Universität Augsburg

SecureMDD: A Model-Driven Development Method for Secure Smartcard Applications

N. Moebius, H. Grandy, W. Reif, K. Stenzel

Report 10 2008

Institut für Informatik
D-86135 Augsburg

SecureMDD: A Model-Driven Development Method for Secure Smartcard Applications

Nina Moebius, Holger Grandy, Wolfgang Reif, Kurt Stenzel
Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, 86135 Augsburg, Germany

Abstract

In this paper we introduce a method to apply model-driven ideas to the development of secure systems. Using MDD techniques, our approach, called SecureMDD, provides a possibility to verify the correctness of a system at the modelling stage. To do so, we generate different platform-specific models from one common platform-independent UML model. The considered platforms are JavaCard and a formal model. The formal model is used for the verification of security properties. For the verification results to carry over to the Java(Card) code, these models have to be equivalent with respect to security aspects. This requires complete code generation without the possibility to manually complete the Java(Card) code. To devise such sophisticated models, we extend action elements of activity diagrams. In this paper we focus on the part of our approach which is used to generate secure smartcard code.

Chapter 1

Introduction

Model-driven development has become one of the most promising approaches to handle the complexity of computer systems. This is achieved by rigorous application of domain modeling techniques and accomplished by using transformation functions e.g. for the generation of executable code. Especially the domain of security-critical distributed systems is a promising field for the application of model-driven development because the required quality of the applications and their correctness regarding specifications is crucial. Applications of this domain are often E-Commerce applications, for example electronic payment systems or electronic ticket systems. But also applications such as the German electronic health card which is realized as service-oriented architecture and has to cope with security aspects such as secrecy and role-based access control fit into this domain. These applications have in common that they are based on cryptographic protocols. These protocols are very difficult to design.

In this paper, we present SecureMDD, which is a model-driven technique for developing security-critical applications. In our opinion, only by integrating security considerations from the very beginning of application development the resulting product can be really secure. On the other hand, only by using established modeling techniques already elaborated in the Software Engineering research, the inherent complexity of such applications can be handled with more ease. Then, design errors, e.g. in the design of the security protocols, are less likely to occur.

Besides just treating the generation of secure executable code we furthermore extend our approach to the model-driven generation of formal specifications from the same platform-independent model. Using this formal model we verify certain security properties for the modeled application. For verification techniques our group already developed a formal approach [11]. On the other hand, model-driven development also allows for generation of executable code. Our goal is to generate code that is, in terms of refinement, correct with respect to the formal specification. We already developed a refinement methodology for hand-written code and model [9]. By extending this methodology to generated code and model we are bringing together formal methods and model-driven Software Engineering into one integrated approach.

To the best of our knowledge this is the first approach dealing with the integration of formal verification of security aspects and the generation of executable code. To be able to prove the security of the modeled application and generate an implementation of it at the same time, we have to be able to completely model an application, including method bodies. For this reason we extend UML activity diagrams by a simple language which allows the expression of state changes. Note that our aim is to automatically generate an implemenation of the security-critical parts of the application, we do not consider user interfaces, data base access and so on.

Significant research has already been done on the application of model-driven development to security-critical applications, e.g. [17] [20] [2] [19]. But none of those approaches integrates the application of formal techniques on the one hand and the generation of correct code on the other hand. We will give a detailed comparison to those approaches in Section 6.

We currently focus our work on smartcard scenarios. Smartcards are inherently security-critical, but also relatively small in their size and therefore easier to cope with. In a second step, we plan to extend our method to other areas of security-critical applications, e.g. using Web Services. All considered applications have in common that they are built on application-specific security protocols.

In this paper we introduce a method to generate executable JavaCard code from a platform-specific model. This PSM is generated from a platform-independent model that also serves as source model for a formal specification. The platform-specific model can be completed by platform-specific information but this does not cause any inconsistencies with the platform-independent model. The introduced approach can be applied to all (smartcard) applications that are based on cryptographic protocols.

Section 1.1 presents our integrated model-driven approach and introduces the different modeling levels. Section 2 gives an overview of the specifics regarding the programming of Java smartcards. We illustrate our approach by a case study called Mondex which is an electronic payment system and shortly introduced in Section 3. Section 4 presents our modeling of security-critical systems with UML which we extend to be able to completely model the system. In Section 5 we describe the generation of executable JavaCard code from these models. Section 6 introduces work that is related to ours and Section 7 concludes.

