Modeling Security-Critical Applications with UML in the SecureMDD Approach

Nina Moebius, Wolfgang Reif, Kurt Stenzel
Department of Software Engineering and Programming Languages
University of Augsburg
86135 Augsburg, Germany
{moebius, reif, stenzel}@informatik.uni-augsburg.de

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

***Index Terms*—model-driven software engineering, UML, security, cryptographic protocols, verification**

I. Introduction

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

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

In the paper we only introduce the models showing the final view of the system which is used to generate code and the formal model. Of course, the creation of these models is a process that consists of several iterations and the UML diagrams evolve step-by-step. A disadvantage of UML is the lack of a comprehensive semantics directly usable in a verification system. This leads to difficulties for verification of models as well as for generation of code. This is solved by defining a mapping from the semi-formal to a formal presentation using abstract state machines (ASM) [2]. These have a well-defined and relatively simple semantics [3] [2]. Our formal specification is a combination of algebraic specifications and ASMs. Algebraic specifications are used for the description of the used data types as well as the attacker model. ASMs are used for the protocol dynamics. For verification we use the interactive theorem prover KIV [4].

Furthermore, we generate Java resp. Java Card code for smart card applications. Our group proposes a method to prove that an implementation is a refinement of the abstract formal model [5] by using the Java Calculus [6] [7] implemented in KIV.
The major advantage of our approach with respect to other existing techniques (e.g. [8]) is that we give a method seamlessly integrating modeling, formal methods as well as an implementation.

In this paper we describe the first part of the development process, i.e. the modeling of the application with UML. It is an extended and improved version of [9]. Our approach is focused on an easy to learn, and intuitive way of building the required models, and abstracts from details of the formal specification or the implementation. To model internal behavior, we extend activity diagrams with a UML-like language and use a syntax that is close to the one of an object oriented programming language. Our approach provides an opportunity to generate a formal model as well as runnable code without paying attention to the specifics of the formal specification and implementation which are harder to create and understand than the UML models.

Section II gives an overview of our SecureMDD approach. In Section III the SecureMDD UML profile and the used security data types are presented and a short introduction to the Model Extension Language (MEL) is given. Our modeling technique is illustrated by an electronic payment application called Mondex that is introduced in Section IV. In Section V we present the modeling of a security-critical application on the platform-independent level in detail and describe the platform-specific model in Section VI. Section VII gives some details about the MEL syntax and grammar. In Section VIII we shortly address some specifics regarding the generation of Java Card code, Section IX exemplifies some details of the transformation from UML into the formal model. Section X addresses related work and Section XI concludes.

II. The SecureMDD Approach

In this section we give an overview of our framework which aims to develop secure applications (see Fig. 1). The approach is based on model-driven software development (MDSD) methods. The developer creates a UML model of the system under development. Then, several model-to-model (M2M) and model-to-text (M2T) transformations are applied and finally, Java(Card) code as well as a formal model are generated.

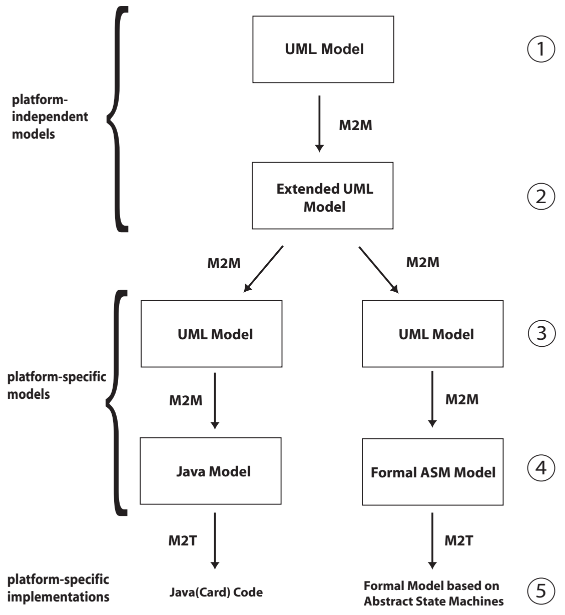


Fig. 1. Overview of the SecureMDD Approach

The approach starts with the modeling of a security-critical application with UML. We model the complete application, i.e. the static view, the structure of the system as well as the dynamic parts of an application. Since UML does not provide abilities to model the whole dynamic view, we extend the UML, especially UML activity diagrams, by a language called Model Extension Language (MEL). This language allows for modeling of e.g. assignments and creation of objects.

In the first step, the developer creates a UML model of the system under development (①). This model is platform-independent, i.e. it does not contain any specifics regarding the formal model or Java(Card) code. To model the flow of information and the processing of messages, activitiy diagrams extended with MEL expressions are used.

In a next step, the MEL expressions are parsed and stored in an abstract syntax tree. The 'Extended UML Model' is an instance of the UML metamodel which is extended by an abstract syntax tree of the MEL language (②). The generation is done automatically using model-to-model transformations.

Afterwards, as well with model-to-model transformations, different platform-specific models (PSM) are generated (③). On this level, the UML meta model is used. On the one hand, a model showing the smart card specific information is generated. This includes primitive types used in Java Card, Java Card expressions in activity diagrams as well as the translation of the stereotypes used in the previous model to Java classes. More details about the PSM can be found in Sect. VI. A smart card application always consists of one or more cards as well as a terminal with a card reader that communicates with the smart card. The terminal can be implemented using any programming language but Java is used in our approach. Since in this paper we concentrate on the modeling of the smart card part of an application, we omit the platform-specific model for generating the terminal code. On the other hand, we generate a platform-specific model containing details regarding the formal model which is based on algebraic specifications and abstract state machines (ASM). The expressions given as a MEL model are translated into syntactically correct ASM rules.

In a next step, a 'Java Model' resp. a 'Formal ASM Model' is generated from the platform-specific models. The Java model is an abstract syntax tree of Java whereas the ASM model is an abstract syntax tree of ASMs. Then, in a model-to-text (M2T) transformation, these models are transformed into Java Card code resp. a formal specification (⑤). The latter can be used to prove security properties of the modeled application using our interactive theorem prover KIV [10]. For hand-written formal models we already developed a method to prove security properties [11] [12].

The model-to-model transformations are implemented with the language QVT [13] and all model-to-text transformations with XPand [14].

III. The SecureMDD Profile and the Model Extension Language

In this section some security related data types and a UML profile which is tailored to cope with specifics regarding security-critical smart card applications are introduced. Furthermore, the Model Extension Language (MEL) that is used to extend UML activity diagrams is explained.

A. Predefined Security Datatypes

To model a security-critical application with UML it is expedient to define a few data types that are useful in these applications. Figure 2 shows the data types defined for the SecureMDD approach.

