Model-driven development of secure service applications

Marian Borek, Nina Moebius, Kurt Stenzel, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Borek, Marian, Nina Moebius, Kurt Stenzel, and Wolfgang Reif. 2012. "Model-driven development of secure service applications." In *2012 35th Annual IEEE Software Engineering Workshop, 12-13 October 2012, Heraclion, Greece*, 62–71. Piscataway, NJ: IEEE. https://doi.org/10.1109/sew.2012.13.

Model-Driven Development of Secure Service Applications

Marian Borek, Nina Moebius, Kurt Stenzel, Wolfgang Reif
Institute for Software and Systems Engineering
Augsburg University
86135 Augsburg, Germany
{borek,moebius,stenzel,reif}@informatik.uni-augsburg.de

***Abstract*—The development of a secure service application is a difficult task and designed protocols are very error-prone. To develop a secure SOA application, application-independent protocols (e.g. TLS or Web service security protocols) are used. These protocols guarantee standard security properties like integrity or confidentiality but the critical properties are application-specific (e.g. "a ticket can not be used twice"). For that, security has to be integrated in the whole development process and application-specific security properties have to be guaranteed. This paper illustrates the modeling of a security-critical service application with UML. The modeling is part of an integrated software engineering approach that encompasses model-driven development. Using the approach, an application based on service-oriented architectures (SOA) is modeled with UML. From this model executable code as well as a formal specification to prove the security of the application is generated automatically. Our approach, called SecureMDD, supports the development of security-critical applications and integrates formal methods to guarantee the security of the system. The modeling guidelines are demonstrated with an online banking example.**

***Index Terms*—UML, model-driven development, Security, Web Service, code generation, security-critical systems**

I. Introduction

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 and standards (e.g. WS-BPEL, SoaML, WSDL, BPMN) as well as a large number of approaches ([14], [2], [9]) exist to develop those kind of applications. Most of the business processes and E-Government applications are security-critical. Of course, standard security properties such as confidentiality, integrity and non-repudiation of data which is sent to and received from services as well as authentication of participants play an important role. These standard aspects are covered by existing standards such as WS-Security [19] and WS-SecurityPolicy [18] and the use of standard security protocols, e.g. TLS [7]. Unfortunately, using those application-independent standards and protocols is not sufficient to guarantee the security of an application. Moreover, for almost all applications application-dependent cryptographic protocols have to be designed additionally that combine the existing security mechanisms and cover the security aspects specific to an application.

One example for a security-critical service application is an online banking system. Here, it is important to guarantee that no money is lost, i.e. if an amount is debited from one account it is credited to another account. This example resp. its security aspects will be explained in this paper. Other examples and their application-specific security properties are E-Voting (where an attacker must not be able to manipulate the votings or cast a vote twice), a ticket system for public transport (where it has to be ensured that the electronic ticket cannot be used twice or by different persons and that an unused ticket may only be returned once). Other examples are E-Government services, e.g. the change of registration after moving to another city. Here, it has to be ensured that every inhabitant is registered exactly once (in Germany each city has its own registration office that manages the data of the people living in this city).

Our approach to develop security-critical service applications allows to completely model the whole system with UML. Thus, in contrast to other approaches (e.g. 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 are able to generate a formal specification for interactive verification of application-specific security properties. Those give stronger guarantees to the security of an application than standard properties like secrecy and integrity.

This paper focuses on the modeling of a security-critical service application with UML. To illustrate our modeling approach we use an online banking application as case study. Furthermore, we exemplify that the consideration and verification of application-specific security properties is mandatory to guarantee the security of an application based on service-oriented architectures. Additionally, some interesting aspects of code generation are explained.

This paper is structured as follows. Section 2 gives an overview of the model-driven approach and Section 3 discusses related work. Section 4 and 5 demonstrate the modeling of a service application with an example of an online banking system. Section 6 explains selected aspects regarding the code generation as well as the deployment. In Section 7 the security aspects of secure service applications are described. Section 8 concludes this paper and describes future work.

II. The SecureMDD Approach

The goal of the SecureMDD approach is the adaptation of model-driven software development to the characteristics of security-relevant systems. The focus of the project lies on the application domains of E-Commerce and E-Government. For applications of this domains it is very important to give guarantees about application-specific properties (as to prevent economic damage) in an unsecure environment. This goal is reached by a continuous integration of formal methods into a model-driven software development method.

Besides service applications SecureMDD supports the development of secure smart card applications [16]. Smart cards are resource-constrained devices which are not very powerful. Challenges to deal with are the missing garbage collection on the cards, unsupported large primitive types such as integers and the limited storage space. Smart cards are programmed using Java Card [23], a dialect of Java which is tailored for the use on resource-constrained devices. For smartcard applications all transformations are fully implemented and the approach is evaluated thouroughly. For service applications this is work in progress.

The SecureMDD approach is shown in Figure 1.

