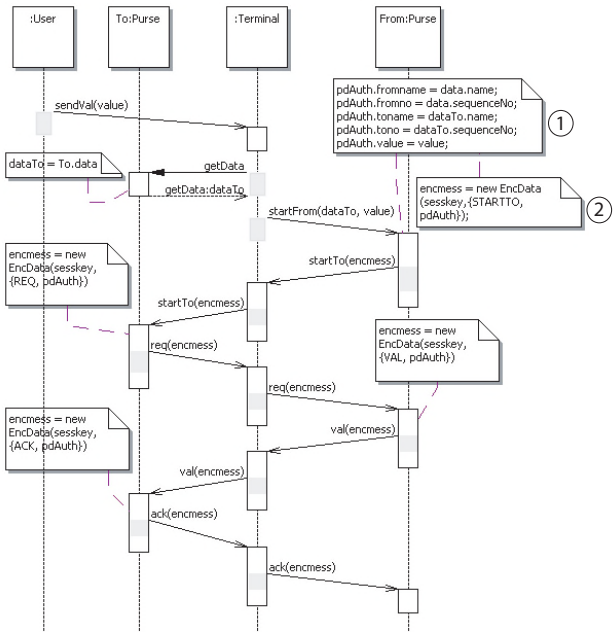

Fig. 3. Protocol Description for Mondex

slight modification of the original protocol given in [11]. The improved version avoids a denial of service attack which is possible in the original version (see [14]).

E. Dynamic Behavior

Sequence diagrams describe the data and sequence of messages that is exchanged between agents but do not capture internal actions or the behavior in case an error occurs. For this reason we additionally use 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. We do not use UML state diagrams because for applications we focus on (with many condition checks) they turned out to be hard to read and confusing.

For each agent participating in the protocol one swimlane exists in the diagram. A protocol can be divided into segments where one segment consists of one protocol step. A protocol step has the following parts: An agent receives a message, performs several tests to check whether the message can be handled and processes the data. Finally, the agent may send a message to another agent.

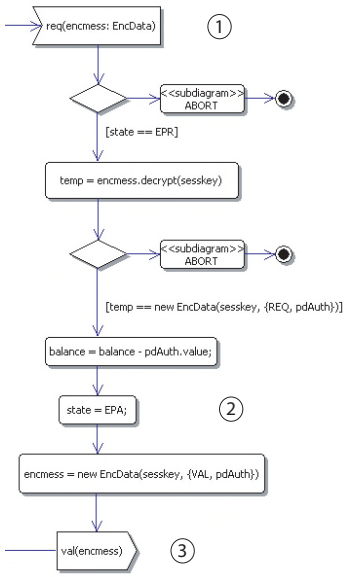

Fig. 4. Segment of the Mondex Activity Diagram for Transferring Money

The segment given in Fig. 4 shows the swimlane of a purse, in this case representing the *from* purse. It receives a `req` message, processes it and afterwards returns a `val` message. The UML acceptEventAction symbol indicates the receiving of a message with instruction `req` and data of type `EncData` that is stored in a (new) variable `encmess` (see Fig. 4, ①). After reception it is tested whether the agent is in state `EPR`. To model the testing of conditions we use UML decision nodes. For better readability and reusability we introduce sub diagrams. For example, a sub-diagram `ABORT` is defined that resets the state of a purse and, if necessary, logs the current transaction. If the agent was not in state `EPR`, the `ABORT` diagram is executed. A reference to a sub-diagram is modeled by an action node that is annotated with stereotype ≪sub-diagram≫ and contains the name of the sub-diagram. If all tests succeed the variables of the *from* purse are updated (②), i.e. the balance is decreased, the state is set to `EPA` and an encrypted message that is going to be sent is created. To denote the updating of values resp. the creation of new variables we use UML action nodes. Finally, the message `val` is sent to the terminal (③). We use the UML sendSignalAction symbol to indicate the sending of messages.