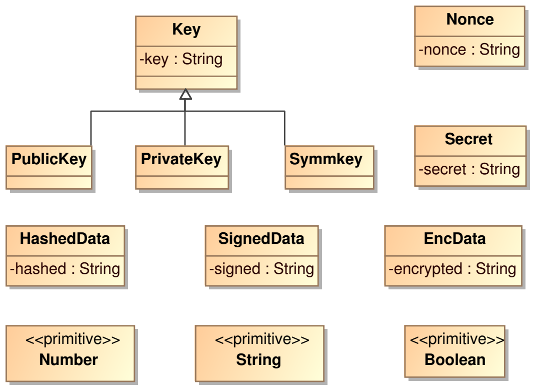


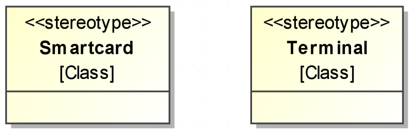
Fig. 2. Security Datatypes defined for the SecureMDD Approach

One important aspect is the use of keys. Thus, we define an abstract class `Key` that contains a cryptographic key. To capture the difference between symmetric and asymmetric encryption, i.e. public and private, keys, three subclasses of the `Key` class exist. Furthermore, a class `Nonce` representing nonces, i.e. random numbers used only once, is given. For example, nonces are used in cryptographic protocols to avoid replay attacks. Besides we define a type `Secret` which contains values that have to be kept secret, e.g. pin numbers or pass phrases. We explicitly distinguish secrets from primitive strings because this simplifies the formal verification of security properties. The classes `HashedData`, `SignedData` and `EncData` represent data that is hashed, digitally signed resp. encrypted. To facilitate the modeling on an abstract level without committing to an implementation language we additionally use primitive classes called `Number`, `String` and `Boolean` that represent numbers, strings as well as boolean values.

B. The SecureMDD Profile

Since UML is designed only to model standard application scenarios there is a need to extend it to specific application domains. For this reason the Object Management Group (OMG) [15] provides a mechanism to extend the scope of UML in a lightweight way by defining UML profiles. A profile extends the UML meta model and defines a set of stereotypes, tagged values and constraints.

In this section the SecureMDD UML profile is introduced.

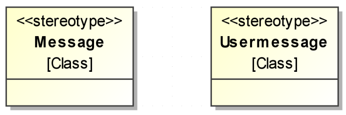


<<stereotype>> Smartcard [Class]
<<stereotype>> Terminal [Class]

Fig. 3. UML stereotypes defining the components smart card and terminal

Figure 3 illustrates the stereotypes defined for the components of a smart card application, i.e. one stereotype to annotate a class representing a smart card and one stereotype to label a class representing a smart card terminal with a card reader. These stereotypes are used in class diagrams to describe the static view of the application as well as in deployment diagrams to define the structure of the system. In deployment diagrams we use the meta model element *Node* to describe the components of the system. Since the *Node* element is derived from the meta model element *Class* it is sufficient to extend the meta class *Class* with the stereotype.

In the SecureMDD approach the message types exchanged during a protocol run are modeled as classes instead of operations. This is motivated by the fact that data in smart card applications is sent from resp. to the card in the form of sequences of bytes. Thus, the idea is to have a message as a (serialized) object instead of a remote method call. In Figure 4 the stereotypes annotating message classes are given.

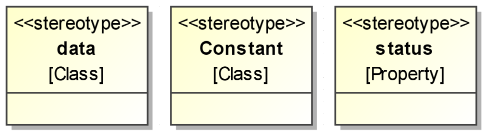


<<stereotype>> Message [Class]
<<stereotype>> Usermessage [Class]

Fig. 4. UML stereotypes annotating message classes

Here, we distinguish message objects exchanged between the card and the terminal and message objects sent from the user of the system to the system, for example by entering data using a GUI. The latter is explicitly modeled because for verification we need a formal model of the whole application, including the user inputs. Since the messages are defined in the class diagram, the stereotypes extend the meta class *Class*.

Figure 5 shows the stereotypes to label data classes and constants.

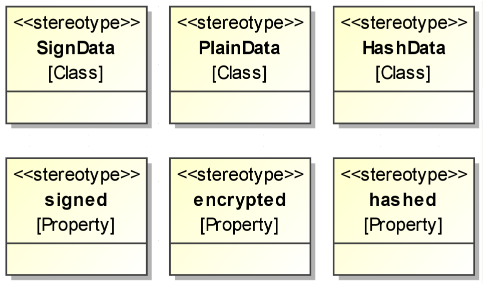


<<stereotype>> data [Class]
<<stereotype>> Constant [Class]
<<stereotype>> status [Property]

Fig. 5. UML stereotypes defining data, constants and status

These classes extend the meta class `Class`. Classes annotated with stereotype ≪data≫ are non-cryptographic data types. Classes not annotated with any stereotype are considered as ≪data≫ data type. To define constants used in the models the stereotype ≪Constant≫ is used. The stereotype ≪status≫ indicates the state of a component. While executing a protocol it is often essential to keep track of the step in the protocol that must be executed next. Depending on this step, the component may react differently by processing the next message or abort if the received message differs from the expected one. All possible states are modeled as an enumeration. An association between the component class, i.e. the terminal or the smart card, to the state class (annotated with stereotype ≪status≫) indicates the state of the component.

Figure 6 shows the stereotypes defined for digital signatures, encryption and hashing.



<<stereotype>> SignData [Class]
<<stereotype>> PlainData [Class]
<<stereotype>> HashData [Class]
<<stereotype>> signed [Property]
<<stereotype>> encrypted [Property]
<<stereotype>> hashed [Property]

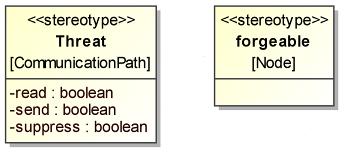
Fig. 6. UML stereotypes for encryption, hashing and signatures

If data modeled in the diagram is going to be signed, encrypted or hashed, it is annotated with stereotype ≪SignData≫, ≪PlainData≫ resp. ≪HashData≫. These stereotypes extend the meta class `Class`. Furthermore, we define stereotypes that denote the signing, encryption resp. hashing of data. If data is going to be encrypted during a protocol run, the data class is marked with stereotype ≪PlainData≫. If this data is encrypted and the result stored in a field of, e.g. the smart card or a message object, the corresponding association between this object and the PlainData object is annotated with stereotype ≪encrypted≫. In the class diagram we do not specify which key is used for encryption. Since this is a dynamic aspect, the concrete encrypt operation including the specification of the used key is specified in activity diagrams. Note that the generation of the formal model and Java Card code would also be possible if we omit the use of the stereotypes ≪SignData≫, ≪PlainData≫ and ≪HashData≫, i.e. all required information is already given when using the remaining stereotypes. However, we feel that it is good practice to use them because they increase the readability of the platform-independent models.