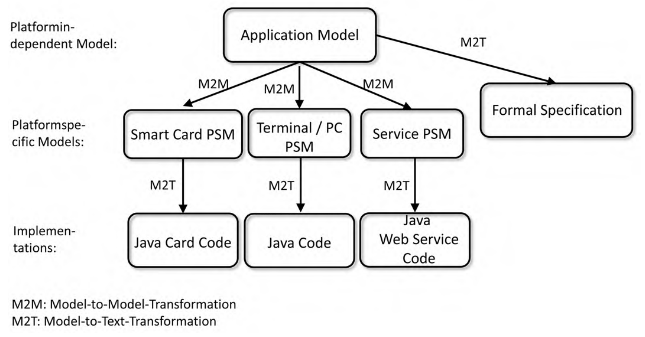


Fig. 1. SecureMDD Approach

The developer creates a platform-independent UML model of the application under development. This model is an abstract view of the system, omitting implementation details. To tailor UML to the domain of security-critical applications, SecureMDD defines a UML profile. Stereotypes are used, e.g. to apply TLS [7] to a communication path or to encrypt a message. To define the static view of the modeled system, class and deployment diagrams are used. The dynamic parts of the application, i.e. the cryptographic protocols, are modeled using sequence and activity diagrams. Since we completely model an application with UML, the activity diagrams were extended by an object-oriented domain-specific language called MEL (Model Extension Language). This language is used in the UML elements of activity diagrams, e.g. Actions, SendSignal Nodes and AcceptEvent Actions. The language is tailored to model cryptographic protocols and supports, e.g. object creation, declaration of local variables, assignments and several cryptographic operations such as encryption, decryption, digital signatures.

From the platform-independent UML model, three platform-specific models are generated using model-to-model transformations. As platforms we support smart cards, terminals and/or PCs as well as services. The platform for terminals and PCs comprises the components (e.g. PCs or vending machines) that are able to communicate with services. Besides, those term also denotes components that can be connected with a smart card reader and are able to communicate with a smart card. The terminals and PCs also have a user interface (e.g. a GUI). Thus, they get inputs from the user of the application and are able to return some output information to the user. The platform-specific models contain implementation details specific for the platform and are used to minimize the gap between the platform-independent UML model and the generated code.

Using the platform-specific models as input, executable code of the application is generated automatically using model-to-text transformations. For the smart card components, Java Card code is generated. For the terminals and PCs we generate Java code. Services are implemented as Java Web Services.

Furthermore, we automatically generate a formal specification from the platform-independent UML model. The specification is based on algebraic specifications and abstract state machines [5] and is used as input for the interactive theorem prover KIV [3]. The generated formal model is used to prove the security of the modeled application.

We are also working on the integration of AVANTSSAR[1], which combines several model checkers. It can be used to find flaws in the protocols automatically.

III. Related Work

There are some approaches related to ours but all differ in certain aspects.

MDD4SOA developed by Mayer et al. [14] is a model-driven approach for service orchestration that transforms a platform-independent model into several platform-specific models and those to code for the languages BPEL, WSDL, Java and the formal language Jolie. It uses its own UML profile to allow the modeling of service-oriented architectures and verify properties with the formal language Jolie. This approach does not consider security properties nor the security of a service application at all. It also does not generate executable code.

Further there are several model-driven approaches for web service development. Biana et al. [2] use state machines to describe service communication and generate BPEL-based service skeletons that implement conversation management logic. Gronmo et al. [9] 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. [22] also introduced a model-driven approach for web services but focus on context-aware web services and do not consider security.

[1]www.avantssar.eu

The following papers consider security in a model-driven approach for web service architectures. Nakamura et al. [20] 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 Extended Access Control Markup Language (XACML) policy files to define a security infrastructure. This approach focuses on access control only.

The approach developed by Deubler et al. [6] considers the development of security-critical service-oriented systems. For modeling and verification it uses the tool AUTOFOCUS [11] that provides an 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.

Other approaches are SecureUML [4] as well as UMLSec [12] that both aim to develop security-critical systems with extended UML. SecureUML is tailored to role-based access control applications and supports specific authorization constraints with OCL. UMLSec allows to express security properties like secrecy, integrity and role-based access control with stereotypes. Both do not focus on web services and consider only standard security properties whereas we are able to prove application-specific properties.

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

IV. Modeling of Secure Services Applications

This section demonstrates the modeling of a security-critical service application with an online banking example. The application allows its users to make bank transfers from anywhere over the internet. The example uses the indexed transaction authentication number (iTAN) protocol and considers the user view as well as the internal transfer process between bank services.

An account owner who wants to transfer money from one bank account to another has to do the following steps: First, he has to log in to the bank application (on his PC) using his username and password. Secondly, he enters the details of the transfer (e.g., account number and bank code of the receiver) and validates them. The validation (e.g., whether the bank number of the receiving account is valid) is done by the bank application. If the validation succeeds, the customer confirms the correctness of the transfer data and gets an iTAN index from his bank. Finally, he chooses the iTAN with the received index from his iTAN list and enters it. Afterwards, he is notified about the success or failure of the transfer. The failure of a transfer can have two reasons. The first is because of wrong user input (e.g. the receiving account does not exist) or because there are not sufficient funds in the account. In this case the account owner is informed that the transfer cannot be executed. The second reason for a transaction failure can be a technical issue (e.g., connection problems) or an attack on the system. In this case the customer is informed that the transfer could not be executed but, if there are sufficient funds in the account, the transfer will be done later.

The dynamic part of the outlined example is modeled with activity diagrams. Each protocol resp. functionality is modeled in a separate activity diagram. Fig. 2 shows the activity diagram for the online transaction of money (as described above). Due to space limitations only a part of the process is shown. Each activity diagram has one partition for each participant, i.e., user, PC, and service. To transfer money, three participants are required: an *AccountOwner* that represents a real person, the PC of the account owner (*AccountOwnerPC*) which has the bank software installed as well as a bank service (*Bank*) that offers the operations *login*, *validateTransactionInfo*, *requestTanIndex* and *processTransfer*. These operations can be invoked by a PC resp. by the bank application on the PC. The service operations are encapsulated in a separate activity diagram and may contain invocations of other service operations (see Fig. 5 for the invocation of *processTransfer*).

