Integrating a Model-Driven Approach and Formal Verification for the Development of Secure Service Applications

Marian Borek, Kuzman Katkalov, Nina Moebius, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel

Abstract We present SecureMDD, a development method for secure service applications that integrates a model-driven approach with formal specification techniques using abstract state machines (ASMs), refinement to code and verification with the interactive theorem prover KIV. A larger case study is used to highlight various aspects of the method with a focus on services and their formal verification.

1 Introduction

Distributed security-critical applications with different communicating components like (Web) services, computers, terminals, or smart cards rely on cryptographic protocols. However, the development of such protocols is notoriously difficult and error prone [2, 52]. This is true even for short protocols with only few communication steps [37]. The reason is the presence of a human attacker who actively tries to break security by eavesdropping and modifying the communication between two components. Often, flaws in an application or in the underlying protocols are detected only after years of usage, e.g., in the Europay–MasterCard–Visa (EMV) protocol [48] used in millions of debit and credit cards or in the Transport Layer Security (TLS) protocol [55].

To be able to develop secure applications based on cryptographic protocols, it is essential to integrate formal verification into the development process. Moreover, the security aspects of the application under development have to be considered in all phases of the development process. SecureMDD is a model-driven development method that realizes both aspects and is tailored to develop security-critical smart card and service applications. Moreover, executable code that is correct and secure with respect to the formal model is generated automatically. This eliminates the problem that buggy implementations of secure protocols often render the application insecure.

M. Borek • K. Katkalov • N. Moebius • W. Reif • G. Schellhorn • K. Stenzel (✉)
Institute for Software and Systems Engineering, Augsburg University, 86135 Augsburg, Germany
e-mail: stenzel@informatik.uni-augsburg.de

© Springer International Publishing Switzerland 2015

B. Thalheim et al. (eds.), *Correct Software in Web Applications and Web Services*,
Texts & Monographs in Symbolic Computation,
DOI 10.1007/978-3-319-17112-8_3

This chapter focuses on (Web) services in SecureMDD. Service-oriented architectures (SOA) are a common way to develop business or e-government applications. Functionalities are deployed as exchangeable services that can be reused and orchestrated to complex systems. Many languages, standards (e.g., Web Services Business Process Execution Language (WS-BPEL), Service-oriented architecture Modeling Language (SoaML), Web Services Description Language (WSDL), Business Process Model and Notation (BPMN)), and approaches [4, 24, 38] exist to develop such systems. Standard security aspects are covered by existing standards such as WS-Security [49] and WS-SecurityPolicy [50] and the use of standard security protocols like TLS [20]. However, using such application-independent standards and protocols is not sufficient to guarantee the security of an application.

Our approach to develop security-critical service applications allows to completely model the whole system with Unified Modeling Language (UML). Thus, in contrast to other approaches (e.g., Business Process Execution Language (BPEL)), an implementation of the modeled application can be generated automatically. The manual implementation of method bodies is not necessary. Moreover, from the UML model of the application, we generate a formal specification based on abstract state machines (ASMs, [15]) for interactive verification of application-specific security properties.

SecureMDD started with smart card applications. Services differ a lot from smart cards because of the different communication abilities, the different behaviors, and the different operational areas. In order to integrate services into SecureMDD, it has to be clarified how services, their communication, and their security are modeled, how a code is generated, how their behavior is specified formally, and how their security is verified. Some of these aspects have already been described in [11]. The contribution of this chapter is a detailed explanation of how to formally specify services and a new case study, the first fully verified SecureMDD application using services.

The rest of the chapter is structured as follows: Section 2 gives an overview of the SecureMDD approach and Sect. 3 introduces the case study. Section 4 describes the formal specification and verification of the example. Section 5 explains the code generation as well as its deployment. Section 6 discusses related work, and Sect. 7 concludes this chapter. The full model of the case study, the formal verification, and the generated code can be found on our Web page.[1]

2 The SecureMDD Approach

SecureMDD is a model-driven development method to create secure applications based on cryptographic (or more broadly speaking, security) protocols. Figure 1 contains an overview.

[1] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/.

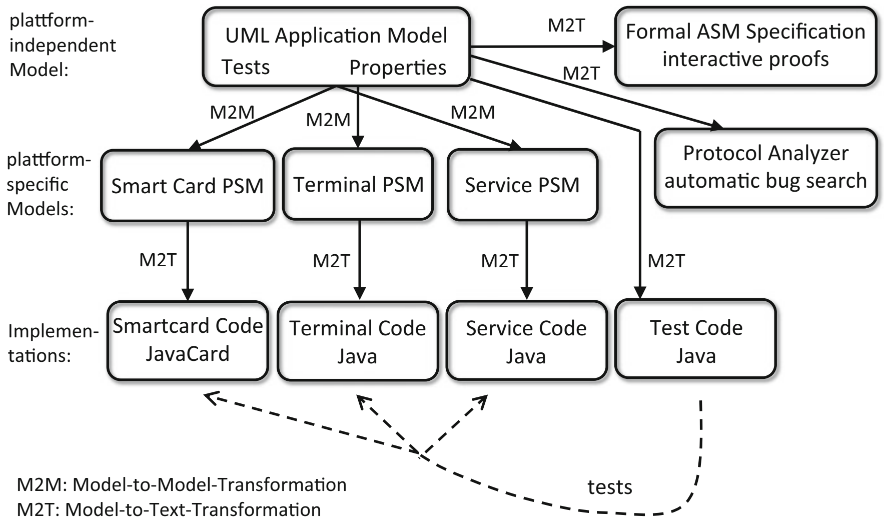

plattform-independent Model:
UML Application Model
Tests
Properties
M2T
Formal ASM Specification interactive proofs
M2T
Protocol Analyzer automatic bug search
M2M
M2M
M2M
plattform-specific Models:
Smart Card PSM
Terminal PSM
Service PSM
M2T
M2T
M2T
M2T
Implemen-tations:
Smartcard Code JavaCard
Terminal Code Java
Service Code Java
Test Code Java
tests
M2M: Model-to-Model-Transformation
M2T: Model-to-Text-Transformation

Fig. 1 Overview of the SecureMDD approach

The development of an application starts with the creation of a platform-independent UML model [42, 44]. This is an abstract view of a system, omitting implementation details. To be able to model security-critical applications, UML was tailored to this domain by defining a UML profile. The static part of the application (i.e., components and their attributes, data, etc.) is modeled with a class diagram, and a deployment diagram models the communication structure. To support the modeling of the dynamic part (the communication protocols and behavior of the components) of an application, we defined a domain-specific language called Model Extension Language (MEL) which is used in UML activity diagrams. With this language, it is possible to make assignments to the attributes of component classes, to create objects, or to call predefined cryptographic operations. The platform-independent UML model of an application consists of all information that is needed to generate executable code as well as a formal model of the whole application automatically. An example is presented in the next section.

Additionally, the model contains tests (functional tests as well as attempts to break the security, i.e., attacks) that are modeled with sequence diagrams [34] and application-specific security properties expressed in OCL [13]. Examples for application-specific properties are the following: *An electronic prescription that is stored on an electronic health card cannot be filled twice in a pharmacy*, *an electronic ticket cannot be forged*, and *no money is lost in an electronic payment system*. In our opinion, application-specific security properties give better guarantees to the security of an application than standard properties like secrecy, integrity, or authenticity. However, standard properties are often prerequisites for proving application-specific properties and, thus, have to be verified as well. Our approach generates a formal specification based on algebraic specifications and

ASMs for the modeled application. The generated formal model is suitable for the theorem prover KIV [5] and is used for interactive verification of the application-specific security properties [45].

Interactive verification often requires substantial effort, and if an error in the protocol is found, the verification has to start again with the corrected model. In order to detect flaws early and efficiently, the application model can be translated into the input language of the automatic protocol analyzer AVANTSSAR [3]. AVANTSSAR systematically generates all possible traces of the system for a fixed number of components and a fixed number of protocol runs and checks for a violation of security properties. From our experience [12], this can find simple bugs fast, but fails for more intricate errors. One problem is that the search space becomes too big, so that AVANTSSAR does not terminate or runs out of memory. Another problem is that some application-specific properties cannot be expressed in AVANTSSAR, so that only a simplified approximation can be checked. Only the interactive verification fully proves the security of the application.

Runnable code for the application can also be generated from the abstract model. The platform-independent model is transformed by model-to-model transformations into three platform-specific models (PSMs), one for the modeled smart card components, one for the terminals and computers, and one for the modeled services. The PSMs contain the relevant information for one component type and add technical details about the implementation. Using the PSMs as input, executable code of the application is generated automatically using model-to-text transformations. For the smart card components, Java Card [28] code is generated. For the terminals and PCs, we generate Java code. Services are implemented as Java Web Services. Additionally, Java test code is generated from the modeled tests and it is used to test the other codes.

Smart cards are small, secure, tamper-proof devices. They are the basis for many security-relevant applications like debit and credit cards, SIM cards, electronic passports, electronic identity cards, access control, etc. The challenges to generate code for these cards in SecureMDD are explained in [43]. This chapter focuses on services.

The security properties that are proved to hold on the formal model should also hold on the code level. To guarantee this, the generated code has to be a refinement of the generated formal model for every modeled application. This requires the proof that the transformations ensure this refinement relation. However, this is research in progress. The first result is the definition of a calculus for QVT [53] (the language used to implement the model-to-model transformations) in KIV. This calculus can be used to prove the correctness of QVT transformations and of generated Java code [59].

The application model can be created with any UML tool that is compatible with Eclipse. All transformations are realized with the Eclipse modeling framework. QVT [53] is used for the model-to-model transformations and XPand [61] for the model-to-text transformations. All artifacts can be generated by a single click in Eclipse.

3 Case Study: Banking

We present a banking case study that uses smart cards, computers, automated teller machines (ATMs), and different services. As the example is concerned with money, it is security critical. This section shows how such a system is modeled in the SecureMDD approach. The next section describes the formal verification of the example, and then the code generation for services is explained.

The banking case study has two use cases:

- A customer can withdraw money at an ATM. The ATM may belong to the customer's bank or to another bank. The ATM communicates with its own bank, and in the second case, the bank owning the ATM communicates with the customer's bank to authorize the withdrawal.
- A customer can transfer money online to another account that can be located at another bank. The customer uses a PC that communicates with an online banking service which in turn communicates with the customer's bank. In case of an inter-bank transfer, the customer's bank then communicates with the other bank.

The main application-specific security property that the banking systems ensures is the following:

> The amount of money in the banking system is constant (if money paid out at ATMs is also counted).

This will be explained in more detail in Sect. 4.6 where the formal specification and verification of this property is described. Next we describe the communication structure of the banking system, then the static view, and finally the protocols.

3.1 Communication Structure

The communication structure and components are defined in a UML deployment diagram (Fig. 2). It consists of nodes, communication paths, and different stereotypes. They are described in turn.