To verify certain security properties that have to hold for the modeled system it is necessary to describe a possible attacker resp. his abilities. An attacker may be able to interfere with the communication between smart card and terminal. This can be modeled appropriately with deployment diagrams. We use the communication path element to annotate the capabilities an attacker has to affect the communication. For this purpose we define the stereotype ≪Threat≫. The stereotype has three tags `read`, `send` and `suppress` that indicate if the attacker is able to read messages sent over that path, send or suppress messages. In some scenarios an attacker may try to forge a component, e.g. he may program his own smart card. If a fake component is conceivable it is annotated with stereotype ≪forgeable≫. The stereotypes defined to describe the attacker are shown in Figure 7.

C. The Model Extension Language (MEL)

The Model Extension Language (MEL) is used to extend activity diagrams. It is a simple language whose expressions are used in `Action` elements, `SendSignalActions`, `AcceptEventActions` as well as in `guard`s to model e.g. object creation, assignments, conditions, or the sending of a message. The aim is to have a language that can be used to model cryptographic protocols and at the same time is more abstract than a programming language. For example, MEL has a copy semantics and the developer does not have to take care about memory management and object creation which must be handled with care on smart cards. Since MEL is tailored to model the protocols of security-critical applications, it contains several keywords resp. predefined methods to express e.g. encryption, decryption, the generation of nonces and hash values. More details about MEL are given in Section VII.



<<stereotype>> Threat [CommunicationPath]
-read : boolean
-send : boolean
-suppress : boolean
<<stereotype>> forgeable [Node]

Fig. 7. UML stereotypes specifying the attacker capabilities

IV. Mondex

The SecureMDD approach is illustrated with the Mondex application which is introduced in this section.

Mondex cards are smart cards that are used as electronic purses with the aim of replacing coins by electronic cash. Mondex is owned by Mastercard International [1]. The main field of application is the secure transfer of money from one smart card to a second card. To perform a transfer both cards are inserted into a smart card terminal that also acts as user interface. The security properties that have to be verified for Mondex are that no money can be created and any value must be accounted for. In detail, this means that no money can be loaded onto a Mondex card without subtracting it from another card. Furthermore, if a transaction fails, no money should be lost. The Mondex case study recently received a lot of attention because its formal verification has been set up as a challenge for verification tools [16] that several groups [17] as well as our group [18] [19] worked on. For Mondex, several approaches dealing with formal methods and verification (model-checking, theorem proving and constraint solver) exist. But, they are not combined with an engineering discipline for system development. Rather, they use only formal techniques for specification and verification of the Mondex application. In the SecureMDD approach software engineering techniques and formal methods are integrated.

The Mondex application is another example that the design of security-critical systems is difficult. While verifying the security of the application our group has found a flaw in the original protocol [16]. Exploiting this flaw it is possible to cause a denial of service attack that fills the memory of the card. In this state the card is disabled unless the owner returns it to the bank. More details about the flaw can be found in [18]. The protocol given in this paper is a slight modification of the original protocol introduced in [20] and avoids the denial of service attack.

V. Modelling of security-critical smart card applications with UML

In this section our method to develop a security-critical application is introduced. All steps and artefacts are exemplified by the Mondex application. In subsection V-A the description of functional and security requirements using use cases is given. In subsection V-B our methodology to describe cryptographic protocols on a very abstract level is introduced. In subsection V-C the modeling of the static view using class diagrams is presented, subsection V-D describes the specification of the dynamic behavior using activity diagrams and the Model Extension Language. Subsection V-E introduces the modeling of the communication model as well as the attacker abilities using deployment diagrams.

A. *Use Cases describing functional and security requirements*

Use cases are used to capture functional requirements of the system in an informal way. As in a traditional software engineering process one or more use cases are written that describe the interaction between the system and external actors or systems. They describe the application in a way that can easily be understood. In our modeling method, use cases are the basis for the sequence and activity diagrams that are used to build the formal model as well as executable code. Below five of the use cases for Mondex are given. The first one, Person-to-Person Payment, is then used as running example in the following subsections.

Person-to-Person Payment

Basic Flow:

1) The customer of a shop wants to pay with his Mondex card.
2) He as well as the shop owner insert their cards into the corresponding card reader.
3) The shop owner enters the amount to pay.
4) The customer confirms the amount and starts the transfer of money.
5) The entered amount is transferred from the card of the customer to the card of the shop owner.
6) The system confirms the transfer by returning a receipt.
7) Both participants remove their cards from the reader.

Alternative Flows:

- 3) The entered amount is wrong: The shop owner cancels the process.
- 4) The customer does not agree with the entered amount: He cancels the transfer and the system aborts.
- 5) The balance of the customer card is lower than the amount to pay: The systems aborts and returns an error message.
- 5) The entered amount added to the current balance of the shop card exceeds the maximum value that can be loaded: The system aborts and returns an error message.
- 5) An error occurs while transferring the money or one of the participants removes his card too early: The system aborts and returns an error message. If the amount was already reduced on the customer card but has not been added to the card of the shop owner this is recorded on both cards. To recover the original balance of the customer card both cards have to be shown at the bank (see use case ”Recovery of Money”).

Security Requirements:

- No money is lost: If a transfer fails, either no money is charged from the customer card or if money was already charged it can be recovered correctly.
- An attacker is not able to program his own card such that he can use it as customer card and pay with it.
- It is not possible to load money onto a card without subtracting the same amount from a second card, i.e. no money can be created.
- It is not possible that a shop owner debits a higher amount than has been agreed.

Payment using Internet

Basic Flow:

1) A customer wants to pay with his Mondex card using an internet shop.
2) He inserts his card into his card reader (which is connected to his PC) and opens the web presentation of the shop.
3) He selects the products he wants to buy, enters his postal address for shipment and selects that he wants to pay now.
4) A connection to the remote card reader of the shop owner is established. The Mondex card of the shop owner is in this reader.
5) The amount to pay is transferred from the card of the customer to the card of the shop owner.
6) The system confirms the transfer.
7) The customer removes his card from the reader.
8) The shop owner sends the goods to the customer.

Alternative Flows:

- 3) The balance of the customer card is lower than the amount to pay: The systems aborts and returns an error message.
- 4) The entered amount added to the current balance of the shop card exceeds the maximum value that can be loaded: The system aborts and returns an error message.
- 5) An error occurs while transferring the money or one of the participants removes his card too early: The system aborts and returns an error message. If the amount was already reduced on the customer card but has not been added to the card of the shop owner this is recorded on both cards. To recover the original balance of the customer card both cards have to be shown at the bank (see use case ”Recovery of Money”).