1.1 From Abstract Models to Secure Executable Code

A lot of research work, for example [22] and [23], addresses the abstract specification of security protocols as well as the proof that the specified systems are secure. Most of these approaches deal with proofs of security properties only at the level of abstract specifications but do not consider the implementation of the system. In practice, this does not suffice since additional weaknesses may be added on the code level. Initially, we tried to bridge the gap between abstract specification and code by adopting a refinement technique and using interactive verification. This approach guarantees that security properties of the abstract specification are also valid at code level [12] [10]. This work turned out to be successful but also very time-consuming. Therefore, our new approach aims to generate both, a formal model for verification as well as executable code, from a common platform-independent UML model. We define the formal specification as well as the smartcard application to be two different platforms. We generate platform-specific models for each platform from the platform-independent model and, in a next step, the formal model resp. code. Figure 1.1 gives an overview of our approach.

The platform-independent level defines an abstract view of the application under development including dynamic and static aspects of all involved components. This model is transformed into a platform-specific model that contains information needed for the generation of JavaCard code for the smartcard as well as Java Code for the component communicating with the card. Thus, we generate platform-specific models defining the behavior as well as data types for each component of the application. In this paper we focus on the generation of the smartcard part of an application but our approach is extendable to generate the Java code runnable on a smartcard terminal as well. Then, the JavaCard code, resp. Java code for the terminal, is created by model-to-text transformation from the platform-specific Card PSM resp. Terminal PSM.

Furthermore, the platform-independent models are translated into a platform-specific model, Formal PSM, containing the required information to generate a formal model based on abstract state machines [13] [4]. In a next step, after generating the formal specification from the Formal PSM by model-to-text transformation, security properties can be specified and verified using the formal model in the interactive verification system KIV [1]. For the formal specification as well as the verification we adapt the specification methods and techniques developed in the Prosecco approach. Here, a formal specification of a security-critical application is given as an algebraic specification in combination with abstract state machines. An overview of the approach can be found in [11].

In the following we focus on the code generation for smartcards and the corresponding platform-specific modeling. The transformations are not yet implemented but the concepts introduced here are going to be used as templates for their realization.

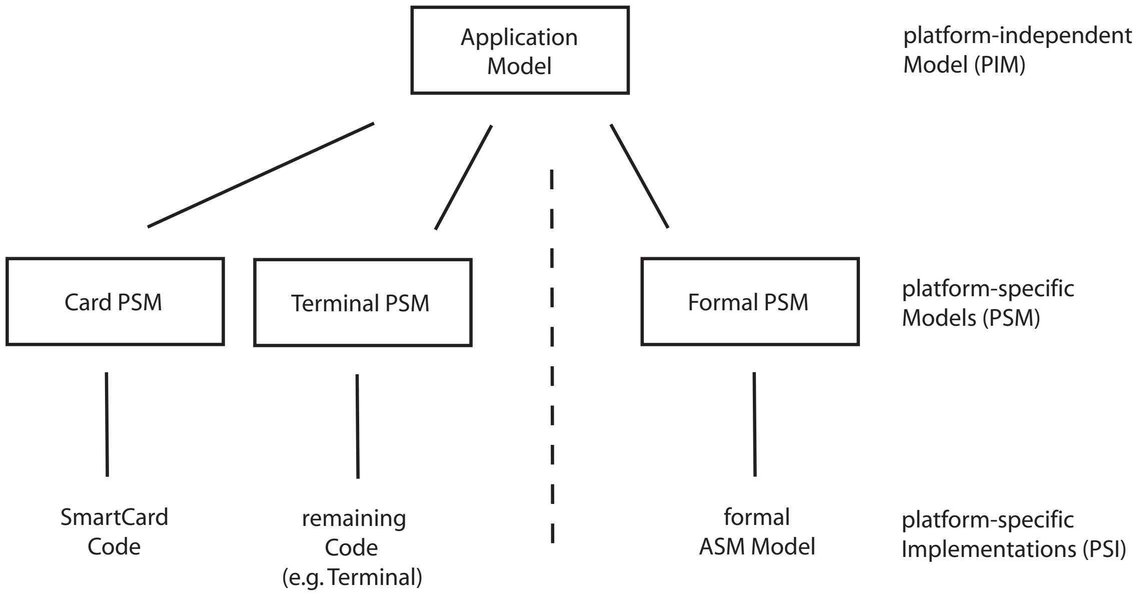
Application Model
platform-independent Model (PIM)
Card PSM
Terminal PSM
Formal PSM
platform-specific Models (PSM)
SmartCard Code
remaining Code (e.g. Terminal)
formal ASM Model
platform-specific Implementations (PSI)