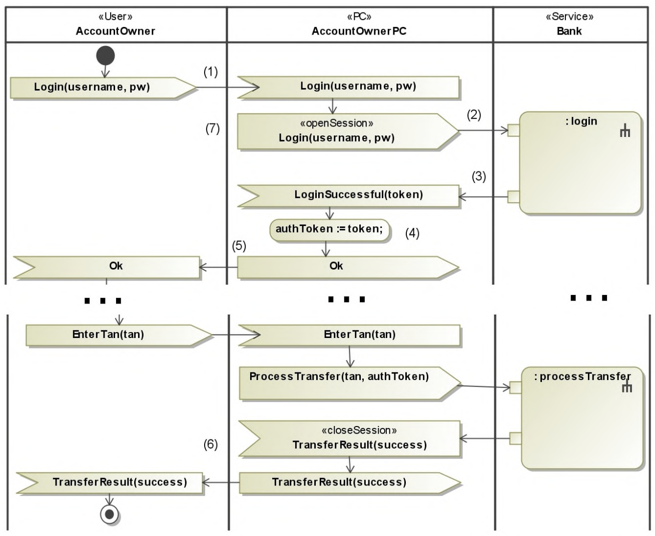


Fig. 2. Part of the activity diagram showing the online transfer of money

The transaction of money is initiated by the account owner. This is modeled with an initial node in the *AccountOwner* swimlane. The end of a protocol is denoted by a final node. In our example the protocol ends with the account owner receiving the information whether the transfer succeeded. The sending of a message to another participant as well as the invocation of a service operation is modeled with a UML SendSignal Action. The receiving of a message is modeled with a UML AcceptEvent Action and the service operation is modeled as a CallBehavior Action that is linked to the activity diagram defining the operation. All data that is exchanged between two participants is defined with message classes in the class diagram. Also, the arguments of a service invocation as well as the return value of a service are modeled as instances of messages classes. UML Actions are used to change the internal state of a participant. All UML elements contain expressions of the Model Extension Language (MEL) which is a domain-specific language that is tailored to model security-critical applications. For example, the language supports class instantiation, assignments and comparisons but also the use of predefined cryptographic operations. In a MEL expression the classes and attributes defined in the class diagrams can be accessed and manipulated.

The online transaction protocol modeled in Fig. 2 starts with the login of the account owner. This is modeled as the sending of a message of type *Login* (defined as a class in the class diagram) that has two attributes, *username* and *password*. The MEL expression *Login(username, pw)* (1) denotes that an instance of the message class is created and sent to the *AccountOwnerPC*. The *AccountOwnerPC* receives this message (modeled by an AcceptEvent Action). Implicitly, this means that local variables with name *username* and *pw* are declared and initiated with the values contained in the message. These local variables can be used, e.g., accessed and assigned later. Then, the *AccountOwnerPC* invokes the service operation *login* of the service with the name *Bank*, the operation has one parameter of type *Login*. The invocation is modeled with a SendSignal Action (2). The corresponding MEL expression creates a new message object of type *Login* which is used as argument for the operation call. The *Bank* service returns a message of type *LoginSuccessful*. The reception of this message by the *AccountOwnerPC* is modeled as AcceptEvent Action (3). The message contains a unique authentication token that is used for further communication with the *Bank* service. The AccountOwnerPC stores this token in his attribute *authToken* (which is defined as attribute of class *AccountOwnerPC* in the class diagram). This is modeled with a MEL assignment in a UML-Action (4). Then, the *AccountOwnerPC* confirms the login to the *AccountOwner* by sending a message of type *Ok* without arguments (5).

The steps *validateTransactionInfo* and *requestTanIndex* are omitted in Fig. 2. The last step is the process of a transfer which is initiated by the user who enters the iTan number for this transaction (EnterTan(tan)).

The stereotypes *openSession* and *closeSession* (6 and 7) define where a session between *AccountOwnerPC* and *Bank* begins and ends. This is necessary because *Bank* is a stateful service (see Fig. 3). A PC or terminal that invokes a stateful service for the first time, opens a session and gets a new instance of the service. During the session the invoker communicates with this service instance that can store session-dependent information like session keys, without synchronizing with other instances. The access of global attributes and classes is described in Sect. V-B. Because it is ambiguous at which point in a protocol a session starts and where it ends it is necessary to model this information explicitly.

Fig. 3 shows the deployment diagram of the case study. It describes the communication structure, attacker abilities and connection security. The components, i.e., all users, PCs and services, are modeled as UML-Nodes. The type of the component is annotated using one of the stereotypes ≪User≫, ≪PC≫ and ≪Service≫. The communication structure is modeled with UML-CommunicationPaths. Every path connects two components and can be unidirectional or bidirectional. In this example we use only unidirectional ones. This means that every communication is triggered by the *AccountOwner*; an *AccountOwnerPC* receives a message from the *AccountOwner* and is then able to send a message to the *Bank*. Afterwards the *Bank* is able to respond to this message but cannot send messages of its own accord. Because *Bank* has a path to itself an instance of the *Bank* service is able to invoke other instances of the *Bank* service and especially itself. The node *Bank* represents different banks (e.g. Chase or Bank of America). Each *bank* has its own service processing the transfers for the accounts of its bank. Thus, every bank service can be accessed by several account owner PCs for online banking and also by other bank services to process the transfers with the receiver having his account at this bank. So, this diagram describes a system that consists of many bank services that use the same protocols. There is no need to model server resp. the connection between services and server. The advantage is that the model is independent of the deployment of services by not defining which service is to be deployed on which server (see Fig. 3).