Security Requirements:

- see use case ”Person-to-Person Payment”

Recovery of Money

Basic Flow:

1) If a transaction fails (i.e. money was charged from the customer card but has not been added to the shop card) both participants of the transfer go to the bank.
2) Showing their Mondex cards it can be discovered if and what amount of money was reduced from the customer card.
3) The system adds the corresponding amount to the customer card.

Alternative Flows:

- 3) If the amount added to the current balance of the customer card exceeds the maximum balance of the card the amount will be paid out in cash.

Security Requirements:

- If money was lost it can be recovered only once, i.e. showing the cards again it is not possible to force a recovery again.
- It can be detected if the transfer has been aborted after the amount was added to the shop card. In this case no money is recovered.

Recharge of money at an automatic teller machine (ATM)
Basic Flow:

1) The card owner goes to the ATM (within his bank) and inserts his Mondex card.
2) The card owner specifies the details of his bank account.
3) He authorizes by entering his PIN number.
4) The system checks that the PIN is correct.
5) The card owner enters the amount he wants to recharge.
6) The entered amount is debited from the bank account of the card owner and loaded onto the card.
7) The card owner removes his card from the terminal.

Alternative Flows:

- 4) The entered PIN is not correct: The system returns an error message and asks for retry. After three times entering a wrong PIN the card is locked.
- 5) The balance of the owners bank account is less than the entered amount: The system returns an error message and requests to enter a lower amount.
- 5) The entered amount added to the current balance of the card exceeds the maximum value that can be loaded: The system returns an error message and requests to enter a lower amount.

Security Requirements:

- The amount loaded onto the card equals the one charged from the bank account. It is not possible to load money onto a card without reducing the bank account by the correct amount.

Discharge at an ATM
Basic Flow:

1) The card owner inserts his card into an ATM at the bank.
2) He selects that he wants to have repaid the money.
3) The ATM pays out the amount currently stored onto the card and sets the current balance of the card to zero.
4) The customer removes the card from the reader.

Alternative Flows:

- 3) The customer removes his card from the reader too early: No money is paid out.

Security Requirements:

- The amount paid out in cash equals the balance of the card.
- If returning the cash to the card owner the balance of the card is set to zero.

Other use cases cover the viewing of the last transactions, storing money of different currencies on the same card or payments using mobile phones. Also the recharge of money using the internet or the use of money in cash instead of a bank account for recharge is possible. Since the entire application is too large to present here we only model the transfer of money between a shop owner card and a customer card (Use Case Person-to-Person Payment).

B. *The Protocol Description*

Our goal is to give an intuitive way to model security protocols. A reader of the model should be able to understand the protocol without getting lost in details. We use sequence diagrams to specify the protocol steps and the flow of information. The idea is to start with a very abstract view of the possible protocols and refine these sequence diagrams step by step. The diagram shown in Fig. 8 shows the final sequence diagram for ”Person-to-Person Payment”. At this point the protocol which is later implemented is already elaborated. This diagram is used as basis to develop the complete dynamic behavior of the system using activity diagrams. Note that we do not show the diagrams that were drawn while working out the final models.

The sequence diagram contains one lifeline for each component participating in the protocol and additionally one lifeline for the ”user”. The user represents the customer of the service and usually initiates a protocol, i.e. ’sends’ the first message. For Mondex, we distinguish the card of the shop owner (in the following called *to* purse) and the card of the customer (in the following called *from* purse). Since a Mondex card can act as *to* card as well as *from* card this distinction is only to achieve a better readability of the diagrams.

The protocol used for payments between two persons (see Fig. 8) works as follows:

The user, i.e. the shop owner, initiates the protocol run by sending the value to be transferred to the terminal (`UTransferMoney`). Afterwards the terminal queries the *to* purse to provide its data, e.g. its name (= unique number), by sending the instruction `getData`. The *to* purse returns this data (message `ResGetData`). In a next step the terminal sends a message called `StartFrom` to the *from* purse which initiates the transfer on the `from` purse. This message contains all information required to start the transfer, i.e. the value to be transferred as well as the unique data of the other purse. Then, the *from* purse sends a `StartTo` message to

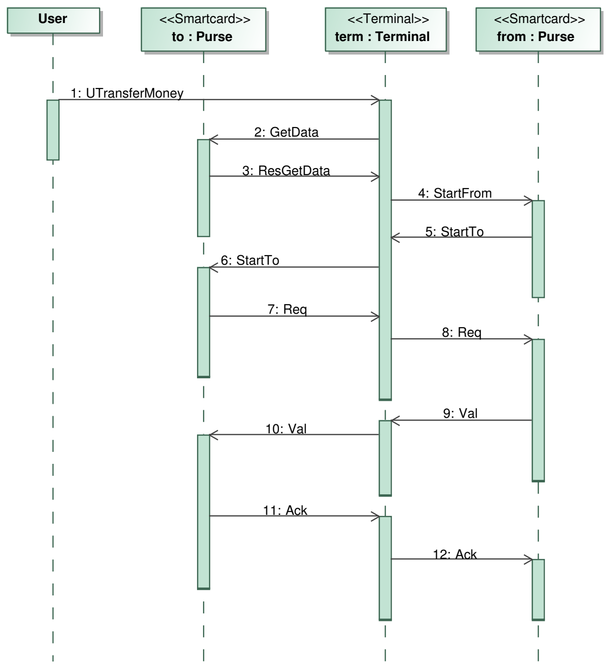


Fig. 8. Protocol Description for Person-to-Person Payment

the terminal which forwards it to the *to* purse. This message contains all data required to run a transfer and, after receiving it, the *to* purse initiates the transfer. Note that from now on the terminal only forwards message that it receives, i.e. if receiving a message from the *from* purse, it forwards it to the *to* purse without modifying the message or its state. In a next step, after checking that the received transfer information is correct, the *to* purse generates a `Req`(uest) message to request a transfer, i.e. requests the decrease of the balance of the *from* purse. After receiving this message the *from* purse decreases its balance and sends back a `Val`(ue) message which states that its balance has been decreased. Then, the *to* purse increases its balance and sends back an `Ack`(nowledgement) message that confirms the transfer.

C. Static View of the System

In the following the modeling of the static view of a smart card application is introduced. To model specifics regarding the domain of security-critical applications we use the UML profile as well as the security data types defined in Section III. The method is exemplified by the Mondex application but is applicable for smart card applications in general.

Fig. 9 illustrates the class diagram of the Mondex application. Note that the diagram only shows the part of the static view which is needed for Person-to-Person payments, other parts e.g. regarding the recovery or recharge of money are omitted.