3.1.1 Components

The customer (called `AccountOwner` in Fig. 2) represents a real human being. He/she interacts either with an `ATM` or a `PC` that in turn communicates with an `OnlinebankingService`. Both the `PC` and the `ATM` need the customer's `Debitcard` (the protocols will be explained later in this section). In this scenario, we have two types of banks: `AffiliatedBanks` that operate ATMs and `DirectBanks` that only provide online services. For simplicity, we consider only one affiliated and one direct bank in this example, but multiple instances of services

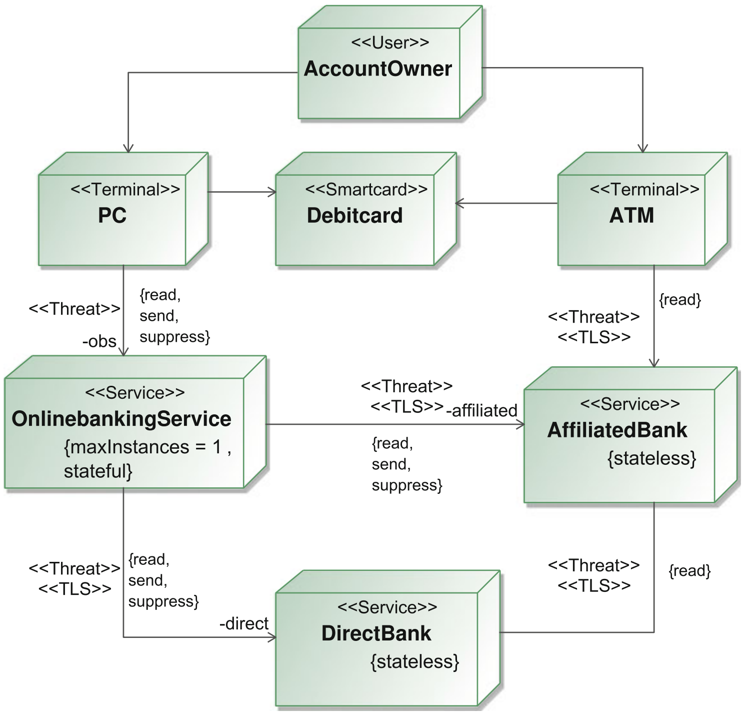


Fig. 2 Deployment diagram for the banking example

are supported in SecureMDD. The online banking service can communicate with both banks, and the two banks can communicate with each other.

The type of each component is indicated by the stereotype in the node. Hence the stereotype `«Service»` shows that `OnlinebankingService`, `DirectBank`, and `AffiliatedBank` are services. A service can be stateful or stateless. `OnlinebankingService` is stateful. This means it can maintain a session and keeps information like a protocol state and a session key for every invoker. This allows to secure messages between `Debitcard` and `OnlinebankingService` with an application-specific cryptographic protocol that uses `PC` only as an intermediate. The other services (`AffiliatedBank` and `DirectBank`) do not need to store session-dependent information, and thus, it is sufficient that they are stateless.

3.1.2 Communication Paths

The communication between components can be unidirectional (indicated by an arrow head) or bidirectional (no arrows). The connection between `AccountOwner` and `ATM` is unidirectional. This means that every action is triggered by the user;

only after an ATM has received an instruction from the user will it become active. The same is true between ATM and Debitcard: The ATM will send a message to Debitcard, and afterward the card can answer, but it cannot send messages of its own accord. In contrast, the connection between DirectBank and AffiliatedBank is bidirectional. This means that any of them can start a communication with the other if it has received a message from an ATM or the OnlinebankingService.

3.1.3 Threats

The communication paths also contain «Threat» stereotypes. They describe attacker capabilities for the connection. He/she can be a full Dolev and Yao [21] attacker who is able to *read*, *send*, and *suppress* messages on the fly, but he/she can also have only a subset of these abilities.

The attacker's abilities and connection security influence each other. If a connection has no applied «Threat» stereotype like the connection between AccountOwner and ATM or ATM and Debitcard, then the connection is assumed to be secure (which must be achieved by physical means). A threat between AccountOwner and ATM would mean that the attacker can observe the personal identification number (PIN) the user types (by shoulder surfing or a hidden camera); a threat between Debitcard and PC could be the result of malware on the PC. Both are not considered in the example. The PC communicates with the OnlinebankingService over the Internet. Here, the full Dolev–Yao attacker is assumed (e.g., a malicious employee of the Internet service provider). The connection does not use TLS (see below) because the protocol will realize an end-to-end encryption between Debitcard and OnlinebankingService which makes TLS unnecessary.

3.1.4 Transport Layer Security

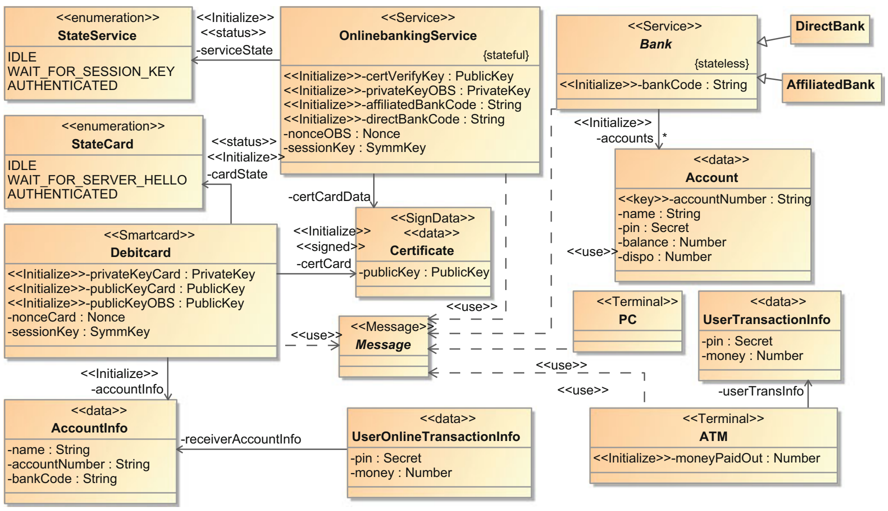
If a connection has an applied «Threat» stereotype with any ability, the connection can be secured with TLS (the standard Transport Layer Security protocol [20], indicated by the «TLS» stereotype) with mutual authentication as the default. This means that both communication partners know each other in advance and authenticate each other when the communication starts. An alternative is server-side authentication where only the server authenticates itself. TLS is used for the communication between OnlinebankingService and the banks. The connection has a «Threat» stereotype with the properties *read, send, and suppress*. That means that the attacker can read, send, and suppress messages on this connection. However, in combination with «TLS», it means that the attacker can only read encrypted messages and that the properties *send* and *suppress* imply that the attacker can only disconnect the connection (see Sect. 4 for more details). The assumption for the connection between DirectBank and AffiliatedBank is that the attacker can only read messages. Thus, the attacker is not able to affect the connection or the transmitted messages.

3.2 *Static View of the Banking System*

A UML class diagram specifies all components with their attributes, data types, and messages. Figure 3 shows the relevant part of the class diagram; only the messages are omitted.

3.2.1 Classes

The component classes are annotated with the stereotypes «`Service`», «`Terminal`», «`Smartcard`» as in the deployment diagram (Fig. 2). The two components `AffiliatedBank` and `DirectBank` are specializations of the abstract `Bank`. The other classes are data types that are either stored in attributes or transmitted in messages or both. Messages and their content are also modeled as classes.[2] Since components send and receive messages, they have a UML



<<enumeration>> StateService
IDLE
WAIT_FOR_SESSION_KEY
AUTHENTICATED
<<Initialize>> <<status>> -serviceState
<<Service>> OnlinebankingService {stateful}
<<Initialize>>-certVerifyKey : PublicKey
<<Initialize>>-privateKeyOBS : PrivateKey
<<Initialize>>-affiliatedBankCode : String
<<Initialize>>-directBankCode : String
-nonceOBS : Nonce
-sessionKey : SymmKey
<<Service>> Bank {stateless}
<<Initialize>>-bankCode : String
DirectBank
AffiliatedBank
<<enumeration>> StateCard
IDLE
WAIT_FOR_SERVER_HELLO
AUTHENTICATED
<<status>> <<Initialize>> -cardState
<<Initialize>> -accounts *
<<data>> Account
<<key>>-accountNumber : String
-name : String
-pin : Secret
-balance : Number
-dispo : Number
-certCardData
<<SignData>> <<data>> Certificate
-publicKey : PublicKey
<<Initialize>> <<signed>> -certCard
<<Smartcard>> Debitcard
<<Initialize>>-privateKeyCard : PrivateKey
<<Initialize>>-publicKeyCard : PublicKey
<<Initialize>>-publicKeyOBS : PublicKey
-nonceCard : Nonce
-sessionKey : SymmKey
<<use>>
<<Message>> Message
<<use>>
<<use>>
<<use>>
<<use>>
<<Terminal>> PC
<<data>> UserTransactionInfo
-pin : Secret
-money : Number
<<Initialize>> -accountInfo
-userTransInfo
<<data>> AccountInfo
-name : String
-accountNumber : String
-bankCode : String
-receiverAccountInfo
<<data>> UserOnlineTransactionInfo
-pin : Secret
-money : Number
<<Terminal>> ATM
<<Initialize>>-moneyPaidOut : Number

Fig. 3 Class diagram of the banking example

[2]All 21 messages can be found on our web page http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/. Some of them are used in the activity diagrams in Figs. 4 and 6.

dependency to the abstract message class `Message` from which all messages are derived.

3.2.2 Cryptographic Data Types

The `OnlinebankingService` and the `Debitcard` employ cryptographic keys (`PublicKey`, `PrivateKey`, `SymmKey`), nonces (i.e., random numbers, `Nonce`), and a certificate to establish a secure communication between each other. These data types are predefined in SecureMDD. The `Certificate` class is annotated with the stereotype «`SignData`» and the association `certCard` with «`signed`». This means that the data is not used as clear text but digitally signed, and the signature is used. Other cryptographic data types are `Secret` for information that the attacker should never know and types and operations for encryption and hashing.

3.2.3 Attributes

The «`Initialize`» stereotype indicates that this attribute or association has to be set during the initial deployment of the system. For example, a `Bank` has a code (attribute `BankCode`) and a list of accounts (association `accounts`) that are fixed since we do not consider opening or closing accounts in this example.

The `Debitcard` contains account information of an account owner who should be the owner of the card. They have no keys, states, or nonces because their communication will be secured with TLS. The `PC` has no attributes because it only forwards the messages between `OnlinebankingService` and `Debitcard`. An `ATM` temporarily stores the PIN and the money that an account owner tries to withdraw in a `UserTransactionInfo` class. Additionally, it also keeps track of how much money it has ever paid out in an attribute `moneyPaidOut`. This information is important to express the main security property mentioned at the beginning (*the amount of money in the banking system is constant*) even though it is not really necessary for the functionality of the system.

3.3 Dynamic View: Protocols and Behavior

The dynamic part of the system is described with activity diagrams. They describe the communication protocols between components and what happens inside a component. The full functionality of the system is defined. All in all, eight activity diagrams are modeled: three communication protocols and five activity diagrams for internal behavior.

3.3.1 Protocol for Withdrawing Money

The activity diagram for withdrawing money from an ATM is shown in Fig. 4. Five components are involved in this scenario: an `AccountOwner`, an `ATM`, a `Debitcard`, an `AffiliatedBank`, and optionally a `DirectBank`. They are modeled as swim lanes. First an `AccountOwner` inserts his/her card into an `ATM` slot, chooses "withdraw money", and enters his/her PIN as well as the sum of money to withdraw on the user interface of the `ATM` (1). The interaction of a real human with the ATM is modeled as a message `UDebit` with argument `userTransactionInfo` (an instance of the class `UserTransactionInfo`) that is sent to the `ATM`. Message passing is indicated by UML SendEvent and AcceptEvent nodes. The `ATM` stores this information in its attribute `userTransInfo`, asks the `Debitcard` for the account information (2), and sends all data to its owner, the `AffiliatedBank` (3). The bank first checks if the account belongs to itself or another bank by comparing the received bank code with its own. This is modeled with a UML decision node and guards. If the account belongs to the bank itself, it calls the `debitFunction` (4) and then, depending on the return value, the money is issued or not. The fork symbol in the node indicates a reference to another activity diagram where the behavior of `debitFunction` is modeled, and the flow final node indicates abortion of the protocol. `debitFunction` debits an account if everything is ok (PIN correct and credit line not exceeded). If the account belongs to another bank, the message is forwarded to the `DirectBank` (5) and the amount will be debited there. After that, if the debit action was successful, a message `TerminalPayOut` is sent back (6) and the `ATM` pays out the money (7).