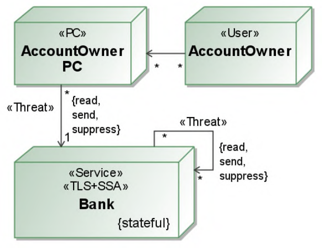


Fig. 3. Communication structure of the online banking example

Furthermore, the abilities of an attacker are modeled with the ≪Threat≫ stereotype that can be applied to any connection. The attacker can be a full Dolev-Yao [8] attacker who is able to read, send, and suppress messages on the fly, but he can also have only a subset of these abilities. To secure the communication between a service and its invokers Transport Layer Security (TLS) is supported. In this example, *Bank* applies the stereotype ≪TLS+SSA≫. This means TLS with server side authentication. A secure connection between *Bank* and *AccountOwnerPC* or *Bank* and *Bank* will be established and *Bank* has to authenticate itself towards the invoker. The use of standard protocols like TLS make the modeling of security-critical applications easier and should be used if the requirements are met. Besides TLS we also support special security data types for encrypted, signed or hashed message parts as well as for keys and other data types.

Fig. 4 depicts a snippet of the class diagram. It describes the static part of the example and shows the class *Bank* and some of its attributes. *Bank* is stateful, which means that for every

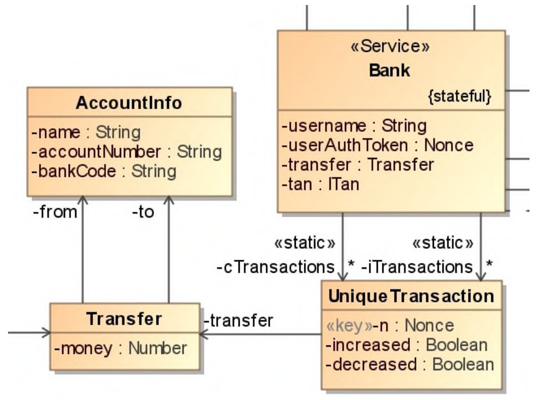
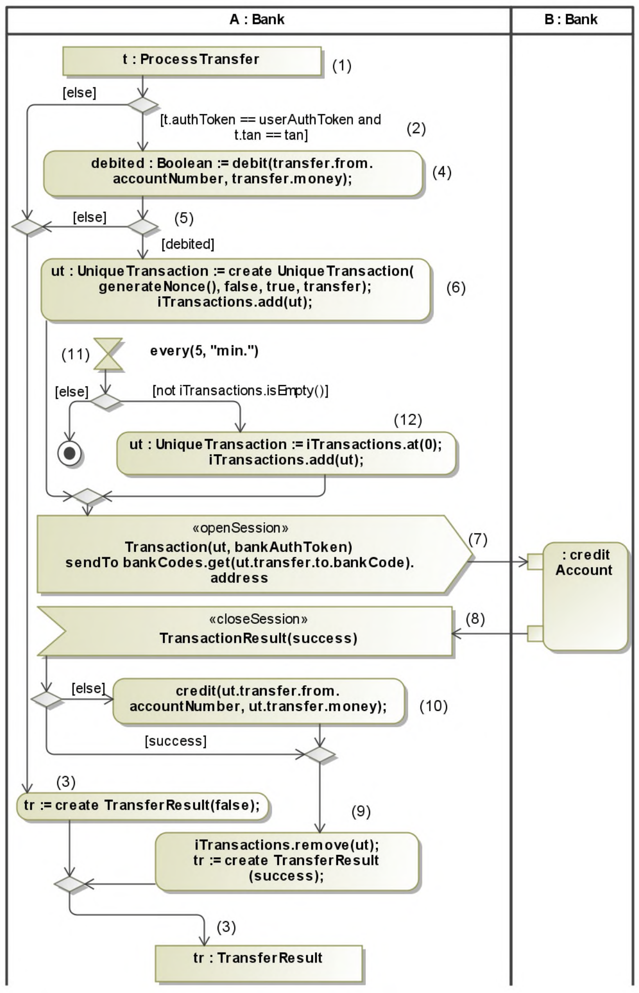


Fig. 4. Snippet of the class diagram

invoker a new service instance will be created. The class *Bank* has attributes to stores for a transfer the current username, user authentication token, transfer data and iTAN for every invoker created service instance. The user authentication token is used to allow only the service invoker to access the invoker created service instance. This token will be created and stored during the login process. The attribute transfer contains the amount of money that should be transferred and the account information of the source and destination accounts. Additionally, *Bank* contains two key-value container attributes named *cTransactions* and *iTransactions*. A key-value container is modeled by a star association to a class, in this case, to the class UniqueTransaction. This class has to contain an attribute with the applied stereotype ≪key≫. Each key in such a container is unique and is associated to exactly one value. *cTransactions* describes complete transactions, while *iTransactions* contains incomplete transactions. These containers are necessary for realization of a recovery protocol to ensure that a transfer is processed completely or not at all. They are used in Fig. 5. The association ends have the stereotype ≪static≫, which means that all service instances access the same attribute instances. ≪static≫ is only valid when applied on attributes of stateful services.

Fig. 5 depicts the activity diagram that defines the service operation *processTransfer*. This operation is called from the online transfer protocol (see Fig. 2). This service operation debits the money from an account of Bank A and credits it to an account of Bank B. A and B are different service instances. If the source and the target account are on the same bank then A and B are the same instance. To process the credit on Bank B a second service operation named *creditAccount* will be invoked. Furthermore, a recovery protocol is modeled as part of the process transfer diagram (see Fig. 5) to ensure that a transfer is processed completely or not at all if the connection is disconnected or an attacker suppresses messages. This is achieved by storing incomplete and complete transactions and a service that repeatedly tries to complete incomplete transactions.