Every component of the system, i.e. smart card and terminal, are represented by a class annotated with stereotype ≪Smartcard≫ resp. ≪Terminal≫. This distinction is necessary because the generated code (e.g. Java Card vs. Java) and the formal model differ depending on the type of component. In the Mondex application we have the class `Purse` which is representing the smart card as well as the `Terminal.`

The message types are modeled as classes. Here, we use an abstract class annotated with stereotype ≪Message≫ from which all concrete message classes are derived. In Fig. 9 several concrete message classes, e.g. `Req`, `Val` and `Ack`, are defined. Note that these messages are derived from the messages modeled in the corresponding sequence diagram (see Fig. 8).

All data types are modeled as classes and annotated with corresponding stereotypes, i.e. ≪data≫ for non-cryptographic data types and ≪PlainData≫, ≪HashData≫ and ≪SignData≫ for data that is going to be encrypted, hashed or signed. In the Mondex model we have defined the data class `PurseData` that consists of the unique name of the purse as well as a sequence number that increases after every protocol run and ensures the uniqueness of every `PayDetails`. A `PayDetails` object records the details of the current transaction, i.e. the participating purses as well as the amount to transfer. Furthermore, we define one class called `Msgcontent` that is going to be encrypted and thus annotated with stereotype ≪PlainData≫. This class contains the pay details of the current transaction and a message flag denoting if the (encrypted) data belongs to a `Req`, `Val` or `Ack` message. If this flag is omitted, the following atack is possible.

An attacker captures and suppresses a `Req` message and uses the contained encrypted data to send a correct `Val` message to the sender. Receiving this message, the sender of the `Req` message, i.e. the *to* purse, assumes that the *from* purse has decreased its balance correctly and increases its balance. Then, the balance of the *to* purse has been increased without decreasing the balance of the *from* purse.

Since an object of type `Msgcontent` is encrypted and afterwards sent with a `Req`, `Val` or `Ack` message, the corresponding associations are annotated with stereotype ≪encrypted≫. To denote the types of used attributes we use the self defined primitive types `Number`, `Boolean` as well as `String` and the security data types described in III-A. To cover associations with multiplicity greater than one we use a predefined list. For example, the `Purse` class has an exception log for failed transactions. This is modeled by an association with multiplicity 0..LOGLENGTH. This exception log is translated to a list that can be accessed with predefined methods e.g. to add an object to the list. These predefined operations are later used in the activity diagrams.

The possible states a component may be in are defined as an enumeration. An association from a component to this enumeration, annotated with stereotype ≪status≫ defines the states of a component. A purse may be in state `IDLE`, `EPR` (expecting request), `EPV` (expecting value) or `EPA` (expecting acknowledge). Since the terminal simply forwards messages to the cards and accepts all kinds of messages, it needs no state.