3.3.2 Details About Protocols in SecureMDD

The actions and guards contain statements of our MEL language. `x := y` is an assignment and `b : Boolean := ...` a local variable declaration. Static analysis will ensure that all identifiers exist in the current scope of the class of the swim lane and that everything is type correct w.r.t. the class diagram.

The protocol uses no cryptography because all communications are assumed to be secure against an attacker as specified in the deployment diagram (Fig. 2). The ATM counts how much money it issues with `moneyPaidOut := moneyPaidOut + money`. This is not necessary for the protocol, but needed in order to express the security property.

Smart cards, services, and terminals are very different in reality, but there is almost no difference in their treatment in the activity diagram. This is the idea of model-driven development—the model abstracts from technical details. Only the class and deployment diagrams distinguish the components by applying different stereotypes. However, this is not the complete truth. The modeler must be aware that a smart card has very limited resources and cannot be used to store data in the same manner as a service or terminal.

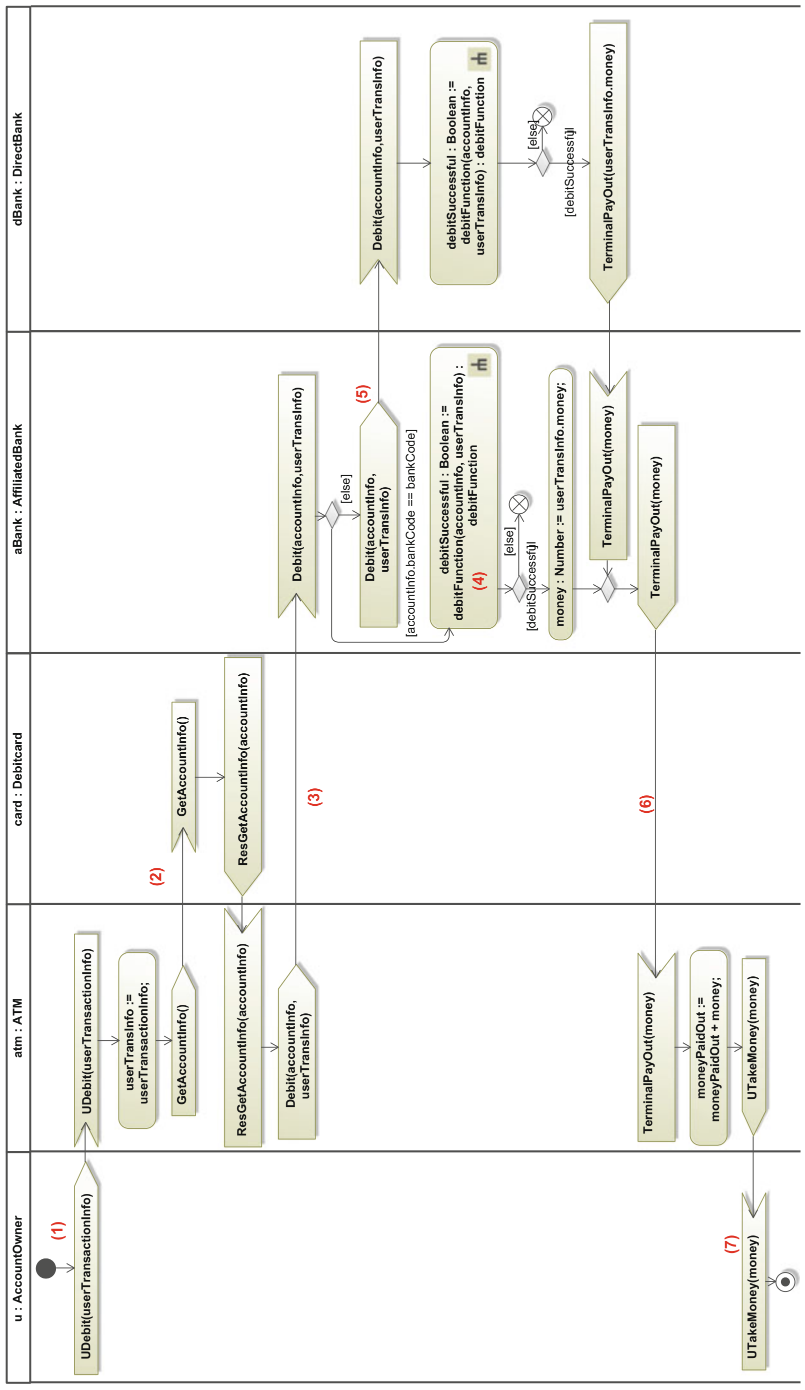

u : AccountOwner
atm : ATM
card : Debitcard
aBank : AffiliatedBank
dBank : DirectBank
(1)
UDebit(userTransactionInfo)
UDebit(userTransactionInfo)
userTransInfo := userTransactionInfo;
GetAccountInfo()
(2)
GetAccountInfo()
ResGetAccountInfo(accountInfo)
ResGetAccountInfo(accountInfo)
Debit(accountInfo, userTransInfo)
(3)
Debit(accountInfo,userTransInfo)
[else]
Debit(accountInfo, userTransInfo)
(5)
[accountInfo.bankCode == bankCode]
debitSuccessful : Boolean := debitFunction(accountInfo, userTransInfo) : debitFunction
(4)
[else]
[debitSuccessful]
money : Number := userTransInfo.money;
TerminalPayOut(money)
TerminalPayOut(money)
Debit(accountInfo,userTransInfo)
debitSuccessful : Boolean := debitFunction(accountInfo, userTransInfo) : debitFunction
[else]
[debitSuccessful]
TerminalPayOut(userTransInfo.money)
(6)
TerminalPayOut(money)
moneyPaidOut := moneyPaidOut + money;
UTakeMoney(money)
(7)
UTakeMoney(money)

Fig. 4 Protocol to withdraw money from an ATM

3.3.3 Overview over the Handshake Protocol

The other use case in the example is the online transaction. It is much more complicated than using an ATM because first a secure channel must be set up between the smart card and the online banking service by cryptographic means and second because the money transfer to another bank may fail in which case the transaction must be reverted.

In Fig. 5, the handshake protocol is shown as a sequence diagram. A sequence diagram in SecureMDD serves only documentation purposes and shows only the involved components and message types that are exchanged. This makes it impossible to generate code, whereas an activity diagram contains everything needed.

The handshake protocol establishes a secure session between `Debitcard` and `OnlinebankingService` using security data types and cryptographic operations defined in SecureMDD. There are four participants involved in this scenario: the `AccountOwner`, a `PC`, the `Debitcard`, and the `Onlinebanking-Service`. The user starts the protocol by sending the message `UHandshake` to the PC (i.e., by interacting with GUI of a program on the PC). The PC begins by sending `Handshake` to the `Debitcard`. Afterward, the PC only forwards messages between the card and the online banking service. The card sends a `ClientHello` containing a nonce encrypted with the public key of the online banking service and the card's certificate. The server answers with `ServerHello` containing the card's nonce and a server nonce encrypted with the card's public key. As the last major step, the card generates a session key and sends it together with the server nonce encrypted with the server's public key to the online banking

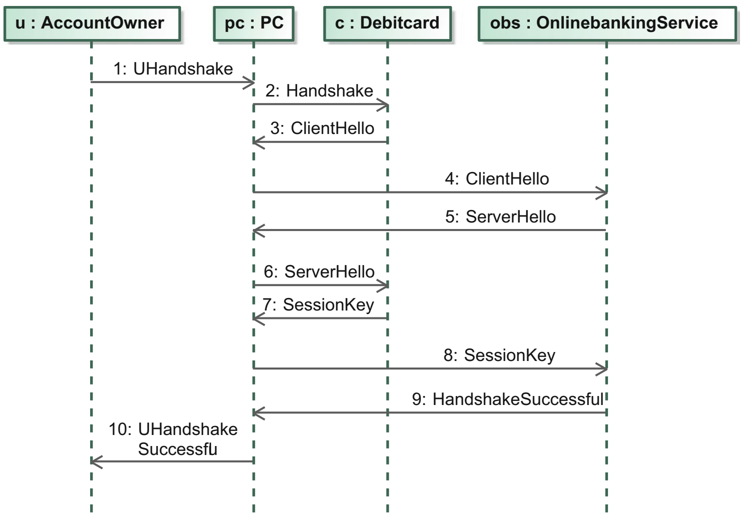


Fig. 5 Messages of the handshake protocol

service. This is a standard challenge-response protocol that is used in a similar form in TLS and many other protocols. Since the protocol has nothing that is specific to services and since cryptographic protocols in SecureMDD have been described in detail elsewhere (e.g., [42, 45, 46]), the activity diagram is omitted here.

3.3.4 Protocol for Money Transfer

Figure 6 shows the protocol to transfer money from one account to another one. It will only run after a successful handshake. This is indicated by a state `AUTHENTICATED` that is checked by the card (2). If the check succeeds, the online transaction can be processed. The account owner has to type in the transaction data, namely, the PIN, the amount of money that should be transferred, and the receiver's account information. This data (`uoti`) is sent from the PC to the card (1). The `card` checks the state, wraps `uoti` and the account information (`accountInfo`) that is initially stored on the card in the message `TransactionData`, and encrypts it with the previously exchanged session key (3). `encrypt` is a predefined operation in MEL. Then the state is set to `IDLE`, and the encrypted data is sent to the online banking service via the PC. After receiving the message, the service checks that its state is also `AUTHENTICATED` and decrypts the received data with the previously exchanged session key. The state is set back to `IDLE` to avoid replay attacks, and on the basis of the card holder's bank code, it is checked which bank the card holder account belongs to. Depending on this, the message is either sent over the port `affiliated` to the affiliated bank (4) or over the port `direct` to the direct bank (5). The ports are modeled in the deployment diagram (see Fig. 2). If bank `b1` receives a transaction message with PIN, the amount of money that should be transferred, the sender's account information, and the receiver's account information, it checks if the card holder account belongs to it. If so, it checks if the receiver's account also belongs to it and processes the transaction internally (6). Otherwise, the bank deducts the sum from the card holder's account and sends a request to the bank of the receiver's account to increase the sum in its account (7). If this fails because of a nonexistent account number, `b1` is notified, and the deduction from the card holder account is revoked (8). The failure (denoted by the flow final) will be propagated back to the user. Otherwise, the transaction was successful, and a message is sent back over the online banking service and the PC to the user. But before the notification is sent to the user, the PC closes the session with the online banking service using the stereotype «`closeSession`» (9). The online banking service must store the state and the session key across several protocol steps. Therefore, the service is stateful and provides a new instance for every invoker. Because the same instance could also be used in another protocol, it is necessary to model the first call of a stateful service. Therefore, the stereotypes «`openSession`» and «`closeSession`» are supported. «`openSession`» was used in the handshake protocol which was not shown.