Figure 1.1: Overview of SecureMDD

Chapter 2

Smartcards using JavaCard in a Nutshell

The JavaCard [25] technology facilitates the use of Java on resource-constrained devices such as smartcards. Since these small devices are not very powerful, only a subset of the functionality of Java is supported. Unsupported functions are, for example, garbage collection, dynamic class loading and threads. Even the use of large primitive types such as integers is unsupported. Another difference between JavaCard programs and Java programs is the limited storage space on the smartcard. To avoid memory leaks best practice is to allocate all memory during the initialization phase.

Smartcards communicate with a terminal by receiving and answering commands using Application Protocol Data Units (APDUs). The card does not initiate the communication, instead it waits until receiving a command APDU from the terminal. Then, the card processes the command APDU and returns a response APDU. Typically, JavaCard programs are not written using object-oriented paradigms. Instead, byte arrays and primitive data types such as `shorts` are manipulated directly.

Chapter 3

An Electronic Payment System: Mondex

We illustrate our approach with an example, namely an electronic purse system called Mondex [6]. Mondex is a product of Mastercard International [15] and is used to replace coins by electronic cash. The Mondex case study recently received a lot of attention because its formal verification has been set up as a challenge for verification tools [26] that several groups [16] as well as our group [24] [14] worked on. To pay with his Mondex card, the customer of a shop inserts his card into a card reader which is also connected to the Mondex card of the shop owner. Then, the amount payable is withdrawn from the card of the customer and debited to the one of the shop owner. To ensure the practicability of the purse system, several security properties have to be considered. At first, it must not be possible to "create" money, i.e. to add money on one card without reducing the amount of another card. Furthermore, it has to be ensured that "no money is lost". If, for example, a card is removed from a card reader too early, a recovery mechanism has to guarantee that the lost amount can be recovered. Another point is the requirement that a Mondex card, after released to the customer, is on its own, i.e. it has to ensure the security for all transactions without the help of a central server.

Chapter 4

Modeling of security-critical applications with UML

The modeling methodology of our approach differs from the work of other groups (which is discussed in Section 6) by providing a way to completely model an application. This includes the static aspects as well as dynamic views.
UML facilitates the description of a system from different views. In our approach the following types of diagrams are used: Use case diagrams and descriptions are used to give an overview of the functionality of the system under development. Class diagrams are used to model the static view of an application. Sequence diagrams and activity diagrams are used to model the dynamic aspects of the system. Deployment diagrams serve for definition of the structure of the (distributed) system under development and to model the attacker abilities that are needed to prove that the system is secure. The aim of our modeling approach is to extend the Unified Modeling Language to be able to model security-critical applications that are based on cryptographic protocols.
Static aspects of a (distributed) application, modeled in class diagrams, are the components of the system, i.e. smartcards and terminals, that communicate to run a protocol, the message types that are used for communication as well as the data types. Specifics regarding the modeling of a security-critical application are the representation of encrypted data, digital signatures and hashed values. Furthermore, we need a facility to represent nonces (random numbers) as well as private, public and symmetric keys. We solve this by defining UML stereotypes to model the encryption, signing and hashing of data as well as appropriate data types to represent keys, secrets and nonces. To be able to model the processing of a received message (resp. to automatically generate method bodies) we define a language to extend UML activity diagrams such that it is possible to express state changes like variable assignments. The language is tailored to security-critical applications. For example, primitives exist to express the encryption and decryption of data. Furthermore, we use UML stereotypes and tagged values to add security-specific information to UML model elements.

In this Section we illustrate our methodology to model a security-critical application with UML. After presenting the platform-independent modeling, we discuss the addition of information on the platform-specific level. In this paper we concentrate on the smartcard part of the UML model. Since JavaCard code in our approach is mainly generated from class and activity diagrams, in the following the use of these diagrams is introduced in detail.

4.1 Platform-independent Models

Figure 4.1 shows the part of the class diagram representing the smartcard and associated classes for Mondex.