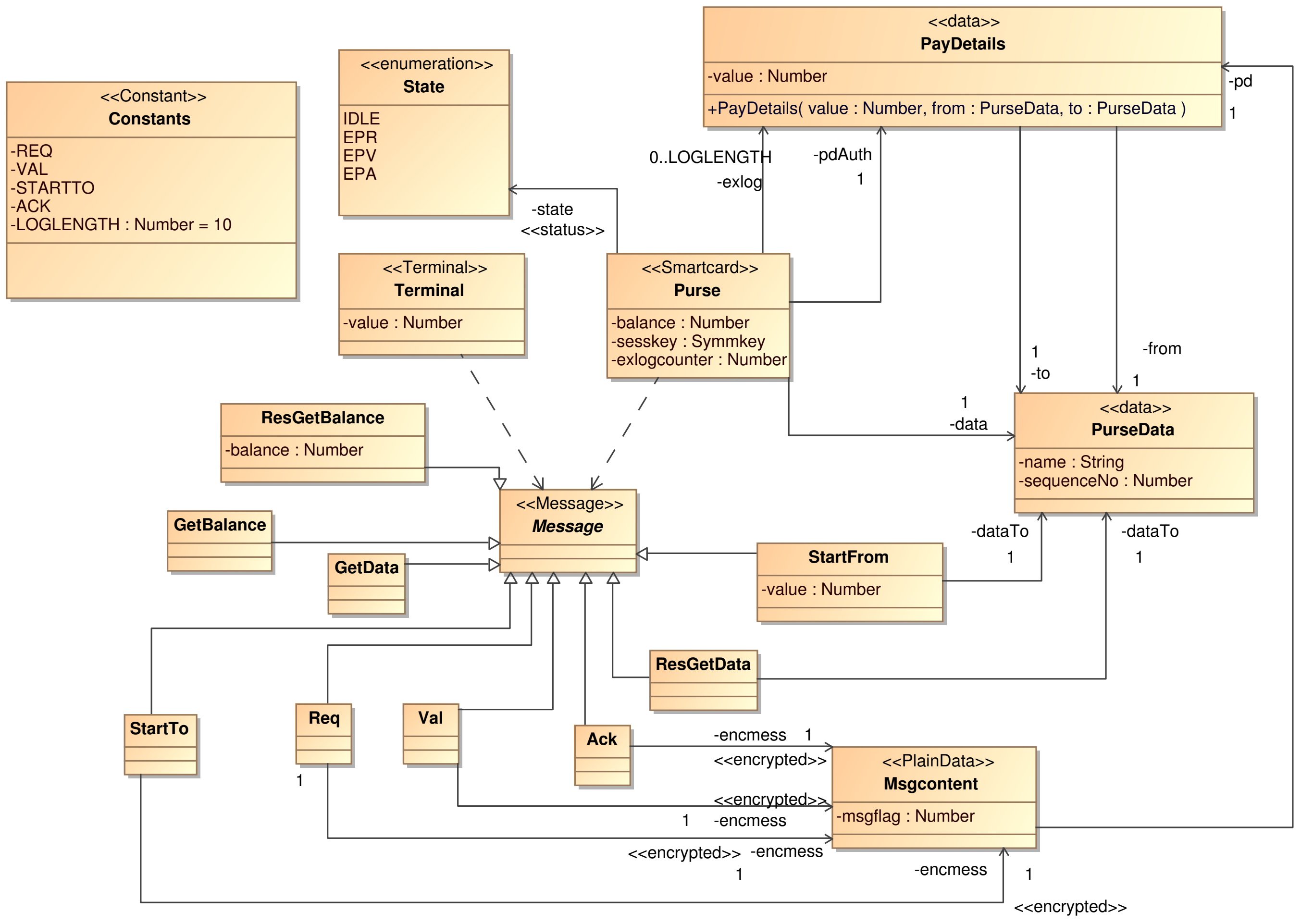


Fig. 9. Static View of the Mondex application

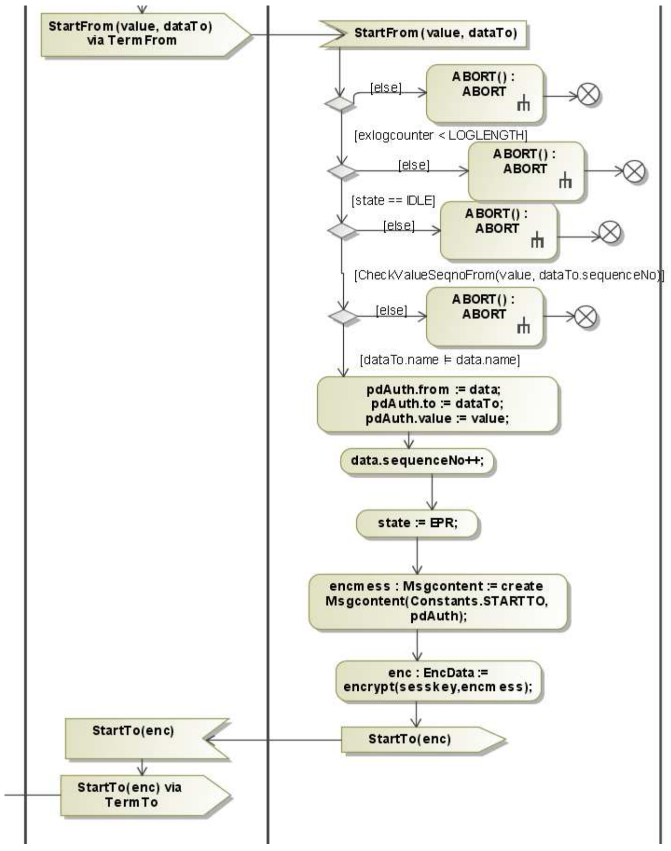
D. Dynamic Behavior

Sequence diagrams describe the sequence of messages that is exchanged between components but do not capture internal actions or the behavior in case an error occurs. For this reason we additionally use activity diagrams that extend the sequence diagrams and describe changes in the internal state of the components after processing a message. The activity diagram describes the communication as well as the sequence of actions taken as a result of receiving a message. At this point we use our Model Extension Language (MEL) which was shortly introduced in Section III. MEL allows to describe e.g. creation of objects, assignments or guards of conditions. We use activity diagrams instead of UML state diagrams because they turned out to be hard to read and confusing for applications we focus on (with many condition checks).

For each use case we define one activity diagram. For a better readability we additionally allow the definition of sub activities that are called within an activity. In Fig. 10 one part of the activity defining the protocol executed for Person-to-Person payments is given. The whole activity diagram can be found in the appendix.

For each component participating in the protocol one swim lane exists in the diagram. As in the sequence diagram we have a swim lane for the user, the *to* as well as *from* purse and for the terminal. A protocol can be divided into segments where one segment consists of one protocol step. A protocol step has the following parts: A component receives a message, performs several tests to check whether the message is correct and can be handled and processes the data. Finally, the component may send a message to another component. We use `SendSignalActions` to denote the sending of a message, `AcceptEventActions` to indicate the receiving of a message as well as `Action` elements to denote MEL expressions like object creation, assignments and calls of predefined operations.

The segment in Fig. 10 shows the swim lane of the terminal on the left as well as the one of the *from* purse. The terminal sends a `StartFrom` message to the *from* purse. This message contains the `value` to be transferred as well as the data of the *to* purse. The *from* purse receives this message. The content of



StartFrom(value, dataTo) via TermFrom
StartFrom(value, dataTo)
[else]
ABORT() : ABORT
[exlogcounter < LOGLENGTH]
[else]
ABORT() : ABORT
[state == IDLE]
[else]
ABORT() : ABORT
[CheckValueSeqnoFrom(value, dataTo.sequenceNo)]
[else]
ABORT() : ABORT
[dataTo.name != data.name]
pdAuth.from := data; pdAuth.to := dataTo; pdAuth.value := value;
data.sequenceNo++;
state := EPR;
encmess : Msgcontent := create Msgcontent(Constants.STARTTO, pdAuth);
enc : EncData := encrypt(sesskey,encmess);
StartTo(enc)
StartTo(enc)
StartTo(enc) via TermTo

Fig. 10. Mondex Activity Diagram showing the sending, receiving and processing of a `StartFrom` message. On the left side one can see the swim lane of the terminal, on the right side the one of the *from* purse

it, i.e. the value and data, are handled as local variables. Then, the purse checks if the counter which counts the exception log entries is less than the possible maximum length. If not, the protocol aborts. The abort step is defined in a separate activity diagram and is called from this protocol (defined by a rake element). A sub activity has access to the properties of a component but not to the local variables. If the condition is satisfied it is tested whether the `state` of the purse is set to `IDLE`. Next, it is checked if the received value and sequence number of the *to* purse fulfill certain conditions, for example that the value to be transferred is greater than zero. These checks are also defined in a separate activity `CheckValueSeqnoFrom` which has two parameters and returns a boolean value with the result of the tests. Since a sub activity has no acces to the local variables, these have to be passed as arguments. The sub activity can be found in the appendix. If one of the checks fails the `ABORT` sub activity is called. Otherwise, the purse modifies some fields, e.g. the field `pdAuth` is filled with the current pay details, the purse's `sequenceNo` is increased and the `state` is updated to `EPR`. Our Model Extension Language has a copy semantics but updates of fields modify the fields of the original object. In a next step, a local variable `encmess` of type `Msgcontent` is created and, in the next action, encrypted with the symmetric key stored in field `sesskey`. The `encrypt` method is predefined in MEL and used for symmetric and asymmetric encryption. The result of the encryption of data is an object of type `EncData` that consists of a string containing the encrypted data (see Section III). This `EncData` object is stored in a local variable `enc`. Afterwards a `StartTo` message containing the created `enc` object is sent to the terminal. The terminal receives this message and forwards it to the *to* purse. The keyword `via` denotes to which components the message is sent resp. denotes the used port (see subsection V-E for more details). If the communication path is unique, e.g. the purse only communicates with the terminal, the `via` keyword can be omitted.

Activity diagrams are used to define the communication between the different components as well as the processing of a message, i.e. they are used to model cryptographic protocols. In applications with large protocols it may be desirable to add some code by hand after generating the modeled parts of the system instead of creating activity diagrams for the whole application. For this reason the developer can add own method calls where the corresponding method bodies are added later by hand on code level. Note that this causes problems resp. inconsistencies when verifying the security of the system using a formal model automatically generated from the UML models. To ensure that the security properties which are proved on the formal model also hold on code level, the formal model has to be a suitable representation of the code. This means that all changes and additions which are made on the code (by hand) have to be done on the formal model as well.

E. Attacker and Communication Model

To verify cryptographic protocols it is necessary to formally specify the communication infrastructure as well as an attacker model. Almost all formal approaches (e.g. [21] [22]) for verifying cryptographic protocols use a rather simple model of communication and the Dolev-Yao [23] threat model. There, no constraints regarding the communication structure are given and it is assumed that the attacker may access all communication links, i.e. he can read all messages sent over that link, suppress them or write messages on that channel. In these approaches (mainly addressing internet protocols) it is ignored that certain components cannot communicate directly with other components for physical reasons.