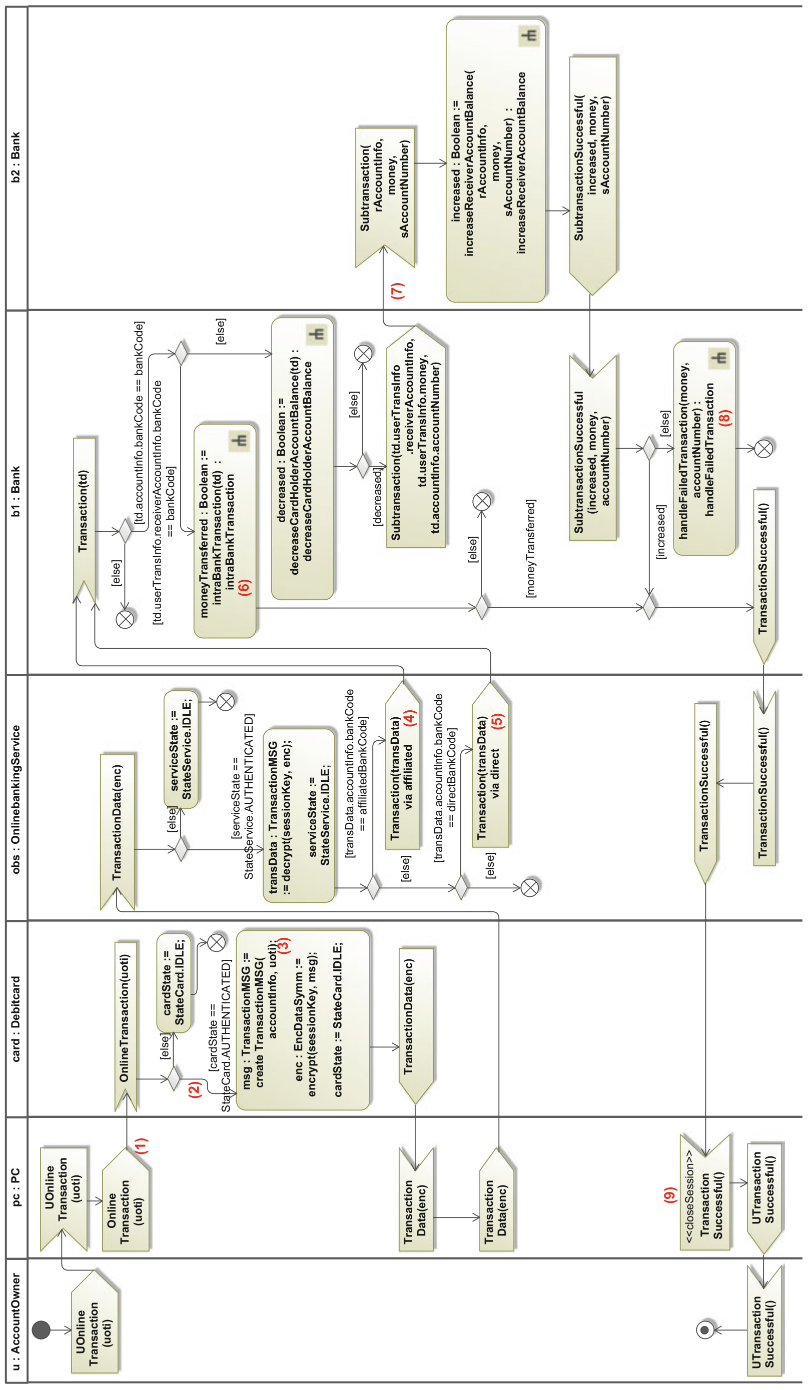

u : AccountOwner
pc : PC
card : Debitcard
obs : OnlinebankingService
b1 : Bank
b2 : Bank
UOnline Transaction (uoti)
Online Transaction (uoti)
(1)
OnlineTransaction(uoti)
[else]
cardState := StateCard.IDLE;
(2)
[cardState == StateCard.AUTHENTICATED]
msg : TransactionMSG := create TransactionMSG(accountInfo, uoti);
(3)
enc : EncDataSymm := encrypt(sessionKey, msg);
cardState := StateCard.IDLE;
TransactionData(enc)
Transaction Data(enc)
TransactionData(enc)
serviceState := StateService.IDLE;
[serviceState == StateService.AUTHENTICATED]
transData : TransactionMSG := decrypt(sessionKey, enc);
serviceState := StateService.IDLE;
[transData.accountInfo.bankCode == affiliatedBankCode]
Transaction(transData) via affiliated
(4)
[transData.accountInfo.bankCode == directBankCode]
Transaction(transData) via direct
(5)
Transaction(td)
[td.accountInfo.bankCode == bankCode]
[td.userTransInfo.receiverAccountInfo.bankCode == bankCode]
moneyTransferred : Boolean := intraBankTransaction(td) : intraBankTransaction
(6)
decreased : Boolean := decreaseCardHolderAccountBalance(td) : decreaseCardHolderAccountBalance
[decreased]
Subtransaction(td.userTransInfo .receiverAccountInfo, td.userTransInfo.money, td.accountInfo.accountNumber)
(7)
Subtransaction(rAccountInfo, money, sAccountNumber)
increased : Boolean := increaseReceiverAccountBalance(rAccountInfo, money, sAccountNumber) : increaseReceiverAccountBalance
SubtransactionSuccessful(increased, money, sAccountNumber)
SubtransactionSuccessful (increased, money, accountNumber)
[increased]
handleFailedTransaction(money, accountNumber) : handleFailedTransaction
(8)
[moneyTransferred]
TransactionSuccessful()
TransactionSuccessful()
TransactionSuccessful()
<<closeSession>> Transaction Successful()
(9)
UTransaction Successful()
UTransaction Successful()

Fig. 6 Protocol to transfer money from one account to another

3.3.5 Structuring Protocols

In Fig. 6, we have seen the use of the superclass `Bank` in an activity diagram. This is very useful to avoid the modeling of redundant behavior and makes the diagrams clearer. The deployment diagram (see Fig. 2) ensures that if `b1` is a `DirectBank`, then `b2` is an `AffiliatedBank` and vice versa.

Some functionality is encapsulated in methods that are predefined or defined in the model. Methods designed in the model like `intraBankTransaction`, `decreaseCardHolderAccountBalance`, `increaseReceiverAccountBalance`, and `handleFailedTransaction` allow big and complex protocols to be divided into smaller diagrams, so that all of them remain clear.

The activity diagram `increaseReceiverAccountBalance` (Fig. 7) models an internal behavior. Therefore, it has only one swim lane for a `Bank`. Activity

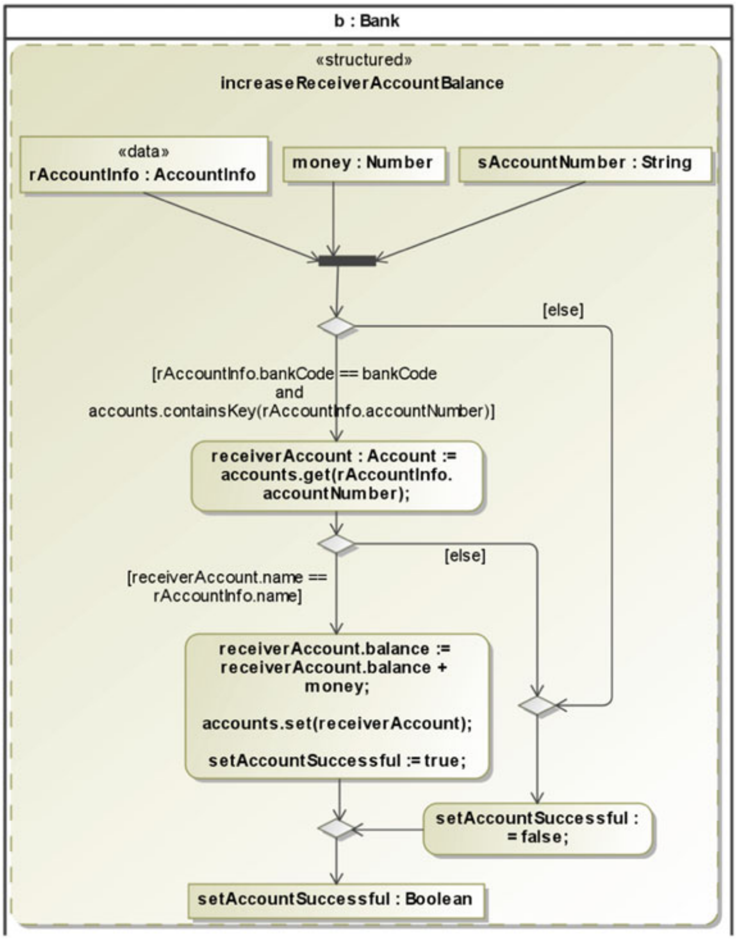

b : Bank
«structured»
increaseReceiverAccountBalance
«data»
rAccountInfo : AccountInfo
money : Number
sAccountNumber : String
[else]
[rAccountInfo.bankCode == bankCode
and
accounts.containsKey(rAccountInfo.accountNumber)]
receiverAccount : Account := accounts.get(rAccountInfo. accountNumber);
[else]
[receiverAccount.name == rAccountInfo.name]
receiverAccount.balance := receiverAccount.balance + money;
accounts.set(receiverAccount);
setAccountSuccessful := true;
setAccountSuccessful : = false;
setAccountSuccessful : Boolean

Fig. 7 Sub-activity increaseReceiverAccountBalance

parameters are used to pass arguments. The operation checks if the bank code is correct and if the account number exists. If this is the case, the account is credited and the operation returns true. Otherwise, nothing happens and false is returned. The graphical visualization of the operation is the choice of the modeler. It is also possible to write the whole code as one big blob.

This concludes the description of the example. All diagrams can be found on our Web page.[3] It shows how complex applications involving different types of components (services, terminals, PCs, smart cards, and users) can be modeled in SecureMDD. An important aspect is that the full behavior of the application can be modeled, the communication protocol and the internal behavior of the components. SecureMDD provides a UML profile, predefined cryptographic operations and data types, and modeling guidelines to easily model distributed security-critical applications.

The next section describes the formal specification and verification of such applications.

4 Formal Specification and Verification

In this section, the formal model which is automatically generated from the platform-independent UML model is introduced. In Sect. 4.1, an overview of the transformation process is given. Section 4.2 introduces the static part of the formal model, i.e., the data types, components of the application, and message types. In Sect. 4.3, the specification of the dynamic aspects of the system are described with ASMs. Sections 4.4 and 4.5 explain specific details for services, and Sect. 4.6 reports on the formal verification of the example.

4.1 Overview of the Transformation Process

The formal model that is generated from the UML diagrams is an abstract view of the whole UML model (which is a representation of the complete system under development). It contains an arbitrary but finite number of components (smart cards, terminals, and services) and has an arbitrary number of interleaved protocol runs.

The static aspects of an application, i.e., the components, data types, communication infrastructure, and the attacker, are defined using algebraic specifications. The dynamic part of an application, i.e., the cryptographic protocols, is given as an ASM. The ASM consists of two sets of rules: rules defining the behavior of the attacker and rules defining the dynamic behavior of the components (*agents* in the formal model). Executing rules induces a trace of states. Since applicable rules are

[3]http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/.

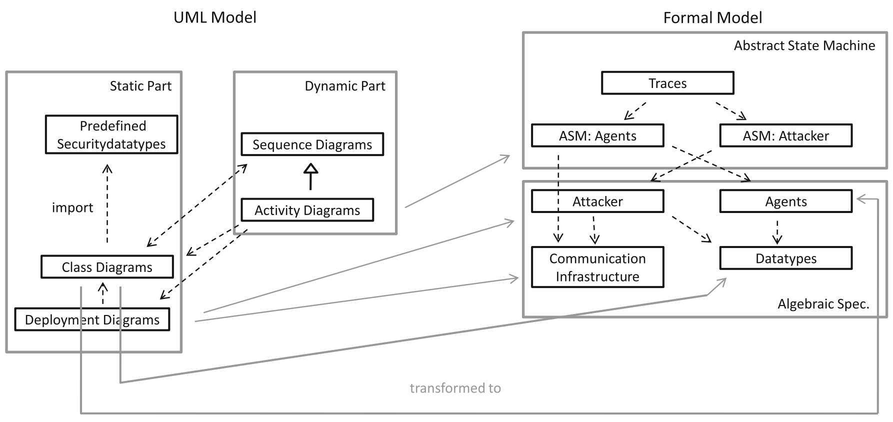


Fig. 8 Dependencies between the UML model and the formal model

chosen nondeterministically, a set of traces is obtained, "everything that can happen in this world". Figure 8 shows the components of the formal model and from which part of the UML model they are generated. Technically, a UML model is loaded into Eclipse with the Eclipse modeling framework, and specification text suitable for our interactive theorem prover KIV [5] is generated with XPand [61], also part of the Eclipse modeling framework. Other provers could be supported by writing new XPand transformations.