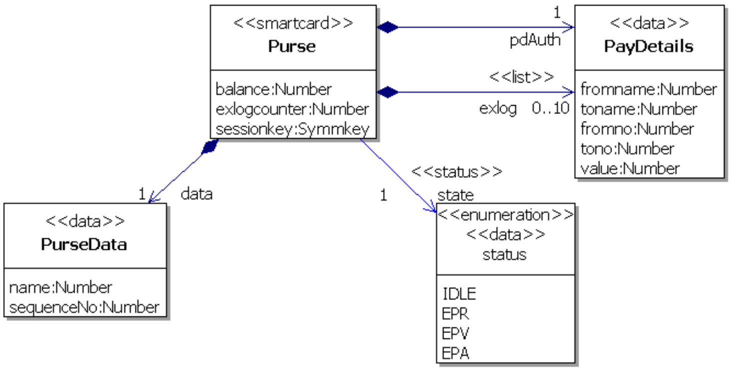

Figure 4.1: Part of the platform-independent class diagram consisting of the purse and its associated data types

The class representing the smartcard component, called `Purse` in Figure 4.1, is denoted by the stereotype ≪smartcard≫. Furthermore, the associated data types are denoted by stereotype ≪data≫. A purse stores its name and a continuous transaction number in an object called `PurseData`. Each component of the system that participates in the protocol run requires a status that indicates the state the component is in. This state is given as an enumeration of all possible states and the association is annotated with the stereotype ≪status≫. Moreover, the current transaction details are recorded by a data type called `PayDetails`. If a transaction fails, the current transaction details are stored in a list of `PayDetails` (denoted by stereotype ≪list≫) where at most 10 failed transactions can be logged. Furthermore, a purse stores its balance, a symmetric key which is the same on all cards as well as a counter that counts the number of failed transactions. On the platform-independent level all types of numerical values are defined by an abstract data type `Number`. For strings we use the abstract type `String`. These values have to be refined on the platform-specific level.

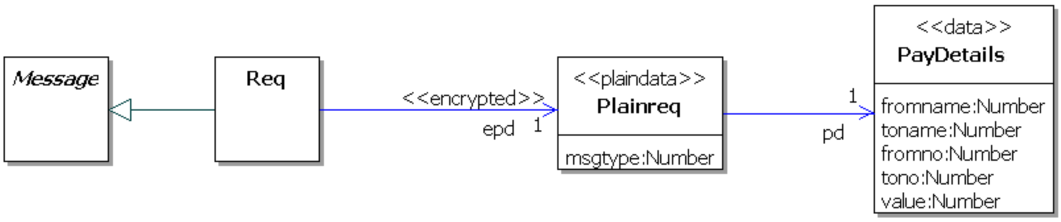

Figure 4.2: Message type Req with associated classes

During a protocol run messages are exchanged between the protocol participants. On the platform-independent level each message type is modeled by a class derived from an abstract class `Message`. Figure 4.2 shows one message type of the Mondex application, called `Req` (= request). A `Req` message is sent from the TO purse to the FROM purse to initialize a transfer of money. A `Req` message contains encrypted data. Data that is going to be encrypted is labeled as plaindata in the diagram using the stereotype ≪plaindata≫. In our case the data to be encrypted is of type `PlainReq`. An object of this type contains the `PayDetails` of the current transaction as well as a constant `msgtype`. During a protocol run an object of type `Plainreq` is going to be encrypted. The `msgtype` information is needed to prevent replay attacks since there are other messages in the protocol that contain the encrypted pay details of the transaction but use another `msgtype`.

To model the exchange of messages between the protocol participants we use UML sequence diagrams. This part of the dynamic modeling which only shows the message flow is then used to generate the skeleton of an activity diagram for each protocol by applying a model-to-model transformation. Then, the generated diagrams are completed by the developer. Note that we use activity diagrams instead of state machines because we emphasize the protocol flow between the participants which can easily be visualized when using activity diagrams. Furthermore, since we define the state changes of every participant in detail, activity diagrams are suitable to refine the modeled message-based view.

Here, we support the use of `activity partition`s to have partitions for each component participating in the protocol and `sendSignal` as well as `acceptEvent` actions to model the sending and receiving of messages. Furthermore, we provide support for `actions` to express assignments and object creation, `decision nodes` as well as `initial` and `final nodes` to model the begin and end of an activity. The use of parallel control flows is not allowed.