Also, the possibility that some connections are secure against eavesdropping and others are not, is abstracted away. In contrast, our formal model is not limited to Dolev-Yao attackers. The main reason for an attacker model with reduced (but more realistic) abilities is that it becomes possible to have simpler protocols still preserving the desired security properties.

In our approach we explicitly model the existing connections. For each connection we denote if the attacker is able to read or suppress messages and whether he can send messages over that channel. But these annotations do not suffice to describe all possibilities an attacker might have. For example, an attacker could program his own forged smart card. If the protocol has a flaw such that the forged card takes advantage of the weakness of the protocol it may be possible that the attacker gains some information e.g. about secret keys.

UML provides the use of deployment diagrams to define the physical structure of a system. In our approach we use them to describe the communication structure as well as the attacker model of our application. Fig. 11 shows the deployment diagram for the Mondex application.

The components participating in the application are modeled as nodes. The terminal has one connection to the *to* purse, one to the *from* purse as well as one to the user. If a component sends a message it has to be determined which connection is used for sending. To be able to reference the connections the connection ends, also called ports, are named. If multiple connections exist between two components, the connection that is used for sending is addressed using the `via` keyword in the activity diagram.

For Mondex we assume that an attacker may have full access to the connections between terminal and cards. Thus, these connections are marked with `read`, `send` and `suppress`. Furthermore, an attacker may program his own smart card and use it as a Mondex card to attack the system.

VI. Platform-specific Smartcard Model

Based on the platform-independent model of the application a platform-specific model (PSM) is generated for each

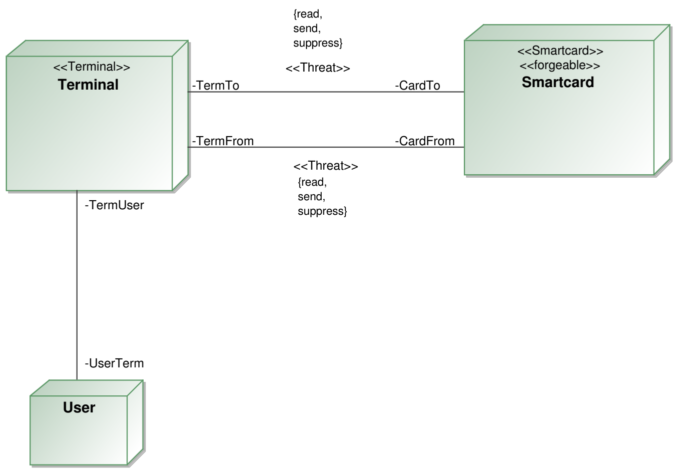


Fig. 11. Deployment Model for Mondex

platform. For the Mondex application, we distinguish three platforms: one for the terminal, one for the smart card as well as one for the formal model. In this section we present the static view of the platform-specific model for the smart card in more detail.

Figure 12 shows the platform-specific class diagram of the Mondex application.

In the class diagram the abstract data types for the smart card are replaced by Java Card [24] specific data types. Note that Java Card does not support integers or strings. Thus, all fields of type `Number` are translated to shorts, `String`s are translated into byte arrays and `Boolean` are replaced by the Java type boolean. Furthermore, for each class a constructor is added.

One main aspect of the PSM is the removal of stereotypes dealing with cryptography which were used in the platform-independent model. Instead, some classes and interfaces are added. The resulting class diagram is close to the structure of the Java Card code but omits some technical details. Remember that in the platform-independent model the encryption of data was modeled by adding a stereotype named ≪encrypted≫ to the corresponding association. The referenced class is then annotated with a stereotype ≪PlainData≫ which denotes that this data type can be encrypted. In the platform-specific model we add an interface called `PlainData` which is implemented by all classes that were marked as ≪PlainData≫ in the PIM. Moreover, we add the data type `EncData` that represents encrypted data. This class has a field `encrypted` of type byte array which stores the encrypted data. Since the Java Card Crypto API operates on byte arrays, it is of type byte[]. The class has two static methods, `encrypt` and `decrypt`, which correspond to the predefined methods of the same name defined in MEL. The `encrypt` method takes an object of type `Key` and a `PlainData` object and returns an `EncData`. The `decrypt` method operates on a `Key` and an `EncData` object and returns the decrypted `PlainData`. The classes `StartTo`, `Req`, `Val`