The generated formal model is then used to prove the security of the modeled application with KIV. One relevant security property for the banking example is that the sum of (electronic) money in the system is constant, i.e., no money is lost, and it is not possible to "generate" money. Money dispensed at an ATM is also counted. This can be formulated as an OCL constraint for the class diagram and is translated into a proof obligation in the formal model.

4.2 The Static Part: Data Types and Algebraic Specifications

4.2.1 Data Types

The data of the application, i.e., the messages, predefined security data types, and data types defined in the class diagrams, are translated into algebraic specifications. To reduce the gap between the UML models and the formal model used for interactive verification, we use the same data types as in the class diagrams and do not add a generic data format like some other approaches, e.g., [26, 54]. This simplifies the verification considerably. For example, an account is specified as a freely generated data type

```
Account = mkAccount(. .accountNumber : string;
                    . .name : string;
                    . .pin : Secret;
                    . .balance : int;
                    . .dispo : int);
```

and the data type message consists of all 21 messages:

```
message = mkTransactionData(. .msg : EncData) with isTransactionData
        | mkTransactionSuccessful with isTransactionSuccessful
        ⋮
```

The specification is not intended to faithfully represent instances of arbitrary class diagrams. Without a heap or references (or something similar), it is not possible to model arbitrary pointer structures like cycles or shared objects. However, this is not necessary since messages and data used in a cryptographic protocol do not (and should not) use such features.

4.2.2 Dynamic Functions

Every instance of a component class (service, terminal, smart card, or user) becomes an agent in the formal specification because it has a behavior as defined by the protocol steps. The attributes of the component classes are modeled as dynamic functions that are modified (updated) by the ASM rules. For each attribute, one dynamic function that maps an agent to the attribute's value is defined. For example,

ATM-moneyPaidOut : agent $\rightarrow$ int;

is the dynamic function for the `moneyPaidOut` attribute of class `ATM`, and ATM-moneyPaidOut(ag) returns the value of the attribute for a given ATM agent ag. The banking system contains 20 dynamic functions. Experience has shown that slicing a class into its attributes simplifies verification as compared to one dynamic function that returns the full state of an agent. The reason is that an update of one attribute and a lookup of another attribute are syntactically disjoint:

update(ATM-userTransInfo(ag), uti) and ATM-moneyPaidOut(ag)
as compared to something like
update(ATM(ag), update(ATM(ag).userTransInfo, uti))(ag).moneyPaidOut

The specification guarantees that the number of agents is finite so that it is safe to iterate over all agents and summarize ATM-moneyPaidOut(ag), i.e., the sum of all money dispensed at all ATMs.

4.2.3 Cryptography and the Attacker

The predefined security data types do not depend on the concrete application, and the transformation is generic for all applications. For example, encrypted data is specified as a freely generated data type `EncData`:

EncData = mkEncData(. .key : SymmKey; . .plain : PlainData);

This means the encrypted data contains the key used for encryption. This approach is similar to [54]. This allows to easily specify whether a decryption will succeed or not: The key must be the same as the one used for encryption (private and public key pair for asymmetric encryption). In the specification of the attacker, he/she cannot access the key directly so that the behavior of the cryptographic data types and operations is the same as in reality.

The attacker is only implicitly present in the UML model by `«Threat»` stereotypes. In the formal specification, the attacker is modeled explicitly as a separate agent that can interact with other components by sending and receiving messages. He/she is associated with a set of data that represents his/her knowledge. If the attacker eavesdrops on a communication path and data is sent over that path, it is analyzed and added to the attacker's knowledge. For example, if the attacker obtains an encrypted message and knows the key, he/she decrypts the message and analyzes its contents to find, e.g., secret PINs. Or, if he/she obtains a key, he/she can decrypt some previously obtained encrypted messages that may contain other keys. Again, this is similar to [54]. Conversely, the attacker can only send messages he/she can construct from his/her knowledge. He/she can encrypt and send a message with a key he/she knows (and he/she must know the data to encrypt) or he/she can simply send a previously obtained message (a replay attack) even if he/she cannot decrypt it. The attacker is not able to decrypt messages without knowing the key, or to generate arbitrary keys, or to guess arbitrary nonces or secrets. This models the fact that it is virtually impossible to find a key or nonce by chance or brute force in the lifetime of the application. This is sometimes called the "perfect cryptography assumption" or "symbolic cryptography" as compared to "computational cryptography". Non-cryptographic data such as numbers or strings are never a secret and can always be used by the attacker in messages. Therefore, a PIN must be modeled as a secret (1234 in itself is known as a number to everybody, but as a PIN, it should be kept secret).

Only the specifications for predefined security data types and cryptographic operations that are actually used will be generated. The banking system uses symmetric and asymmetric encryption, digital signatures, nonces, and secrets (PINs).

4.3 The Dynamic Part: Abstract State Machine and Traces

The dynamic part of the application, i.e., the security protocols, is defined with activity diagrams in UML. In the formal model, they are translated into an ASM. The ASM consists of a number of rules, basically the individual protocol steps. An applicable rule is chosen nondeterministically in a given state and evaluated yielding another state (one STEP). Rules are applied arbitrarily often (STEP*). In effect, all possible (finite and infinite) traces of the specified "world" are generated:

```
ASM = STEP*
```

STEP nondeterministically chooses an action to perform. Possible steps are an attacker step or a step for a component type. For example, if the chosen step is the OnlinebankingService-agent step, an arbitrary component `ag` of the `OnlinebankingService` class is chosen and the ASM rule ONLINEBANKINGSERVICESTEP is executed for this component.

```
STEP =
    choose asm-step do
        if asm-step = attacker-step then ATTACKER
        else if asm-step = OnlinebankingService-agent-step then
            choose ag with exOnlinebankingService(ag) do
                ONLINEBANKINGSERVICESTEP
        else ...
```

The ONLINEBANKINGSERVICESTEP contains all protocol steps the online banking service can perform. Message passing is modeled with input queues (inboxes for short). A protocol step can be executed if the inbox of a component contains a message of the appropriate type; otherwise, nothing happens.

```
ONLINEBANKINGSERVICESTEP =
choose port with is-valid-port(ag, port) and inputs(ag)(port) ≠ [] do
let inmsg = inputs(ag)(port).first in
    inputs := rem(ag, port, inputs) seq
    if (isSessionKey(inmsg)) then SESSIONKEY else
    if (isClientHello(inmsg)) then CLIENTHELLO else
    if (isTransactionData(inmsg)) then TRANSACTIONDATA else
    if (isTransactionSuccessful(inmsg)) then TRANSACTIONSUCCESSFUL else
    ...
```

The message is removed from the inbox, and the correct ASM rule runs. The rule processes the message, performs checks, and updates the state of the component as defined in the corresponding protocol step in the activity diagram. The sending of a new message is modeled by placing it in the inbox of the receiving component. If the attacker can eavesdrop on the communication path (*read* capability in the deployment diagram), the message is also added to the attackers knowledge.

```
TRANSACTIONDATA =
let enc = inmsg .msg in
    if (serviceState(ag) = AUTHENTICATED) then
if (not (can_decrypt(sessionKey(ag), enc) and
    isTransactionMSG(decrypt(sessionKey(ag), enc)))) then
    STOPSTEP
else
    let transData = decrypt(sessionKey(ag), enc).transactionMSG in ...
```

TRANSACTIONDATA encapsulates the actual ASM rule. All the intermediate steps are just a convenient grouping of related rules. Aside from some syntactical differences, the abstract program is very similar to the activity diagram. It has the same control structure, checks, and assignments. But there are differences. For example, in MEL, the if statement with the test `if (not (can_decrypt ...` does not occur. It is added during the transformation. MEL only contains the statement

```
transData : TransactionMSG := decrypt(sessionKey, enc);
```

`decrypt` is a predefined MEL operation that behaves in the generated code (and hence in the real world) as follows: It performs the actual decryption (by applying an algorithm like DES or AES) and then checks that the result is actually an object of the expected type (`TransactionMSG` above). If this is not the case, an exception is thrown. The abstract code has the same behavior. If the key is not correct (in this case can_decrypt is false), the result of decryption is a meaningless sequence of bytes (for good algorithms like AES). If the key is correct, the result is something meaningful, but could be of a different type. This is tested in the second part (`isTransactionMSG(decrypt(...))`). Since the abstract programming language has no exceptions, the same behavior is obtained with the if-then-else.

One ASM rule is evaluated atomically. We assume that an attacker cannot influence or modify what happens inside a component and cannot read a component's local state. Components are considered as secure in the model. An attacker can only interact with messages and inboxes according to the deployment diagram (Fig. 2, described in Sect. 3.1). Another consequence of this atomicity is that the access to a Web service must be serialized as explained in Sect. 5.2. This can be an efficiency problem but is currently necessary to achieve the same behavior for the model and the generated code.

In case the attacker was chosen instead of a component, the ASM rule for the attacker is called. Then, it is nondeterministically chosen if the attacker suppresses or sends a message. If the attacker suppresses a message, a nonempty inbox is chosen that belongs to a channel where messages can be suppressed by the attacker. Then, a message is deleted from that inbox. In the send case, the attackers generate an arbitrary message from his/her current knowledge. The message is then sent to a randomly chosen inbox accessible by the attacker (i.e., the channel has the attacker-send property).

4.4 Transport Layer Security and the Attacker

In Sect. 3, it was mentioned that security stereotypes for connections like «TLS» influence the stereotype «Threat». If an attacker has the abilities to *read*, *send*, and *suppress* messages (i.e., a Dolev–Yao attacker [21] for this connection) and the connection is secured with TLS, the attacker loses some of those abilities. Because TLS is a secure protocol, we use some of its security properties [20]. TLS begins with a handshake that authenticates one or both communication partners. Then messages are encrypted with a session key, their integrity is ensured by a message authentication code (MAC), and a sequence number is used to detect missing or replayed messages. If an error is detected, the connection is closed.

We assume that an attacker is not able to obtain a valid TLS certificate that is accepted by other components. This means he/she cannot initiate a TLS-secured communication if mutual authentication is used. His/her abilities regarding a communication between two components are also limited: The attacker can only read encrypted messages. The used key is a session key exchanged during authentication that will never be used in another session. Therefore, reading the messages is useless for the attacker because he/she cannot decrypt them and they cannot be used for replays because of the sequence number. As mentioned previously, we are only concerned with logical security properties, not traffic analysis where the message length or timing may be important.

Furthermore, in the formal model, the attacker loses his/her ability to send messages, because if the message is not encrypted with the correct session key (which the attacker does not possess), the MAC verification will fail and the message will not be accepted. A replayed message is encrypted with the correct session key, but will not be accepted because of the sequence number. The ability to suppress messages is lost as well because the next message will have an incorrect sequence number. However, the attacker has the ability to terminate the session by replaying a message, because any error in a TLS session leads to termination.

To summarize, it is appropriate to formalize a TLS-secured connection as one where an attacker can either do nothing or can only abort the connection (depending on the annotations in the deployment diagram Fig. 2).

4.5 Stateful and Stateless Services as Agents

A service component can be stateless or stateful. Both must be treated slightly different in the formal model. A stateless service is similar to other agents like terminals and smart cards. The formal model may have an arbitrary number of services or it may be restricted to exactly one.