F. Attacker and Communication Model

To verify cryptographic protocols it is necessary to formally specify the communication infrastructure as well as an attacker model. Almost all formal approaches (e.g. [17] [18]) for verifying cryptographic protocols use a rather simple model of communication and the Dolev-Yao threat model.There, no constraints regarding the communication structure are given and it is assumed that the attacker may access all communication links, i.e. he can read all messages sent over that link, suppress them or write messages on that channel. In these approaches (mainly addressing internet protocols) it is ignored that certain agents may not be allowed to communicate directly with other agents. For example, in the Mondex application the owner of the Mondex card cannot directly communicate with his smart card, he has to use a smart card terminal to do so. Also the possibility that some connections are secure against eavesdropping and others are not, is abstracted away. In contrast, our formal model is not limited to Dolev-Yao attackers. This can be seen in the deployment diagram in Fig. 5. There, the communication links between the terminal and the smart cards are considered as secure. The main reason for an attacker model with reduced (but more realistic) abilities is that it becomes possible to have simpler protocols still preserving the desired security properties.

In our approach we explicitly model the existing connections. For each connection we denote if the attacker is able to read or suppress messages and whether he can send messages over that channel. But these annotations do not suffice to describe all possibilities an attacker might have. For example, an attacker in the Mondex scenario could program his own forged smart card. If the protocol has a flaw such that the forged card takes advantage of the weakness of the protocol it may be possible that the attacker gains some information e.g. about secret keys. Indeed we found such a flaw in the original Mondex protocol [14]. Thus, additionally to the annotations made for the connections we introduce a flag that denotes if an agent is forgeable, e.g. if the attacker may use a forged smart card.

UML provides the use of deployment diagrams to define the physical structure of a system. In our approach we use them to describe the communication structure as well as the attacker model of our application.

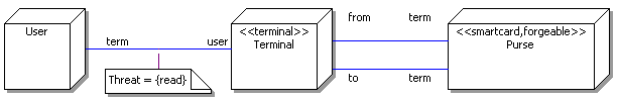

Fig. 5. Deployment Model for Mondex

Fig. 5 shows the deployment diagram for the Mondex protocol. The agents participating in the application are modeled as nodes. If a node is forgeable by an attacker it is annotated with stereotype ≪forgeable≫. For Mondex we assume that the terminal is unforgeable. The existing connections between agents are given by links annotated with the attacker abilities, i.e. read, send and suppress. If an agent sends a message it has to be determined which connection resp. endpoint is used for sending. To be able to reference the endpoints they must have identifiers. If multiple endpoints are available, when sending a message the endpoint that is used has to be denoted in the activity diagram.

IV. The Formal Model

The formal application model consists of two parts: Algebraic specifications and Abstract State Machines.
We use algebraic specifications to define the used data types, the communication between the agents as well as the attacker abilities. To specify the messages exchanged between agents and to define the knowledge of the attacker during a run of the application we use an algebraically specified abstract data type `document`. This abstract type is suitable for a uniform attacker model (that can be used in all applications) whereas the application specific types used in the UML models are suitable for software engineering. There exists two possibilities to solve this mismatch: One can either use an attacker model with application specific data types or translate the application dependent types into abstract types. Since the first possibility is much more complicated we decided to translate the data types defined in the UML models into `documents`. A segment of the type `document` represented as a class diagram that is relevant for the paper is given in Fig. 6. The specification of documents corresponds to the definition of messages given in [19]. A document is either an empty document `emptydoc`,