As noted earlier, we extend activity diagrams by the ability to express state changes. We exploit the fact that the content of some model elements, for example the action elements, has no predefined syntax and semantics. Instead, the OMG allows the use of arbitrary strings. We use this omission to add a simple abstract programming language that is used to specify the processing of a message. The syntax of the language is similar to Java syntax but limited to simple expressions and statements. When generating a formal model from the modeled application, the UML diagrams resp. the activity diagrams extended by the self defined language, are translated into abstract state machines [4]. ASMs have a well-defined and relatively simple semantics. The semantics of our models are defined by giving a mapping from the semi-formal UML descriptions into a formal presentation using abstract state machines.

The language facilitates the extension of `sendSignal` and `acceptEvent` elements by denoting the type of message that is sent resp. received and the assignment of message fields to local variables after receiving a message. For `action` elements assignments, the application of predefined arithmetical operations, the creation of objects as well as method invocation of predefined and self-defined methods is expressible. Complex state changes can be modeled using subdiagrams to improve readability. Furthermore, the language allows the definition of boolean expressions that are used as conditions for decision nodes. For example, updates of the state of a component, checks of preconditions that have to be satisfied as well as decryption of data is expressible. If the modeled protocol contains loops, these have to be defined in a separate activity (sub-)diagram.

Figure 4.3 illustrates the use of activity diagrams to describe a protocol run. For each component participating in the protocol we use a partition (which is generated from the sequence diagram). The Figure shows one detail of the partition of the purse component. The purse increases its field `sequenceNo` which is part of the `PurseData data` (see Figure 4.1). The sequence number is used to have a unique identifier for each transaction and to avoid replay attacks. Afterwards the `state` of the `Purse` is set to `EPV` which stands for "expecting value" and denotes that the purse is ready to receive a `Val` message. Then, an object `plain` of type `Plainreq` is created with `msgtype REQ` (which is defined as a constant in the class diagram) and the current pay detail `pdAuth` of the purse as parameters. Afterwards, the `Plainreq` object is encrypted with the symmetric key `sessionkey` (which is a field of class `Purse`). The encrypted data is sent as `Req` message. The receiving and sending of messages is also generated on the basis of the sequence diagrams. We use certain predefined methods that can be used in the activity diagram such as encrypt or decrypt. It may happen that complex algorithms are used that are difficult to model with activity diagrams. For this reason, the developer is allowed to add self-defined methods that are not specified resp. modeled on the platform-specific level. These have to be annotated with stereotype ≪selfdefined≫ in the activity diagram. For each self-defined method the model-to-text transformation generates a method signature. The code of this method has to be added by the developer on the platform-specific implementation level.

To define the number of components in the system, the communication links as well as the attacker abilities, we use UML deployment diagrams. Since the deployment diagram is more important for the generation of the formal model than for code generation, we omit the details

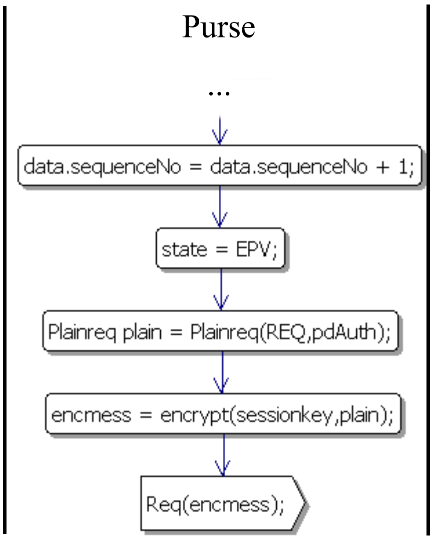
Purse
...
data.sequenceNo = data.sequenceNo + 1;
state = EPV;
Plainreq plain = Plainreq(REQ,pdAuth);
encmess = encrypt(sessionkey,plain);
Req(encmess);

Figure 4.3: Part of the platform-independent activity diagram of the Mondex application

and refer to [21].