In the following the protocol modeled in Fig. 5 is described. First, the service operation receives a message



A : Bank
B : Bank
t : ProcessTransfer
(1)
[else]
[t.authToken == userAuthToken and t.tan == tan]
(2)
debited : Boolean := debit(transfer.from. accountNumber, transfer.money);
(4)
[else]
(5)
[debited]
ut : UniqueTransaction := create UniqueTransaction(generateNonce(), false, true, transfer); iTransactions.add(ut);
(6)
(11)
every(5, "min.")
[else]
[not iTransactions.isEmpty()]
(12)
ut : UniqueTransaction := iTransactions.at(0); iTransactions.add(ut);
«openSession»
Transaction(ut, bankAuthToken)
sendTo bankCodes.get(ut.transfer.to.bankCode). address
(7)
: credit Account
«closeSession»
TransactionResult(success)
(8)
[else]
credit(ut.transfer.from. accountNumber, ut.transfer.money);
(10)
[success]
(3)
tr := create TransferResult(false);
(9)
iTransactions.remove(ut); tr := create TransferResult (success);
(3)
tr : TransferResult

Fig. 5. Activity diagram depicting the transfer protocol

of type *ProcessTransfer* (1). This is modeled with a UML ActivityParameter node. The message type is defined in the class diagram and contains the iTAN of the current transaction as well as the user authentication token. The values of the message (t.authToken and t.tan) are compared to those that were stored by the login and the transfer confirmation process in attributes *userAuthToken* and *tan* earlier (2). If this check fails, a return message indicating a failure during the transfer process will be created and the invoker of the *processTransfer* operation will be informed by receiving the created message as return value of the invoked service operation (3). If the *authToken* and the *tan* are valid, the operation *debit* is called (4). *debit* is an internal operation that debits the source account. This operation is also described in an activity diagram that is not depicted due to space limitations. If the operation fails, the local variable *debited* is set to false. In this case the invoker is also informed (3). Otherwise (*debited* == *true*, 5), a local variable *ut* of class *UniqueTransaction* that is depicted in Fig. 4 is created and added to the container of incomplete transactions named *iTransactions* (6). Finally, the message *Transaction* with the unique transaction *ut* and a bank authentication token is sent to Bank B (7). *bankAuthToken* and *bankCodes* are initialized at deployment time. *bankAuthToken* is used to avoid that anyone can invoke the critical service operation *creditAccount*. Every bank service has initially such an authentication token. It consists of a signed bank code and can be verified with a predefined cryptographic operation and a suitable public key that is owned by each bank service. *bankCodes* is a key-value container of bank codes and corresponding service addresses. *sendTo* is a keyword to decide at runtime to which Bank instance a message has to be sent. This is necessary since the invoker knows several different Bank instances, which is described in the deployment diagram by the star-association to the *Bank* node (see Fig. 3).

After *Bank B* has executed the operation invocation of *creditAccount Bank A* receives the message *TransactionResult* from *Bank B* as the result of the service invocation (8). Then, the bank service *A* checks if the received value *success* is true and removes the local variable *ut* from the container *iTransactions* because the return message is arrived and a recovery for the current transaction is not needed (9). Afterwards, a message of type *TransferResult* is sent back to the invoker. If the success variable is false, the operation *credit* will be additionally invoked (10). That is necessary if a credit on the target account is not possible (e.g. due to a wrong account number) and the debit on the source account has to be undone. *debit* and *credit* are private operations and can only be invoked within a *Bank*. They are defined by other activity diagrams. Because service applications can be very complex, it is important to be able to encapsulate functionality.

A transfer can be interrupted by a disconnect of a physical connection or by an attacker who suppresses messages until a time out occurs. To ensure that a transfer is processed completely or not at all every five minutes a recovery protocol is started. This is modeled with a time event with the name *every(5, "min.")* (11). For more flexibility, it is allowed to use the time event everywhere in an activity diagram. In Fig. 5 the time event is used in the same swimlane that defines a service operation. At start, the recovery protocol checks if incomplete transactions exist by evaluating the term *[not iTransactions.isEmpty()]*. If the result is false the protocol run is finished and will be repeated after five minutes. If *iTransactions* is not empty, the first element is temporarily stored in a local variable *ut* and it is added to *iTransactions* (12). Because this transaction already exists in *iTransactions* and this container is a duplicate free list of elements the existing element will only be moved to the end of the list. Then the service operation *creditAccount* is invoked with the transaction stored in *ut* (7). From now on, the protocol is the same as the transfer protocol.

All transaction information are stored before the transactions are processed. So, if an attacker disrupts a transaction, the transaction can be repeated. But if a transaction is repeated the amount of money should not be credited again. For that, the service operation *creditAccount* stores all completed transactions and checks if the received transaction has already been executed earlier and returns the information about the success of the credit operation. Because every transaction contains a unique nonce, the transactions are unique and can not be mixed up. To avoid that anyone other than a bank service can invoke the service operation *creditAccount* every bank service has an authentication token that is initialized at deployment time.

This section described how to model a complex security critical service application in such a detail that executable code can be generated automatically. As example an online banking application was developed which describes the interaction between an account owner and the application as well as the internal processes between different bank systems and uses a recovery protocol to avoid that money is lost. To model the application, extended UML is used. To define the behavior inside UML elements like actions and guards our domain-specific language MEL is used. The security specification is embedded in the specification of the system functionality. Thus, predefined security data types (e.g. SignedData, Secret and Nonce) are used in the same UML actions where the functionality is defined. Detailed information about the online-banking example can be found on our website[2].