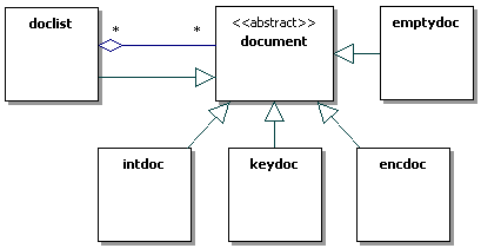

Fig. 6. Segment of the generic data type document

an `intdoc` (containing an integer), a `keydoc` (containing a key), a `noncedoc` (containing a nonce), a `secretdoc` (containing a secret), a `hashdoc` (representing the hash value of a document), an `encdoc` (containing an encrypted document), a `sigdoc` (representing the signature of a document) or a list of documents `doclist`.
The dynamic aspects of the application are given by Abstract State Machines. ASMs are an abstract specification formalism consisting of a set of rules where each rule describes one possible step of one agent. In addition to the agents explicitly defined in the conceptual model the ASM includes rules specifying an attacker as well as the user.
Parts of the formal model are identical in all applications, i.e. the rules determining the current attacker knowledge, connection handling and certain predefined data types. They are taken from a library of formal specifications. The ASM is constructed by using the generic parts contained in the library and by processing the UML diagrams. When generating the formal model the application dependent data types defined in the class diagram have to be mapped to the generic data type `document`. This is done in the following manner: Primitive types represented by the data types `Number` and `Data` in the UML model such as the balance are mapped to the formal document type `intdoc`. All other primitive types such as keys or nonces are mapped to the corresponding document types. All complex data types such as `PayDetails` can be seen as a list of elements where each element is again a predefined or complex data type. A complex data type is mapped recursively: The data type is represented by a `doclist` where each entry represents one element. Each element is mapped in the same way: If it is of type `Number` or `Data`, an `intdoc` is generated, otherwise a `doclist` is created. `Intdocs` are unstructured types that contain arbitrary data. For example, a `PayDetail` is translated into a `Doclist` with five `IntDoc`s that represent the toname, tono, fromname, fromno and value.
Below the main ASM rule for the purse agent is given.

if CHECKINDOC(indoc)
switch(getInstruction(indoc))
  case(STARTFROM): outdoc := STARTFROM(indoc);
  case(REQ): outdoc := REQ(indoc);
  ...
  default: ABORT;
  if outdoc $\neq$ emptydoc
  then SEND(outdoc)

The field `indoc` contains the input document that is analyzed and processed next. An input document `indoc` is a `doclist` with two entries: an `intdoc` that contains the instruction and a `document` representing the data of the message. The indoc is automatically generated from the activity diagram. The method name contained in a send node of the activity diagram is translated into a constant representing the instruction, for example the constant STARTFROM in the code segment. The data part of an indoc is generated by translating the application dependent data to documents as described above. It has to be checked whether the input document has the correct format (CHECKINDOC). For example, if the instruction of the message is REQ the received data must be of type encdoc. The code for these checks can be automatically generated from the UML model. Afterwards, depending on the instruction, the message is processed.
The ASM code for processing a REQ message is given below. It is automatically generated from the activity diagram.

REQ(indoc):

```
if(CHECKREQ(indoc)) then
  balance := balance - pdAuth.value;
  state := EPA;
  outdoc := doclist(VAL +
    encdoc(sesskey, doclist(VAL + pdAuth)));
```

The balance has to be decreased by the value stored in the current *pay details*, the state has to be set to EPA and a `doclist` containing the instruction VAL and an encrypted document with a `doclist` has to be generated. This outgoing message is stored in the field outdoc. The generated formal model is used to verify correctness and security. The security properties are added to the formal specification in KIV.

V. Related Work

The most closely related work is UMLSec developed by Jan Jürjens [20]. As in our approach he proposes to use UML for development of security-critical applications. UMLSec defines a UML profile which adds security-relevant information to the UML diagrams. Security properties are expressed by using stereotypes. Jürjens provides tool support for verifying properties by linking the UML tool to a model checker resp. automated theorem provers. By doing so, the security properties mainly addressed are those that are expressed by the predefined stereotypes. The relevant formal model reflects an abstracted view of parts of the entire system. In our approach we concentrate on a transformation process that generates a formal model of the entire application which can be used for interactive verification of all system aspects. Based on the generated formal model application dependent security properties such as "No money can be created within the Mondex application" can be expressed and proven. In contrast to UMLSec we additionally focus on generation of running Java code as well as the proof that this code is a refinement of the formal model. UMLSec mainly concentrates on the modeling of applications while our approach integrates a verification technique and generation of code.

Another approach dealing with modeling of security-critical applications is Autofocus [21]. Autofocus is a CASE tool for the design and verification of distributed and embedded systems which uses a modeling language similar to UML. The verification is done automatically using a model-checker.

In [22] a framework to simulate UML models is presented. The authors introduce a mapping to translate UML models into Abstract State Machines. Afterwards the models are simulated with the aim of finding inconsistencies in the model. This work analyzes various kinds of applications, the approach is neither tailored to nor optimized for security-critical applications.