4.2 Platform-specific Models

The platform-independent models are transformed into different platform-specific models. For each component of the distributed system we generate one platform-specific model that is later used to automatically generate code. This platform-specific view contains the parts of the system under development that are relevant for the component. Furthermore, some platform-specific details are added by the transformation function. For example, associations annotated with stereotype ≪list≫ in the platform-independent model are realized by arrays. Also, constructor signatures are added in the class diagram. The activity diagram models are refined to models containing JavaCard code. Another aspect that has to be handled is the retrenchment of the abstract primitive type `Number` used in the platform-independent model to platform-specific types. JavaCard only supports the primitive types `boolean`, `short` and `byte`. The use of e.g. integers, characters as well as strings is not possible. By transforming the platform-independent model into a platform-specific model for the smartcard, all fields of abstract type `String` are replaced by byte arrays, e.g. ascii representation. All fields of abstract type `Number` are replaced by the primitive Java type `short`. If the developer prefers to use `byte`s for some fields, he is allowed to change this default type to `byte` in the platform-specific model. For example, the `sequenceNo` of a `Purse`, stored in the `PurseData` object, is of type `Number` in the platform-independent class diagram. In the platform-specific model this field is transformed into a field of type `short` as default value. The developer may change it to `byte`. During a protocol run different arithmetical operations may be performed on these primitive fields. Since the JavaCard types are bounded, an over- or underflow may occur. To prevent this, we add checks for over- and underflow for each arithmetical operation. For example, if two values are added we check if the result is within the valid range. If an over- or underflow is caused, an exception is thrown. If an expression within the program consists of more than one arithmetical operation, a range check is added for each operation. The checking methods for addition, subtraction, multiplication, division and remainder are application-independent and have to be implemented only once. To avoid replay attacks, for every transaction of money a different `sequenceNo` is used. Thus, for each transfer of money the `sequenceNo` field is incremented on both purses. In the platform-specific model, we add a method call of method rangeCheckAdd(short x, short y) that checks if the addition of 1 to the sequenceNo causes an over- or underflow.

```
public static void rangeCheckAdd(short x, short y){
if(x > 0 && y > 0 && (short)(x+y) < 0){
  ISOException.
    throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);}}

if(x < 0 && y < 0 && (short)(x+y) >= 0){
  ISOException.
    throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);}
```

Since JavaCard does not support integers, it is not possible to check whether the result of the addition is greater than the maximal short value, e.g. 32767. If the addition causes an overflow, the resulting short value will be less than zero. To be consistent to the formal model we have to ensure that the JavaCard program behaves in the same way as the formal specification given as abstract state machine. For this reason, when adding a range check in the JavaCard program, we do the same in the ASM model although the formal model uses unbounded integers. The checks in the formal model are realized by testing if the arithmetic operation produces an output that is within a valid range. Another platform-specific detail that is added by the developer is the specification of the used encryption algorithm, padding scheme etc. These information are added in the activity diagram models. Since the cryptographic operations are modeled as abstract operations in the formal model, the addition of these information does not cause any inconsistencies neither with the formal model nor with the platform-independent model.

To model our applications with UML and to define our UML profile we use the modeling tool Magic Draw. To implement our model-to-model transformations we use operational QVT, for model-to-text transformations we make use of the language XPand. Both transformation languages are part of the Eclipse Modeling Project [8].

Chapter 5

Automated Generation of Code

Programming smartcards (with JavaCard) is different from programming Java with J2SE. Although JavaCard is a subset of Java, coding techniques differ a lot. JavaCard programs are usually not written in an object-oriented manner. Typically, Java syntax is used to manipulate byte arrays directly omitting object-oriented paradigms like modularization and encapsulation. In our opinion, one challenge of model-driven code generation approaches is to reduce the gap between input and target platforms. For this reason, we decided to make further use of the classes defined in the platform-independent models (and later transformed to platform-specific classes) instead of flatten out the object-oriented view of the application into a program consisting of byte array representations for each object resp. class. Thus, in our approach the class implementing the protocol steps of the cryptographic protocol operates on the data types defined in the platform-independent model by the developer.

To generate the JavaCard code of the protocol implementations we use the protocol specification given as an activity diagram. Since every protocol of the application is modeled as an activity diagram and completely specified using our protocol definition language, it is possible to generate executable code.

The JavaCard API provides several facilities to encrypt data, create digital signatures or generate hash values. The cryptographic operations are implemented on byte arrays. That is, the data to be encrypted resp. signed or hashed has to be converted into a byte array representation before applying the cryptographic operation. That is, all data types annotated with stereotype ≪plaindata≫ in the class diagram have to be converted into a byte array representation before applying the encryption operation. To accomplish this, we add methods for encoding and decoding of plain data objects that follow the same rules as the en- and decoding of message objects. Then, we define a decrypt operation that returns an object of type plain data. If the decryption fails or the decrypted byte array is not a valid representation of an object of type plain data the return value is null. Referring to the class diagram (see Figure 4.2) and the activity diagram (see Figure 4.3) a `Plainreq` object is encrypted by transforming it into a byte array and applying a JavaCard method that returns the encrypted value as a byte array. If the application stores an encrypted value, this is done using a special data type `EncData` that just contains the encrypted byte array.