A stateful service however creates an instance of itself for each invoker. The actual code of the invoker calls a manager which is also implemented as a stateless service. It creates an instance of the actual service, deploys it, and returns the address of this service instance. All this is done by the framework used in the generated code as described in Sect. 5.1. This behavior is not modeled in the UML model of the application, and it is not reflected in the formal specification because it is an implementation issue only. We assume that an attacker has the same abilities for the communication between the two services as between the client and the service. Therefore, no additional security weaknesses are created in the code. Only the address of the service is transmitted which is not a security-critical information leak because either the attacker cannot read this information (if no threat is present in the deployment diagram) or he/she can act as man in the middle anyway (if a threat is present and TLS is not used) or a man-in-the-middle attack is not possible because TLS is used with mutual authentication as described in Sect. 4.4.

For one stateful service, the formal model has an arbitrary number of agents that represent the different new instances of the same service. They all have the same initial attributes that are reset with every connection establishment. Thus, a stateful service is modeled as a set of agents that can be handled as the other agent types.

4.6 Verification of Security Properties

Besides the automatic code generation, the verification of security properties for the generated applications is a major benefit for the development of secure systems. With this approach, we do not have to guess whether the application is secure, since we were able to formally verify it.

Usually only generic properties like secrecy or authentication are proven for security protocols (see [36] for an overview). In contrast, the SecureMDD approach focuses on application-specific security properties [45]. They give better confidence in the properties of the application as a whole. In the banking system (Sect. 3), it is interesting to know that PINs and session keys remain secret, but the real properties are about money. The system has the property that the amount of money is constant in the following sense:

The sum of all account balances plus the amount of all the money that has been withdrawn from cash machines is constant.

This property can be formalized as an OCL constraint that is added to a class in the UML model:

```
Bank.allInstances().accounts.balance->sum() +
ATM.allInstances().moneyPaidOut->sum() = C
```

where C is an unspecified constant. This OCL constraint is translated into a property of the abstract state machine, i.e., the sum is constant in all states of all runs of the ASM. We define

```
BanksMoney(accounts) = Bank.allInstances().accounts.balance->sum()
ATMMoney(moneyPaidOut) = ATM.allInstances().moneyPaidOut->sum()
sum = BanksMoney(accounts) + ATMMoney(moneyPaidOut)
```

BanksMoney(accounts) and ATMMoney(moneyPaidOut) are the algebraic terms that are the result of the translation of the OCL constraints. BanksMoney iterates over the accounts of the banks and summarizes their `balance` attributes, and ATMMoney iterates over all components of type ATM and sums up their `moneyPaidOut` attributes (see the class diagram Fig. 3).

Then the property can be written formally as

$$\text{init}(\ldots) \wedge \textit{sum} = \text{C} \rightarrow [\text{STEP*}]\ \textit{sum} = \text{C}$$

$[\cdot]$ is the box operator of dynamic logic. The meaning of $[\alpha]\varphi$ is that if program α terminates the condition, φ holds afterward. We start with an initial state (i.e., no protocol steps have been executed, all components are in their initial state, the initial knowledge of the attacker is fixed, and so on). Then, the protocol steps are chosen nondeterministically and executed. Thus, we consider all finite sequences of steps. As described in Sect. 4.3, the ASM consists of a while loop that executes STEP or stops. Therefore, the proof works by proving an invariant for every step of the ASM. Of course, the invariant must already hold in the initial state:

$$(\text{INV}(\ldots) \wedge \textit{sum} = \text{C}) \rightarrow [\text{STEP}]\ (\text{INV}(\ldots) \wedge \textit{sum} = \text{C})$$

It turns out the property stated above is not quite correct because not yet finished protocol runs must be taken into account. For example, during an online transaction (Fig. 6), there is a situation where one account has been debited, but the receiver not yet credited. In a sense, the money is contained in the message between the two banks—it is in transit—and must be included in the money count. So the actual sum must be computed as follows:

$$\textit{sum} = \text{BanksMoney(accounts)} + \text{ATMMoney(atms)} + \text{MoneyInTransit(inboxes)}$$

The definition of `MoneyInTransit` uses the inboxes of the ASM that are used to model message passing. The correct OCL constraint therefore must include inboxes. For example, if all inboxes are empty, then `MoneyInTransit` is zero, and we can write

```
AllInboxes().isEmpty implies[4]
Bank.allInstances().accounts.balance->sum() +
ATM.allInstances().moneyPaidOut->sum() = C
```

It is possible to define `MoneyInTransit` in OCL or to specify it directly in the KIV system. As it turns out, the correct definition is quite complicated and was found only after several corrections. The reason for the complexity is twofold: the behavior of the protocol itself and the attacker abilities.

The protocol for withdrawing money from an ATM is quite straightforward: The user's account is debited, and the amount to dispense is sent in a `TerminalPayOut` message (see Fig. 4) either directly to the ATM or via the AffiliatedBank. Therefore, we have to count the money in the `TerminalPayOut` messages. The online transfer (Fig. 6) debits the sender's account and sends a

[4] `implies` is an OCL keyword since an arrow `->` is used for operations on collections.

`Subtransaction` message to the other bank whose money must be counted. The receiving bank answers with a `SubtransactionSuccessful` message. However, crediting the receiver may fail because of an incorrect account number. In this case, the message contains a `false` flag, `true` otherwise. This means the money of a `SubtransactionSuccessful` message must be counted if and only if the flag is `false`.

However, the attacker must be taken into account. He/she has very limited capabilities in the example as was explained in Sects. 3.1 and 4.4. He/she has full control over only one communication path, the connection between a `PC` and the `OnlinebankingService` (see Fig. 2). This means he/she can inject arbitrary messages, for example, `TerminalPayOut`, `Subtransaction`, or `SubtransactionSuccessful` messages even if this is completely useless (both `PC` and `OnlinebankingService` ignore them). Therefore, we must count the money contained in these messages only if they occur between the banks and/or the ATM. There the attacker cannot inject messages, so they must be genuine.

To prove the main security property, a more generalized invariant must be established that consists of several subproperties. This is necessary for almost all applications. In the banking system, the following properties must be established:

- Generic properties about connections between components. Only those communication paths as specified in the deployment diagram are possible (Fig. 2).
- Properties that some messages occur only between dedicated components. This is related to the exact specification of MoneyInTransit and the exact behavior of service answers (because a service always answers its caller).
- Properties about the sending account. The banks are modeled as stateless services. This means that all needed information must be contained in the messages. If a transaction fails, the receiving bank must know the account number of the sending account to initiate a refund (with message `SubtransactionSuccessful` and flag `false`). This account number is sent in the `Subtransaction` message.

  Therefore, it must be proved (and the property is actually needed for the main proof) that the sending account number in those two messages really exists.

These properties are far from obvious and were found during failed proof attempts. If it turns out that the invariant is not strong enough (or even incorrect, i.e., not really invariant), it must be modified, and the proofs must be done again. This often happens even if the protocol is secure. The effort can be reduced significantly if it is ensured that the invariant proofs are done automatically by the proof tool. In KIV, this can be ensured with suitable rewrite rules.

The final proof uses 244 lemmas that require 953 user interactions and 17,125 proof steps for their proofs. A little less than half of the effort is needed for the invariant, the rest for properties of the various counting functions. A first version of the case study required 484 theorems, 4,229 user interactions, and 27,038 proof steps. This shows how our verification technique for SecureMDD applications and cryptographic protocols has improved over time. Other case studies show similar improvements. All specifications, lemmas, and proofs can be found on our Web page[5] together with several other case studies.

5 Automatic Code Generation

SecureMDD not only supports the modeling and formal verification of security-critical applications but also the automatic generation of runnable code for the application from the class and activity diagrams. As described in Sect. 2, three platforms are supported: smart cards, terminals, and services.

- Smart cards: The full smart card code of the application is generated as Java Card code. It can be deployed without any modifications or extensions. Java Card [28] is a version of Java tailored to resource-constrained devices. Java Card has the usual Java statements and expressions, but no strings, no integers, no floats, no threads, no reflection, and no garbage collection. Communication with a smart card is done with sequences of bytes, i.e., byte arrays.

  These limitations make programming in Java Card very difficult and error prone. The code generation handles serialization of objects for communication, reuse of objects on the card (this is necessary because of the missing garbage collection), details of short arithmetics, and cryptographic operations. More details can be found in [43].
- Terminals: For terminals, the full protocol logic is generated in Java, as well as code for the communication with other components, including object serialization. Only GUI code for interaction with a user must be added. This can be done without modifying or interfering with the protocol logic [23].

  The code is similar to hand-programmed Java with one subtle exception. The MEL semantics knows no pointers, but only values because object identities play no role in security protocols. Therefore, the generated code uses value comparison (with `equals` methods) instead of pointer comparison and ensures that assignments behave as if the object was deep cloned. Copying is used only if aliasing and field updates cause incorrect side effects. This can be analyzed given the activity diagrams.
- Services: The protocol logic and code for communication and serialization are generated. The code can be deployed without additions or modifications in a Java service framework.

  The rest of this section describes the service code generation and deployment in more detail.

Technically, the UML model is loaded into Eclipse with the Eclipse modeling framework. Then model-to-model transformations written in QVT [53] create intermediate platform-specific UML models for each platform. From these models,

[5] http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/.

source code is generated by model-to-text transformations with XPand [61]. Both QVT and XPand are part of the Eclipse modeling framework.

As mentioned in Sect. 2, the transformations are designed so that the generated code is correct and secure with respect to the formal specification. A formal proof that the transformations indeed guarantee this property is an ongoing research.

5.1 Services and Service Communication

The service components are implemented as Java Web services that use SOAP [41] as underlying technology. To implement Web services, Metro[6] that integrates JAX-WS [47] (Java API for XML—Web Services) is used. JAX-WS is a standard and supports server and clients and can be used by annotated plain old Java objects (POJOs) that are generated by the transformations.

A service is implemented as a Java class that is annotated with @*WebService*, and the public interface of a service is defined by service operations that are annotated with @*WebMethod*. The return value of a service operation is sent to the invoker. For asynchronous message passing, the return value is void and the operation is annotated with @*OneWay*.

A SecureMDD service always contains only one service operation `process` (see Listing 1.) that handles all incoming messages. It invokes the internal method `processMessage` with the message that is wrapped in a `MessageWrapper` object. The wrapping is necessary because JAXB (Java Architecture for XML Binding, part of JAX-WS) needs a container to transmit an object of a class hierarchy with information about the object's run time type. `msg.getMsg()` selects the actual message from the wrapper. This is an object of class `Message`, the common superclass for all messages in the class diagram (Fig. 3). The method `processMethod` makes a case distinction over the actual type of the message and dispatches to one method for each message class. This approach (only one public method to handle all incoming messages) is also used in the terminal and smart card code and behaves exactly as the formal ASM (Sect. 4.3).

```
@WebMethod
public synchronized MessageWrapper process(MessageWrapper msg)
       throws ServiceException {
  MessageWrapper m = null;
  try {
          m = new MessageWrapper(processMessage(msg.getMsg()));
  } catch (java.lang.Exception e) { stop(); }
  return m;
}

private Message processMessage(Message inmsg) throws java.lang.Exception {
        switch (inmsg.getCode()) {
                case Code.DEBIT :
                        return processDebit((Debit) inmsg);
```

[6] http://metro.java.net/.

```
                case Code.TERMINALPAYOUT :
                        return processTerminalPayOut((TerminalPayOut) inmsg);
                ...
                default :
                        stop();
                        return null;
        }
}
```

Listing 1. The single service operation of a SecureMDD service and the dispatcher

The OnlineBankingService handles an online transfer by delegating it to the customer's bank. This happens when the service receives a `TransactionData` message (see Fig. 6). For each message, one Java method is generated. The method `processTransactionData` is shown in Listing 2.

```
private Message processTransactionData(TransactionData inmsg)
                throws java.lang.Exception {
synchronized (manager) {
  EncDataSymm enc = inmsg.getMsg();
  if (serviceState == StateService.getAUTHENTICATED()) {
     TransactionMSG transData = (TransactionMSG)
                                  (EncDataSymm.decrypt(sessionKey, enc));
     setServiceState(StateService.getIDLE());
     if (transData.getAccountInfo().getBankCode()
                       .equals(affiliatedBankCode)) {
        return sendMsg(new Transaction(transData), Ports.affiliated);
     } else {
             if (transData.getAccountInfo().getBankCode()
                               .equals(directBankCode)) {
                return sendMsg(new Transaction(transData), Ports.direct);
             } else {
                     stop();
                     return null;
             }
     }
  } else {
          setServiceState(StateService.getIDLE());
          stop();
          return null;
  } } }
```

Listing 2. Main method for an online transfer

The body of the method is generated from the activity diagram. The method is called with a TransactionData object, checks the current state of the (stateful) service, and decrypts the message with the session key exchanged before. The `decrypt` method is part of the SecureMDD implementation of cryptographic operations and raises an exception if the internal decryption does not yield a valid serialized object or if the object is not of the expected type. Both can happen because of an attack. An attacker may send garbage, or a message encrypted with another key, or replay a different message encrypted with the correct key. The ASM rule (in Sect. 4.3) shows the same behavior by corresponding checks in an `if` statement (`if (not (can_decrypt ...)))`). An exception always aborts a protocol step, and the ASM rule simply finishes in this case. A generic method `sendMsg` finishes the protocol step by sending the next message (a `Transaction`) to the next service.