V. Further Modeling Details

The presented online banking example uses only stateful services but stateless services are supported as well. Whether a service is stateful or stateless is represented by an annotation of the service class (see Fig. 4) (stateless is default). Stateless means that all invokers communicate with the same service instance, which is sufficient for protocols that do not store session-dependent information.

A. Service Communication

Two ways to interact with a service are supported. Either by asynchronous message passing or by synchronous service operations.

Asynchronous message passing is modeled with a UML control flow from a UML SendSignal action to a UML AcceptEvent action. The component which can receive such an asynchronous message has to provide a public interface and listen for arriving messages. This kind of communication is more general than synchronous communication. It does not restrict an invoker to wait for a response. For example, a search engine for flight offers would send asynchronous messages to the providers and accept only the results that arrive in a specified time. But this kind of communication has the disadvantage that a component that is able to receive an asynchronous message has to define a public interface. For services this is not problematic but for invokers like a PC it could be one.

Synchronous service operations receive a message as an operation parameter and return a message as return value to the invoker. An advantage of synchronous service operations is that the invoker does not have to provide a public interface and the return message is sent implicitly to the invoker. Furthermore, in contrast to asynchronous message

[2]http://www.informatik.uni-augsburg.de/swt/projects/secureMDD/models/Onlinebanking/Application.html

passing, the local variable context is not lost after sending a synchronous message. Synchronous service operations also have the advantage that the invoker receives a result within a predefined time. Service operations can be reused in other protocols and allow a flexible service orchestration.

B. *Shared Memory*

All attributes of a stateless service are shared between all invokers. Because a stateless service can have many invokers at a time who want to access the same attributes at the same time, this access has to be synchronized. A stateful service creates a new instance for each invoker. This means that the access of the attributes does not have to be synchronized. But we allow attributes that are shared by all invoker-created service instances for one stateful service. This attributes have the stereotype ≪static≫. If this kind of attribute is used, stateful services have to manage the same challenges like stateless services. The synchronisation is managed at the code level and is described in Sec. VII-C.

VI. SECURITY

This section describes the used attacker model for service applications and shows how the TLS protocol influences the attacker abilities. Furthermore, it explains security aspects for stateful services and service operations and discusses security properties for the online banking example demonstrated in Sec. IV.

A. *Attacker*

An attacker is specified with the *Threat* stereotype (Fig. 3). He can be a full Dolev-Yao attacker [8] who is able to read, send, and suppress messages on the fly, but he can also have a subset of these abilities. Since some combinations do not make sense for services only a full Dolev-Yao attacker (read, send, suppress), an attacker who can only read messages (read), and no attacker is supported. The latter two must be used with care. A service that has only connections without the attacker ability *send* means that the attacker cannot call a service operation. The service must be deployed in such a manner that this assumption is fulfilled, e.g. behind a firewall or on a dedicated, secure network.

B. *TLS*

The attacker's abilities and connection security influence each other. If a connection has no applied Threat stereotype the connection is assumed to be secure (which must be achieved by physical means). Otherwise, if a connection has an applied Thread stereotype the connection can be secured by transport layer security (TLS [7]). This is modeled by using the stereotype ≪TLS+SSA≫ for server side authentication or ≪TLS+MA≫ for mutual authentication on a node in the deployment diagram (see node *Bank* in Fig. 3). Other nodes that want to communicate with this service have to establish a transport layer security connection to that node. It is not possible to use TLS for one connection to a service, but not another. The reason is that the connections in the deployment diagram describe only the intended communications between components. In an Internet setting with an active attacker a service can be called from anywhere. For security reasons a service requires TLS for all or none communication. Hence the stereotype is applied to the node.

If an attacker has the abilities to read, send, and suppress messages (i.e. a Dolev-Yao attacker for this connection) and the connection is secured with TLS and mutual authentication, effectively the attacker loses these abilities. TLS has several security features built in. During mutual authentication a unique session key is established. Under the assumption that an attacker cannot obtain a valid certificate from a trusted authority, he cannot succeed in establishing a TLS connection, and can never obtain a session key. After authentication messages are encrypted with the 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. This means the attacker can only read messages that are encrypted with a session key that will never be used in another session. Therefore reading the messages is useless for the attacker because he cannot decrypt them, and they cannot be used for replays as described below. (We focus on logical security properties, not traffic analysis where the message length or timing may be important.)

Furthermore, it is valid to assume that the attacker loses his 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 an incorrect sequence number. The ability to suppress messages is lost as well because the next message will also have an incorrect sequence number. However, the attacker has the ability to terminate the connection by introducing an error.

To summarize, it is appropriate to formalize a TLS secured connection with mutual authentication as one where an attacker can either do nothing or can only abort the connection. The abilities of a Dolev-Yao attacker for a connection that uses TLS with server side authentication only are slightly different. Here, only the service has to authenticate itself with a valid and trusted certificate. The invoker is not authenticated, hence an attacker can establish a secure TLS connection with the service. However, if a session between a client and a server has already been established an attacker can only abort the connection as in the case of mutual authentication.

C. *Stateful Services*

A stateful service creates a new instance of itself for each invoker. Technically, the invoker calls a manager which is realized as a stateless service. The manager creates an instance of the actual service, deploys it and returns the address of this service instance. It is not necessary to model this behavior in the UML model of the application. The indirection also does not introduce additional attacker possibilities. Only the address of the new service instance is transmitted which is not a security critical information leak because the attacker already knows all components in the formal model. However, an attacker who has the ability to send messages to the service can also access an already created stateful service instance using the address even if TLS with server side authentication is used. This is against the idea of a stateful service, but must be considered by the application developer in the protocols.