Listing 5.1 shows the JavaCard code (of class `Purse`) that is generated from the part of the activity diagram shown in Figure 4.3. In line 2 the `sequenceNo` of the `PurseData` with name `data` is incremented. Before, it is checked if the addition of 1 causes an overflow by calling the method `rangeCheckAdd(short x, short y)`. Then, the `state` of the purse is set to `EPV`. In a next step, the sending of a `Req` message is prepared (line 5). An object `plain` of type `Plainreq` is requested from the ObjectStore. The ObjectStore preallocates all required messages needed for protocol execution at the initialization phase. Then, the `plain` object is encrypted with the `sessionkey`, which is a field of the `Purse` class. The returned object is of type `EncData` which has one field of type byte array that contains the encrypted result of the operation. The `EncData` object is requested from the ObjectStore within the `encrypt` method. Afterwards, the

```
 1 Checker.rangeCheckAdd(data.sequenceNo, (short)1);
 2 data.sequenceNo = (short)(data.sequenceNo + 1);
 3 state = EPV;
 4
 5 Plainreq plain = ObjectStore.
 6           newPlainreq(Constants.PLAINREQ,pdAuth);
 7 EncData encmess = (EncData) Crypto.
 8                  encrypt(sessionkey, plain);
 9
10 Message outmsg = ObjectStore.newReq(encmess);
11 comm.sendMsg(outmsg);
12 ObjectStore.returnEncData(encmess);
13 encmess = null;
14 ObjectStore.returnPlainData(plain);
15 plain = null;
16 ObjectStore.returnMessage(outmsg);
17 outmsg = null;
```

Listing 5.1: JavaCard code generated from the activity diagram

ObjectStore is asked for a `Req` message that contains the encrypted value (line 10). The returned message is sent using the `sendMsg(Message msg)` method of the communication interface `comm` which encodes the `outmsg` into a byte array representation. This byte array is written into the output buffer. The content of this buffer is sent to the terminal at the end of the protocol step. Then, the requested objects are returned to the ObjectStore (line 12,14 and 16) and the references pointing at this object are set to `null`. The `encrypt` method as well as the `sendMSg` method do not change the values of the fields of the objects `plain` and `encmess` and do not store these objects or field values without copying it. Otherwise, there would exist references to objects that were returned to the store and therefore ”free”. Furthermore, when receiving a new `Plainreq` object from the ObjectStore (line 5), the passed `PayDetails pdAuth` has to be copied to the `Plainreq` field `pd` (see Figure 4.2) because otherwise a reference to the `pdAuth` object of the `Purse` class would exist. If the `Plainreq` object is later returned to the ObjectStore, the store would manage this object with reference to the PayDetails `pdAuth` and may issue the `Plainreq` object again. This may cause side effects.

Chapter 6

Related Work

The most closely related work is UMLSec developed by Jan Jürjens [17]. To model security-critical systems with UML and to prove that several (predefined) security properties hold for the modeled system, Jürjens defines a UML profile. Using the profile, properties such as secrecy and integrity as well as role-based access control are expressible. Jürjens provides tool support for verifying properties by linking the UML tool to a model checker resp. automated theorem prover. Moreover, model-based testing and non-interference analysis is part of the approach. In [3] the employment of the UMLSec approach in an industrial context is presented. The security properties mainly addressed in UMLSec are those standard properties which are expressible by using the predefined stereotypes. The generated formal model reflects an abstract view of the parts of the application that are required for verification. In our approach we generate a formal model of the entire application which can be used to express and verify application-dependent properties such as "No money can be created within the Mondex application". Another difference is the integrative aspect of our approach. We aim to generate secure code as well as a formal model for verification whereas Jürjens mainly focuses on the generation of a formal model and, based on this model, to verify security aspects.

In [18] another approach of the same author is presented. Here, an implementation of an application is written in Java (by hand). Then, the code is translated into an abstract model of the application. The generated abstract model is used to prove security properties using an automated theorem prover. The approach is evaluated by two case studies, an electronic purse system and an implementation of the TLS protocol.

Basin et al. [2] [20] present a model-driven methodology for developing secure systems which is tailored to the domain of role-based access control. The aim is to model a component-based system including its security requirements using UML extension mechanisms. To support the modeling of security aspects and of distributed systems several UML profiles are defined. Furthermore, transformation functions are defined that translate the modeled application into access control infrastructures. The platforms for which infrastructures are generated, are Enterprise JavaBeans, Enterprise Services for .Net as well as Java Servlets.