VI. Conclusion and Future Work

We presented a method for the modeling of security-critical E-Commerce applications with UML and described how UML models can be semi-automatically translated into a formal model that is used to verify correctness and security of our models. This paper describes one part of our seamless approach to integrate the modeling with UML, formal methods and the actual implementation. We focus on a modeling technique that is easy to learn and abstracts from specifics regarding the formal specification or implementation. One disadvantage of UML is that it is only semi-formally defined. Since in our approach the UML models are translated into Abstract State Machines, we give them a formal semantics. Our technique has evolved over several case studies. E.g. we have analyzed an application where a smart card is used as a copycard for a library. Another case study deals with an application to buy cinema tickets using a mobile phone.

A focus of future research is to finalize our code generation technique and provide tool support for the transformation of the UML diagrams into the formal model as well as into code.

References

[1] *Unified Modeling Language: Superstructure Version 2.0*, The Object Management Group (OMG), 2005.

[2] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif, "Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol," University of Augsburg, Technical Report 2006-32, 2006,

[3] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif, "The Mondex Challenge: Machine Checked Proofs for an Electronic Purse," in *Formal Methods 2006, Proceedings*, ser. LNCS, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer.

[4] H. Grandy, K. Stenzel, and W. Reif, "A refinement method for java programs," University of Augsburg, Technical Report 2006-29, 2006.

[5] Y. Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, Ed. Oxford Univ. Press, 1995.

[6] E. Börger and R. F. Stärk, *Abstract State Machines—A Method for High-Level System Design and Analysis*. Springer, 2003.

[7] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums, "Formal system development with KIV," in *Fundamental Approaches to Software Engineering*, T. Maibaum, Ed. Springer LNCS 1783, 2000.

[8] K. Stenzel, "Verification of Java Card Programs," Ph.D. dissertation, Universität Augsburg, Fakultät für Angewandte Informatik.

[9] K. Stenzel, "A formally verified calculus for full Java Card," in *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, C. Rattray, S. Maharaj, and C. Shankland, Eds. LNCS 3116, 2004.

[10] *Mondex*, MasterCard International Inc., URL: http://www.mondex.com.

[11] J. Woodcock, "First steps in the verified software grand challenge." *IEEE Computer*, vol. 39, no. 10, pp. 57–64, 2006.

[12] C. Jones and J. Woodcock, Eds., *(no title yet)*. Formal Aspects of Computing, 2007, (Journal, to be published).

[13] S. Stepney, D. Cooper, and J. Woodcock, "AN ELECTRONIC PURSE Specification, Refinement, and Proof," Oxford University Computing Laboratory, Technical monograph PRG-126, 2000.

[14] G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif, "A systematic verification Approach for Mondex Electronic Purses using ASMs," Universität Augsburg, Technical Report 2006-27, 2006.

[15] H. Grandy, N. Moebius, M. Bischof, D. Haneberg, G. Schellhorn, K. Stenzel, and W. Reif, "The Mondex Case Study: From Specifications to Code," University a of Augsburg, Technical Report 2006-31, 2006.

[16] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, 2004.

[17] L. C. Paulson, "Inductive analysis of the internet protocol TLS," Computer Laboratory, University of Cambridge, Tech. Rep. 440, 1997.

[18] G. Lowe, "Breaking and fixing the Needham-Schroeder public-key protocol using FDR," in TACAS, Springer, 1996.

[19] L. C. Paulson, "The inductive approach to verifying cryptographic protocols," *Journal of Computer Security*, vol. 6, pp. 85–128, 1998.

[20] J. Jürjens, *Secure Systems Development with UML*. Springer-Verlag, 2005, ISBN: 3-540-00701-6.

[21] F. Huber, S. Molterer, A. Rausch, B. Schatz, M. Sihling, and O. Slotosch, "Tool supported specification and simulation of distributed systems," in *PDSE*, 1998.

[22] A. Cavarra, E. Riccobene, and P. Scandurra, "A framework to simulate uml models: moving from a semi-formal to a formal environment," in *SAC '04*, 2004.