A stateful service is implemented by two service classes. One class is annotated with *@Stateless* and the other with *@Stateful*. The first is the *service manager* and the second the actual *stateful service*. An invoker calls the service manager and obtains an address for a fresh instance of the stateful service that was created by the service manager. After that, the invoker communicates with a service instance created exclusively for it (see Sect. 4.5 for a discussion why this does not create a security hole).

Services are invoked by stubs that are automatically generated by the library *wsimport* that is a part of JAX-WS. Stubs manage the communication between client and service by mapping Java objects to XML documents and vice versa as well as the transport of the XML documents. Which stubs have to be generated for a service invoker component is defined by the deployment diagram. The generated stubs also contain the classes that are transferred, but without method implementations. Hence, the classes generated by *wsimport* are replaced by the classes that are generated by the model transformation.

The deployment diagram specifies how many service instances of the same component can be invoked by one component instance. If a component instance can invoke only one instance of a service component (denoted by multiplicity 1), then the stubs are generated at deployment time and can be used without changes. Otherwise (with multiplicity *), for each service to invoke, its address must be known. The stubs that are generated for one instance can be used with an address for any instance of the same type.

5.2 Parallel Service Invocation

Because services can access shared memory at the same time, it is important to consider parallelism. Especially, protocol steps with read and write access have to be executed atomically. This is the intended behavior, and the ASM behaves like this. A possible solution is that the developer of an application has to manage the synchronization explicitly in the UML model. A more comfortable way is that the synchronization is managed automatically by the code generation. Our current solution is to execute only one service operation at any given time, i.e., to force a completely sequential behavior. For a stateless service, the bodies of the operations are synchronized on the service instance, and for stateful services, the bodies have to be synchronized on the manager instance. Possible future work is to look for a strategy to synchronize only the critical parts of the code.

5.3 Transport Layer Security

TLS with server side authentication is provided by a Java library. If another TLS implementation should be used, the generated TLS code can be disabled. TLS uses keys and certificates; thus, we need a key and trust store to provide them. The key store contains asymmetric key pairs, while the trust store provides signed certificates; if a certificate represents a certificate authority, all certificates issued by this authority will also be accepted. To use TLS, keys and certificates have to be transferred inside a secure environment before the system can be deployed.

5.4 Deployment

The deployment diagram contains all important information to deploy an application (some additional information is defined in the class diagram). To deploy the banking system, the following information is important. The application model contains a user that represents a real human, a smart card component, and components that can be deployed on PCs or servers. Except for the `OnlinebankingService`, each of these components can be instantiated multiple times, i.e., an arbitrary number of `PC` applications and `Bank` services can be deployed. The generated Java Card code can be loaded onto an arbitrary number of smart cards, and the ATM code can be loaded onto an arbitrary number of real ATMs.

The online banking service can be accessed by many account owner PCs at the same time and can invoke operations of the bank services. The communication between a user and a PC has to be secured against an attacker (i.e., against shoulder surfing and malware). A PC can communicate with a service over any kind of network, and services can also communicate over any kind of network.

For the PC, a Java package containing all necessary code is generated. The class `PC` can be instantiated on any device that supports Java and is secured against an attacker (i.e., free of malware). For the bank service, a Java package is generated as well. This package can be deployed inside a container on any Java Web server.

The class diagram defines attributes that have to be initialized at the start of a service (they are annotated with «`Initialize`» in the class diagram). For the banks, these are the initial accounts and bank codes; for the `OnlinebankingService`, public and private keys and bank codes; etc. These attributes must be passed as parameters to the constructor before the service is started.

SecureMDD provides predefined key-value lists that are used for the accounts. The generated code provides a prototypical implementation that resides in memory. It can be replaced by any other implementations or databases that implement the same interface.

Because requirements usually change during software development, it is useful that code for the modeled applications can be automatically generated and tested. For testing, a test case that initializes all instances, deploys all services, and calls the user messages to execute the protocols has to be written. If such a test case exists, the whole process from code generation over service deployment up to running the test code can be invoked with one click inside Eclipse. In our test framework, all services are deployed on one lightweight server that is integrated in Java, but of course it is possible to deploy the services on other servers.

The full generated and runnable code can be found on our website.[7]

6 Related Work

Related work relevant for this chapter can be divided into three categories: verification of cryptographic protocols, model-driven development of security-critical systems, and model-driven development of service applications. They are treated in turn.

6.1 Verification of Cryptographic Protocols

A lot of verification techniques and tools to prove the security of cryptographic protocols exist; an overview is given in [36]. These techniques can be divided into three categories: belief logics (e.g., [17]), state exploration (e.g., [6, 37]), and theorem proving (e.g., [10, 39, 54]). Most of them are based on automatic tools and focus on generic security protocols (which are not specific to an application, e.g., authentication protocols) and prove standard security properties. In contrast, the protocols of the banking system as well as the security property highly depend on the considered application. It is not clear if automatic tools can cope with that kind of security property.

Some approaches dealing with application-specific security properties exist. One method that is related to ours is the inductive approach of Paulson [54] that uses the theorem prover Isabelle for verification and was successfully applied to several case studies. Bella extends the inductive approach to deal with smart cards [8] but concentrates on generic security protocols as well. In [9], Bella et al. give an overview of their work on SET (Secure Electronic Transaction), a set of e-commerce protocols devised by Visa and MasterCard. The case study is formally modeled and verified using the inductive approach. Considered and proven (application-specific) security properties are that the payment information of the customer are only known to the bank, not to the merchant, and that the order information is not known to the bank.

Another interesting case study with application-specific security properties was the Mondex electronic purse. Mondex has received a lot of attention because its formal verification has been set up as a challenge for verification tools [60] that several groups worked on. The results of the participating groups are summarized in [30]. Mondex is about transferring money (from one card to another), and the main property is that no money is lost.

[7]http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/projects/secureMDD/.

6.2 Model-Driven Development of Security-Critical Systems

Surveys can be found in [33] and [29]. The most closely related approach is UMLSec by Jan Jürjens ([32] and newer work). He developed a method to model systems based on cryptographic protocols with UML. Inputs for model checking and automatic theorem proving are automatically generated from the models and standard security properties are proven. Besides several other case studies, Jürjens worked on the security of the Common Electronic Purse Specification (CEPS) for cash-free point-of-sale transactions. One considered security property was that the sum of balances of all smart cards (which are used for payments) and all cards of the merchants (where the earned money is stored) is the same at any time. The appropriate proof was not done by tools but is paper based [31]. UMLSec does not aim at generating runnable code since the focus is on modeling only the security-related parts.

Smith et al. [58] model security protocols with UML class diagrams and state machines. Partial code can be generated from the state machine and tested against attacks. The attacker model and the number of components is fixed. Formal verification is not supported. Bushager et al. [18] use UML use case and sequence diagrams to model smart card protocols. Partial code can be generated, but since the internal behavior of the components is not modeled, that code must be added by hand. Verification is also not supported. There is quite a lot of work on modeling access control. We mention only SecureUML by Basin et al. [7]. UML class diagrams are used to model security-critical applications with role-based access control (RBAC). Specific authorization constraints can be defined with OCL. Code generation is not supported.

6.3 Model-Driven Development of Service Applications

The approach developed by Deubler et al. [19] considers the development of security-critical service-oriented systems. For modeling and verification, it uses the tool AUTOFOCUS [27] that provides its own modeling language similar to UML. The considered security mechanisms are authentication and authorization that are proved with a model checker. Application-specific properties as well as code generation are not considered. AUTOFOCUS was used by Grünbauer et al. [25] to model a banking application where a customer can submit a transfer order online. Essentially this is the handshake protocol and the first part of the transfer protocol of our banking system (the actual transfer is not considered). The confidentiality and authenticity of the order is proved with a model checker. However, the attacker capabilities had to be simplified because the original model was too complex for automatic verification.

SECTISSIMO by Memon et al. [40] is a framework to model security-critical services with a business process language, enrich the model with security policies, and generate code to enforce the policy. The security policy can be composed of a set of given cryptographic primitives and protocols. Formal verification of security properties is not supported.

MDD4SOA developed by Mayer et al. [38] is a model-driven approach for service orchestration that transforms a platform-independent model into several PSMs and those to partial code for the languages BPEL, WSDL, and Java and the formal language Jolie. It uses its own UML profile [22] to allow the modeling of SOA and verify properties with the formal language Jolie. Compliance of service orchestration with their interaction protocols can be checked automatically [56]. Security aspects are not considered, and the full behavior of the components is not modeled.

There is some work on model-driven development of Web services in general. Baina et al. [4] use UML state machines to describe service communication and generate BPEL-based service skeletons that implement conversation management logic. Gronmo et al. [24] import Web service descriptions into UML, composite them, and generate a new Web service description. Both approaches focus on service composition and neither model the complete service behavior nor consider security. Sheng et al. [57] focus on context-aware Web services. UWE4JSF by Kroiss et al. [35] defines a UML profile and can generate executable code.

The following papers consider security in a model-driven approach for Web service architectures. Nakamura et al. [51] describe security with UML by primitives that will be transformed into security configurations such as WS-SecurityPolicy. They do not verify the security of an application and only parts of an application are modeled. Alam et al. [1] use OCL and role-based access control (RBAC) and generate eXtensible Access Control Markup Language (XACML) policy files to define a security infrastructure. This approach focuses on access control only.

Formal approaches to services can also be used to formally verify security properties. SecureMDD uses ASMs to define the behavior of components like services, and if services communicate with other services, this is a simple form of service orchestration. More elaborate approaches to specify services with ASMs by Börger and Thalheim [16] and Börger and Sörensen [14] contain interesting ideas for future extensions of SecureMDD.

We are not aware of an approach like ours that allows model-driven development of security-critical Web service applications, generates executable code, and guarantees application-specific security properties for the modeled system by using interactive verification.