D. Service Operations

The communication mode influences the attacker abilities. If a component invokes a service operation synchronously it transmits the arguments for the operation and blocks until the response is received. If the attacker has the ability to send messages then he can inject a response only after the operation is called and only before the genuine answer arrives (and only if TLS is not used). Additionally, the answer is checked to be of the correct type (for example *TransactionResult* in step (8) in Fig. 5). If the type is not correct an error is raised which limits the attacker possibilities. If asynchronous messages are used, the attacker is able to send a message at any time.

E. Security Properties

The online banking application is obviously security critical, even though no explicit cryptographic operations are used. A customer (AccountOwner) and the banks have different security requirements. The customer has two major requirements:

- Only intended transfers take place, i.e. transfers initiated by the customer with the correct amount and recipient. This precludes phishing (and other) attacks.
- No money is lost, i.e. if the money cannot be credited to the receiving account it is refunded.

The major requirement for the banks is:

- The same amount of money that is credited to one account has been debited from another account.

The idea is that the customer requirements are guaranteed by using TLS, passwords, and iTANs in an online transfer (Fig. 2). But are they? Actually, this depends on the assumptions about the attacker and the behavior of the services. And, it is very easy to make mistakes even in very short security protocols [13]. We make the following assumptions about the attacker:

- An attacker can learn the password of a customer, but not his list of iTANs.
  Actually, iTANs were introduced as an additional security mechanism because passwords often do not remain secret.
- An attacker cannot learn a private TLS key of a bank or obtain a valid TLS certificate from a trusted authority.
  This makes phishing attacks impossible because the application running on the customer's PC will only use TLS connections to a trusted server.

However, there is another possible attack because of the interplay between a TLS session and stateful services. The protocol in Fig. 2 explicitly closes the TLS session after a transfer. But this does not mean that the service instance used in the protocol is also destroyed. This service instance still stores the transfer data e.g. the source account. An attacker that is able to learn the URL of the instance could open a new TLS connection. After logging in with his own password (i.e. he also has an account at that bank) he might gain access to the stored attribute values. In the example he can actually perform a second transfer from the account of the previous customer using one of his own iTANs. Of course, a simple solution is to reset all values at the end of the protocol. Still it is easy to overlook that an attacker can possibly continue a session.

The transfer of funds between two banks (Fig. 5) also uses TLS and additionally an authentication token that must be kept secret. Still, there is a possible attack by actively delaying a message. The idea is to exploit the recovery mechanism for failed transactions, and to get refunded twice. The attacker, who is also a valid bank customer, initiates a transfer to an non-existing target account. The target bank answers with a failure (`success == false` in step (7) in Fig. 5). Now the attacker delays this message until the recovery mechanism (10) starts. The second attempt to finish the transfer will also fail, but the attacker does not interfere. So the money will be refunded to the attackers account. Then the attacker sends the delayed message from the first attempt and is refunded a second time. A solution would be that the recovery protocol can start only after the transaction fails. That can be realized by adding the transaction to the list of incomplete transactions only after receiving an exception from the invoked service operation *creditAccount*.

Once the problems are known they are easy to solve. The two example attacks try to demonstrate that even in a moderately small application the interplay between standard cryptographic protocols, service behavior, and an active, malicious attacker makes a security assessment difficult. We propose to use formal methods to prove the security of such applications.

VII. Executable Code

One advantage of SecureMDD compared to some other model-driven approaches (see Section III) is that runnable code can be generated automatically. This is achieved by modeling an application in detail with UML and MEL.

The Eclipse modeling Framework EMF[3] is used as the development environment for code generation. The platform-independent model is transformed into platform-specific models (PSMs) with the model-to-model (M2M) transformation language QVT [10]. Model-to-text (M2T) transformations based on oAW[4] are used to generate the executable code. Each PSM is transformed into one or more packages that contain the full source code for each component type.

A. Services and Service Communication

The service components are implemented as Java Web Services that use SOAP [15] as underlying technology. To implement Web Services Metro[5] that integrates JAX-WS [21] (Java API for XML - Web Services) is used. JAX-WS is a standard and supports server and clients and can be used by

[3]http://www.eclipse.org/modeling/emf/
[4]openArchitectureWare: http://www.openarchitectureware.org/
[5]http://metro.java.net/

annotated plain old Java objects (POJO) 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 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 (but beware of the attacker, cf. Sect. VI). The time out of services and service invokers is set to ten minutes but can be changed at deployment.

From the modeled service operation in Fig. 5 two Java Web Service methods are generated, one for processing a transfer and one for recovering incomplete transactions. The body of the second method is shown in listing 1.

```
if (! getITransactions().isEmpty()) {
   UniqueTransaction ut = getITransactions().at(0);
   getITransactions().add(ut);
   TransactionResult inmsg = getProxy(getBankCodes()
      .get(ut.getTransfer().getTo().getBankCode()),true, true)
      .creditAccount(new Transaction(ut, getBankAuthToken()));
   boolean success = inmsg.getIncreased();
   if (success) { }
   else {
       credit(ut.getTransfer().getFrom().getAccountNumber(),
              ut.getTransfer().getMoney());
   }
   getITransactions().remove(ut);
   tr = new TransferResult(success);
}else { }
```

Listing 1. Recovery protocol operation

Because invocation of this method is triggered by time and not by an incoming message it is realized as an operation that can be invoked by the service manager only. The manager creates one instance of the stateful service, but does not deploy it. Every 5 minutes the recovery method will be executed. The method does not return a value because no external invoker exists. Even if the return value of type *TransferResult* is modeled no code is generated for it. The invocation of the service operation of the receiving bank is done with the help of the method *getProxy*. It is called with the address of the service to invoke as well as the information that a session has to be opened and closed after receiving the result. *getProxy* returns a stub object of class *Bank* on which the service operation *creditAccount* is invoked.