In [19] Kuhlmann et al. model the Mondex system with UML. Only static aspects of the application including method signatures are defined by using UML class diagrams. To specify the security properties that have to be valid the approach uses the object constraint language. The defined constraints are checked using the tool USE (UML-based Specification Environment). USE validates a model by testing it, i.e. it generates object diagrams as well as sequence diagrams of possible protocol runs. The approach neither considers the generation of code nor the use of formal methods to prove the security of the modeled application. The models are only validated by testing.

Deubler et al. present a method to develop security-critical service-based systems [7]. For modeling and verification the tool AutoFocus [5] is used. AutoFocus is similar to UML and facilitates the modeling of an application from different views. Moreover, the tool is linkable to the model checker SMV. The approach focuses on the specification of an application with

AutoFocus and, in a next step, the generation of SMV input files and formal verification using SMV. The generation of secure code is not part of the approach.

Chapter 7

Conclusion and Outlook

We presented a model-driven approach for the development of security-critical distributed applications. Our goal is to generate both, an implementation and a formal model, from the same platform-independent model to assure correctness and security of the code with respect to the formal model simply by construction. Security properties can be verified on the formal model, for which we already developed a suitable verification approach. This paper presented the transformation of platform-independent models into executable code by using additional platform-specific models. The current methodology focuses on JavaCard applications and can be applied to all security-critical smartcard applications. Our goal is to further extend the approach to be able to cope with more complex distributed systems, e.g using Web Services. Our long-term goal is to develop a model-driven approach where we are able to carry over the security properties to the application code in terms of formal refinement.

Bibliography

[1] M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. KIV 3.0 for Provably Correct Systems. In *Current Trends in Applied Formal Methods*, LNCS 1641. Boppard, Germany, Springer-Verlag, 1999.

[2] David A. Basin, Jürgen Doser, and Torsten Lodderstedt. Model Driven Security: From UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, pages 39–91, 2006.

[3] Bastian Best, Jan Jürjens, and Bashar Nuseibeh. Model-Based Security Engineering of Distributed Information Systems Using UMLsec. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 581–590. IEEE Computer Society, 2007.

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

[5] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFocus - Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik, Forschung und Entwicklung*, 14(3):121–134, 1999.

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

[7] Martin Deubler, Johannes Grünbauer, Jan Jürjens, and Guido Wimmel. Sound development of secure service-based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 115–124. ACM, 2004.

[8] Eclipse Modeling Project. http://www.eclipse.org/modeling/.

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

[10] H. Grandy, M. Bischof, G. Schellhorn, W. Reif, and K. Stenzel. Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In *FM 2008: 15th Int. Symposium on Formal Methods*. Springer LNCS 5014, 2008.

[11] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel. Developing Provably Secure M-Commerce Applications. In Günter Müller, editor, *Emerging Trends in Information and Communication Security (ETRICS)*, volume 3995 of *LNCS*, pages 115–129. Springer, 2006.

[12] H. Grandy, K. Stenzel, and W. Reif. A Refinement Method for Java Programs. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4468 of *LNCS*. Springer, 2007.

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

[14] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. *Formal Aspects of Computing*, 20(1), January 2008.

[15] MasterCard International Inc. Mondex. URL: `http://www.mondex.com`.

[16] C. Jones and J. Woodcock, editors. *Formal Aspects of Computing*, volume 20 (1). Springer, January 2008.

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

[18] Jan Jürjens. Security Analysis of Crypto-based Java Programs using Automated Theorem Provers. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 167–176. IEEE Computer Society, 2006.

[19] Mirco Kuhlmann and Martin Gogolla. Modeling and validating Mondex scenarios described in UML and OCL with USE. *Formal Aspects of Computing*, 20(1):79–100, January 2008.

[20] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language, 5th International Conference*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer, 2002.

[21] N. Moebius, D. Haneberg, G. Schellhorn, and W. Reif. A Modeling Framework for the Development of Provably Secure E-Commerce Applications. In *International Conference on Software Engineering Advances 2007*. IEEE Press, 2007.

[22] Lawrence C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.

[23] P. Y. A. Ryan, S. A. Schneider, M. H. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.

[24] G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses using ASMs. In *Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, LNCS. Springer, 2008.

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

[26] J. Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.