7 Conclusion

We presented in detail how services are supported in our SecureMDD approach. Security in service applications is an important issue. Since it is difficult to express business security requirements with standard security properties, our approach supports the consideration of application-specific security properties in all stages of the development process. Secure applications using smart cards, terminals, and services are modeled with extended UML and the domain-specific language MEL. The banking system used as a case study demonstrates the need to verify application-specific security properties even if standard security protocols like TLS are used. From a platform-independent UML model, runnable code as well as a formal specification is generated automatically. The code describes an application in full detail and can be deployed and used without any changes. The formal specification is used to verify the application-specific security properties. The code is correct and secure with respect to the formal specification.

Future work for services includes handling parallelism in a more efficient way, integrating a real database and WS-Security standards, and extending the communication structure to allow more complex service orchestrations. SecureMDD will also support Android and Apps as an additional platform in the future.

References

1. Alam, M.M., Breu, R., Breu, M.: Model driven security for web services (MDS4WS). In: 8th International Multitopic Conference, 2004. Proceedings of INMIC 2004, pp. 498–505. IEEE, Piscataway (2004)
2. Anderson, R.J., Needham, R.M.: Programming satan's computer. In: Computer Science Today, vol. 1000, pp. 426–440. Springer, Heidelberg (1995)
3. Armando, A., Arsac, W., Avanesov, T., Barletta, M., Calvi, A., Cappai, A., Carbone, R., Chevalier, Y., Compagna, L., Cúellar, J., et al.: The AVANTSSAR platform for the automated validation of trust and security of service-oriented architectures. In: Proceedings of TACAS 2012 – Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 7214. Springer, Heidelberg (2012)
4. Baina, K., Benatallah, B., Casati, F., Toumani, F.: Model-driven web service development. In: Advanced Information Systems Engineering, pp. 527–543. Springer, Heidelberg (2004)
5. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol. 1783. Springer, Heidelberg (2000)
6. Basin, D.A., Mödersheim, S., Viganò, L.: OFMC: a symbolic model checker for security protocols. Int. J. Inf. Secur. **4**(3), 181–208 (2005)
7. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: from UML models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. **15**, 39–91 (2006)
8. Bella, G.: Mechanising a protocol for smart cards. In: Proceedings of e-Smart 2001, International Conference on Research in Smart Cards. Lecture Notes in Computer Science, vol. 2140. Springer, Heidelberg (2001)
9. Bella, G., Massacci, F., Paulson, L.C.: Verifying the SET purchase protocols. J. Automat. Reas. **36**(1–2), 5–37 (2006)
10. Blanchet, B.: Automatic verification of correspondences for security protocols. J. Comput. Secur. **17**(4), 363–434 (2009)
11. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Model-driven development of secure service applications. In: 2012 35th Annual IEEE Software Engineering Workshop (SEW), pp. 62–71. IEEE, Piscataway (2012)
12. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Model checking of security-critical applications in a model driven approach. In: Software Engineering and Formal Methods. Springer, Heidelberg (2013)

13. Borek, M., Moebius, N., Stenzel, K., Reif, W.: Security requirements formalized with ocl in a model-driven approach. In: 2013 IEEE Model-Driven Requirements Engineering Workshop (MoDRE). IEEE, Piscataway (2013)
14. Börger, E., Sörensen, O.: BPMN core modeling concepts: inheritance-based execution semantics. In: Handbook of Conceptual Modeling. Theory, Practice, and Research Challenges, pp. 287–332. Springer, Heidelberg (2011)
15. Börger, E., Stärk, R.F.: Abstract State Machines—A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
16. Börger, E., Thalheim, B.: Modeling workflows, interaction patterns, web services and business processes: the ASM-based approach. In: Proceedings of ABZ 2008. Lecture Notes in Computer Science, vol. 5238. Springer, Heidelberg (2008)
17. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Trans. Comput. Syst. **8**(1), 18–36 (1990)
18. Bushager, A., Zwolinski, M.: Modelling smart card security protocols in systemC TLM. In: IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing, pp. 637–643. IEEE Computer Society, Piscataway (2010)
19. Deubler, M., Grünbauer, J., Jürjens, J., Wimmel, G.: Sound development of secure service-based systems. In: Proceedings of the 2nd International Conference on Service Oriented Computing, pp. 115–124. ACM, New York (2004)
20. Dierks, T., Rescorla, E.: The transport layer security (TLS) protocol version 1.2. IETF Network Working Group. http://www.ietf.org/rfc/rfc5246.txt (2008)
21. Dolev, D., Yao, A.C.: On the security of public key protocols. In: Proceedings of 22th IEEE Symposium on Foundations of Computer Science. IEEE, Piscataway (1981)
22. Foster, H., Gönczy, L., Koch, N., Mayer, P., Montangero, C., Varró, D.: UML extensions for service-oriented systems. In: Rigorous Software Engineering for Service-Oriented Systems, pp. 35–60. Springer, Heidelberg (2011)
23. Grandy, H., Stenzel, K., Reif, W.: Object-oriented verification kernels for secure Java applications. In: Aichering, B., Beckert, B. (eds.) SEFM 2005 – 3rd IEEE International Conference on Software Engineering and Formal Methods. IEEE, Piscataway (2005)
24. Gronmo, R., Skogan, D., Solheim, I., Oldevik, J.: Model-driven web services development. In: 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service, 2004. EEE'04, pp. 42–45. IEEE, Piscataway (2004)
25. Grünbauer, J., Hollmann, H., Jürjens, J., Wimmel, G.: Modelling and verification of layered security protocols: a bank application. In: Proceedings of SAFECOMP 2003. Lecture Notes in Computer Science, vol. 2788. Springer, Heidelberg (2003)
26. Haneberg, D., Grandy, H., Reif, W., Schellhorn, G.: Verifying smart card applications: an ASM approach. In: International Conference on integrated Formal Methods (iFM) 2007. Lecture Notes in Computer Science, vol. 4591. Springer, Heidelberg (2007)
27. Huber, F., Molterer, S., Rausch, A., Schatz, B., Sihling, M., Slotosch, O.: Tool supported specification and simulation of distributed systems. In: Proceedings, International Symposium on Software Engineering for Parallel and Distributed Systems, 1998, pp. 155–164. IEEE, Piscataway (1998)
28. Java Card 2.2.2 Application Programming Interfaces: http://www.oracle.com/technetwork/java/\javacard/specs-138637.html (2006)
29. Jensen, J., Jaatun, M.G.: Security in model driven development: a survey. In: Sixth International Conference on Availability, Reliability and Security, ARES 2011. Lecture Notes in Computer Science, pp. 704–709. Springer, Heidelberg (2011)
30. Jones, C., Woodcock, J. (eds.): Form. Asp. Comput. **20**(1) (2008)
31. Jürjens, J.: Developing high-assurance secure systems with UML: a smartcard-based purchase protocol. In: IEEE International Symposium on High Assurance Systems Engineering. IEEE, Piscataway (2004)
32. Jürjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
33. Kasal, K., Heurix, J., Neubauer, T.: Model-driven development meets security: an evaluation of current approaches. In: 44th Hawaii International Conference on System Sciences (HICSS), pp. 1–9. IEEE Computer Society, Piscataway (2011)

34. Katkalov, K., Moebius, N., Stenzel, K., Borek, M., Reif, W.: Model-driven testing of security protocols with secureMDD. In: Fifth IFIP International Conference on New Technologies, Mobility and Security (NTMS 2012). IEEE, Piscataway (2012)
35. Kroiss, C., Koch, N., Knapp, A.: UWE4JSF: a model-driven generation approach for web applications. In: 3rd Workshop on The Web and Requirements Engineering at ICWE 2012. Lecture Notes in Computer Science, vol. 5648, pp. 493–496. Springer, Heidelberg (2009)
36. Lopez Pimental, J.C., Monroy, R.: Formal support to security protocol development: a survey. Computacion y Sistemas **12**(1), 89–108 (2008)
37. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
38. Mayer, P., Schroeder, A., Koch, N.: MDD4SOA: model-driven service orchestration. In: Proceedings of 12th IEEE International EDOC Conference (EDOC 2008). IEEE, Piscataway (2008)
39. Meadows, C.: The NRL protocol analyzer: an overview. J. Logic Program. **26**(2), 113–131 (1996)
40. Memon, M., Hafner, M., Breu, R.: SECTISSIMO: a platform-independent framework for security services. In: Proceedings of the First International Modeling Security Workshop. CEUR Workshop Proceedings, vol. 413. http://ceur-ws.org/Vol-413/ (2008)
41. Mitra, N., Lafon, Y.: SOAP Version 1.2. W3C (2007)
42. Moebius, N., Stenzel, K., Reif, W.: Modeling security-critical applications with UML in the SecureMDD approach. Int. J. Adv. Softw. **1**(1), 59–79 (2008)
43. Moebius, N., Stenzel, K., Grandy, H., Reif, W.: Model-driven code generation for secure smart card applications. In: 20th Australian Software Engineering Conference. IEEE, Piscataway (2009)
44. Moebius, N., Stenzel, K., Grandy, H., Reif, W.: SecureMDD: a model-driven development method for secure smart card applications. In: Workshop on Secure Software Engineering, SecSE, at ARES 2009. IEEE, Piscataway (2009)
45. Moebius, N., Stenzel, K., Reif, W.: Formal verification of application-specific security properties in a model-driven approach. In: Proceedings of ESSoS 2010 - International Symposium on Engineering Secure Software and Systems. Lecture Notes in Computer Science, vol. 5965. Springer, Heidelberg (2010)
46. Moebius, N., Stenzel, K., Borek, M., Reif, W.: Incremental development of large, secure smart card applications. In: Proceedings of the Workshop on Model-Driven Security. ACM, New York (2012)
47. Mordani, R., Chinnici, R., Hadley, M.: The Java API for XML-Based Web Services (JAX-WS) 2.0. JCP (2006)
48. Murdoch, S.J., Drimer, S., Anderson, R., Bond, M.: Chip and PIN is broken. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 433–446. IEEE, Piscataway (2010)
49. Nadalin, A., Kaler, C., Hallam-Baker, P., Monzillo, R.: Web Services Security: SOAP Message Security 1.0. OASIS (2004)
50. Nadalin, A., Goodner, M., Gudgin, M., Barbir, A., Granqvist, H.: WS-SecurityPolicy 1.2. OASIS (2006)
51. Nakamura, Y., Tatsubori, M., Imamura, T., Ono, K.: Model-driven security based on a web services security architecture. In: IEEE International Conference on Services Computing, pp. 7–15. IEEE, Piscataway (2005)
52. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978)
53. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1. http://www.omg.org/spec/QVT/1.1/ (2011)
54. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. J. Comput. Secur. **6**, 85–128 (1998)
55. Ray, M., Dispensa, S.: Renegotiating TLS. Technical Report, PhoneFactor Inc. (2009)

56. Schroeder, A., Mayer, P.: Verifying interaction protocol compliance of service orchestrations. In: Proceedings of the 6th International Conference on Service-Oriented Computing. Lecture Notes in Computer Science, vol. 5364. Springer, Heidelberg (2008)
57. Sheng, Q.Z., Benatallah, B.: Contextuml: a uml-based modeling language for model-driven development of context-aware web services. In: International Conference on Mobile Business, 2005. ICMB 2005, pp. 206–212. IEEE, Piscataway (2005)
58. Smith, S., Beaulieu, A., Greg Phillips, W.: Modeling and verifying security protocols using UML 2. In: International Systems Conference (SysCon), pp. 72–79. IEEE Computer Society, Piscataway (2011)
59. Stenzel, K., Moebius, N., Reif, W.: Formal verification of QVT transformations for code generation. In: 14th International Conference on Model Driven Engineering Languages and Systems, MODELS 2011. Lecture Notes in Computer Science, vol. 6981. Springer, Heidelberg (2011)
60. Woodcock, J.: First steps in the verified software grand challenge. IEEE Comput. **39**(10), 57–64 (2006)
61. Xpand: http://projects.eclipse.org/projects/modeling.m2t.xpand (2009)