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.

B. Deployment

The deployment diagram contains all important information to deploy an application (some additional information is defined in the class diagram). To deploy the online banking application from Sect. IV the following information is important. The application model contains a user that represents a real human, and two deployable components. Each of these components can be instantiated multiple times, i.e. an arbitrary number of *AccountOwnerPC* applications and *Bank* services can be deployed. A bank service can be accessed by many account owner PCs at the same time and can invoke his own operations or other bank services. The communication between a user and a PC has to be secured against an attacker (i.e. against shoulder surfing). An account owner PC can communicate with a bank service over any kind of network, and bank services can also communicate over any kind of network.

For the account owner PC a Java package containing all necessary code is generated. The class *AccountOwnerPC* can be instantiated on any device that supports Java and is secure 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 server should run in a secure environment. 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.

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 bank service these are all initial accounts, login data, an authentication token for accessing other bank service as well as a public key to verify such a token and a container that maps bank codes to bank service addresses. These attributes must be passed as parameters to the constructor before the service is started.

The online banking application from Sect. IV uses predefined containers to store information like accounts or incomplete transactions with a predefined interface. The generated code provides a prototypical implementation. It can be replaced by any other implementations or databases that implement the 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 until testing the code can be invoked with one click inside Eclipse. In our test framework all services are deployed on one light-weight server that is integrated in Java, but of course it is possible to deploy the services on other servers.

C. Parallel Service Invocation

Because services can access shared memory at the same time, it is important to consider parallelism. Especially, transactions with read and write access have to be executed atomically. A possible solution is that the developer of an application has to manage the synchronisation. A more comfortable way is that the synchronisation is managed automatically by the code generation. Hence, we chose the latter solution. Our current and safe 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 body 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.

VIII. Conclusion and Future Work

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. This paper describes the modeling of secure service applications with extended UML and a domain-specific language named MEL. It also demonstrates the need to verify application-specific security properties despite the use of standard protocols e.g. TLS. Furthermore, from a platform-independent UML model runnable code as well as a formal specification will be 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. Hence, we focus on the generation of secure and executable service applications.

The transformations for code generation are in an advanced stage and will be completed soon. For smart card applications we already generate a formal model [17]. For simple service applications we also generate a formal model and verified some application-specific properties. For more complex service applications like the online banking example some work remains to be done.

References

[1] M. Alam, R. Breu, and M. Breu. Model driven security for web services (mds4ws). In *Multitopic Conference, 2004. Proceedings of INMIC 2004. 8th International*, pages 498–505. IEEE, 2004.

[2] K. Baina, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In *Advanced Information Systems Engineering*, pages 527–543. Springer, 2004.

[3] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering*. Springer LNCS 1783, 2000.

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

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

[6] M. Deubler, J. Grünbauer, J. Jürjens, and G. Wimmel. Sound development of secure service-based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 115–124. ACM, 2004.

[7] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. IETF Network Working Group, August 2008. URL: http://www.ietf.org/rfc/rfc5246.txt.

[8] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. In *Proc. 22th IEEE Symposium on Foundations of Computer Science*. IEEE, 1981.

[9] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-driven web services development. In *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*, pages 42–45. IEEE, 2004.

[10] O. M. Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, 2011.

[11] F. Huber, S. Molterer, A. Rausch, B. Schatz, M. Sihling, and O. Slotosch. Tool supported specification and simulation of distributed systems. In *Software Engineering for Parallel and Distributed Systems, 1998. Proceedings. International Symposium on*, pages 155–164. IEEE, 1998.

[12] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[13] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop (TACAS)*, pages 147–166. Springer LNCS 1055, 1996.

[14] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-Driven Service Orchestration. In *Proceedings of 12th IEEE International EDOC Conference (EDOC 2008)*. IEEE press, 2008.

[15] N. Mitra and Y. Lafon. *SOAP Version 1.2*. W3C, 2007.

[16] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *Workshop on Secure Software Engineering, SecSE, at ARES 2009*. IEEE Press, 2009.

[17] N. Moebius, K. Stenzel, and W. Reif. 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*. Springer LNCS 5965, 2010.

[18] A. Nadalin, M. Goodner, M. Gudgin, A. Barbir, and H. Granqvist. *WS-SecurityPolicy 1.2*. OASIS, 2006.

[19] A. Nadalin, C. Kaler, P. Hallam-Baker, and R. Monzillo. *Web Services Security: SOAP Message Security 1.0*. OASIS, 2004.

[20] Y. Nakamura, M. Tatsubori, T. Imamura, and K. Ono. Model-driven security based on a web services security architecture. In *IEEE International Conference on Services Computing*, pages 7–15. IEEE Press, 2005.

[21] R. M. Roberto Chinnici, Marc Hadley. *The Java API for XML-Based Web Services (JAX-WS) 2.0*. JCP, 2006.

[22] Q. Sheng and B. Benatallah. Contextuml: a uml-based modeling language for model-driven development of context-aware web services. In *Mobile Business, 2005. ICMB 2005. International Conference on*, pages 206–212. IEEE, 2005.

[23] Sun Microsystems Inc. *Java Card 2.2 Specification*, 2002. http://java.sun.com/products/javacard/.