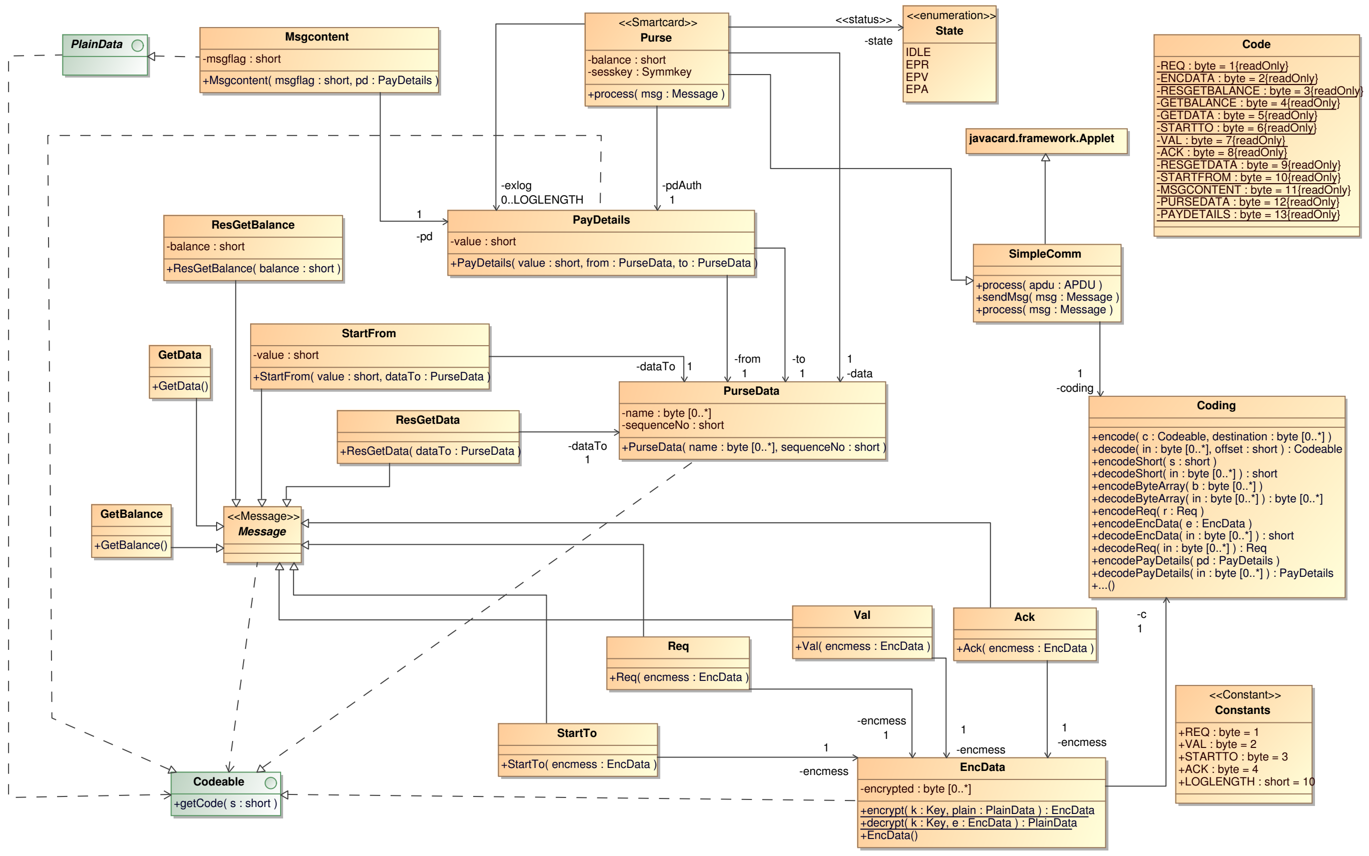

PlainData
Msgcontent
-msgflag : short
+Msgcontent(msgflag : short, pd : PayDetails)
<<Smartcard>>
Purse
-balance : short
-sesskey : Symmkey
+process(msg : Message)
<<status>>
-state
<<enumeration>>
State
IDLE
EPR
EPV
EPA
Code
-REQ : byte = 1{readOnly}
-ENCDATA : byte = 2{readOnly}
-RESGETBALANCE : byte = 3{readOnly}
-GETBALANCE : byte = 4{readOnly}
-GETDATA : byte = 5{readOnly}
-STARTTO : byte = 6{readOnly}
-VAL : byte = 7{readOnly}
-ACK : byte = 8{readOnly}
-RESGETDATA : byte = 9{readOnly}
-STARTFROM : byte = 10{readOnly}
-MSGCONTENT : byte = 11{readOnly}
-PURSEDATA : byte = 12{readOnly}
-PAYDETAILS : byte = 13{readOnly}
javacard.framework.Applet
-exlog
0..LOGLENGTH
-pdAuth
1
-pd
PayDetails
-value : short
+PayDetails(value : short, from : PurseData, to : PurseData)
ResGetBalance
-balance : short
+ResGetBalance(balance : short)
SimpleComm
+process(apdu : APDU)
+sendMsg(msg : Message)
+process(msg : Message)
StartFrom
-value : short
+StartFrom(value : short, dataTo : PurseData)
GetData
+GetData()
-dataTo
-from
-to
-data
-coding
PurseData
-name : byte [0..*]
-sequenceNo : short
+PurseData(name : byte [0..*], sequenceNo : short)
ResGetData
+ResGetData(dataTo : PurseData)
Coding
+encode(c : Codeable, destination : byte [0..*])
+decode(in : byte [0..*], offset : short) : Codeable
+encodeShort(s : short)
+decodeShort(in : byte [0..*]) : short
+encodeByteArray(b : byte [0..*])
+decodeByteArray(in : byte [0..*]) : byte [0..*]
+encodeReq(r : Req)
+encodeEncData(e : EncData)
+decodeEncData(in : byte [0..*]) : short
+decodeReq(in : byte [0..*]) : Req
+encodePayDetails(pd : PayDetails)
+decodePayDetails(in : byte [0..*]) : PayDetails
+...()
GetBalance
+GetBalance()
<<Message>>
Message
Val
+Val(encmess : EncData)
Ack
+Ack(encmess : EncData)
Req
+Req(encmess : EncData)
-c
<<Constant>>
Constants
+REQ : byte = 1
+VAL : byte = 2
+STARTTO : byte = 3
+ACK : byte = 4
+LOGLENGTH : short = 10
StartTo
+StartTo(encmess : EncData)
-encmess
Codeable
+getCode(s : short)
EncData
-encrypted : byte [0..*]
+encrypt(k : Key, plain : PlainData) : EncData
+decrypt(k : Key, e : EncData) : PlainData
+EncData()

Fig. 12. Smart card-specific class diagram of the Mondex application

and `Ack` were defined in the PIM with associations to the class `Msgcontent` (annotated with ≪PlainData≫). Now, these classes have associations to the class `EncData` and hence reflect the implementation with Java Card.

To communicate with the terminal we add a class `SimpleComm` which defines two methods to receive and process a message as well as a method for sending a message. This class extends the class `Applet` defined in the Java Card API. The class `Purse` that represents the smart card extends the class `SimpleComm`.

Since the communication between card and terminal is based on byte arrays we additionally need a serialization mechanism that serializes the objects that are sent to the terminal. This is realised by a class named `Coding` that defines methods for serialization and deserialization of each object which is sent during a protocol run. More details about the implementation in Java Card as well as the generation of code can be found in [25].

The activity diagrams of the platform-specific model still have the same structure but the MEL expressions are parsed and replaced by Java Card expressions.

It is easy to see that our platform-independent model is an abstracted view of a security-critical smart card application that can be created without knowing technical details about programming with Java Cards. It is possible to model an application using the predefined stereotypes without thinking about a possible implementation. Then, in a next step, these abstract models are translated into more Java Card specific models automatically.

VII. The Model Extension Language

In this section the MEL language is presented in detail. The syntax of MEL is shown in Fig. 13. It is based on Java, but a little bit more UML-like.

The description of the grammar can be read from top to bottom. MEL can be used in UML `Actions`, in UML `guards`, and in UML `SendSignalActions` and `AcceptEventActions` which are treated differently. A (normal) action can contain either one expression, or a list of statements. A statement in MEL is simply an expression followed by a semicolon. Java statements like conditional, loop, return etc. are not supported, but must be modeled with activity diagram elements. MEL expressions and types are a subset of Java expressions and types. The most obvious omissions are arrays and generic types. The idea is to use more abstract data types like lists or sets instead of arrays. Generic types may be added for non-Java Card applications in the future. MEL contains an **else** expression that may only be used on top-level in a guard (UML also defines **else** as a special guard). A local variable declaration (locvardecl in

```
Start = Action | Guard
      | SendSignalAction | AcceptEventAction

Action = Expr | Stm*
Guard = Expr
SendSignalAction = Expr
AcceptEventAction = Expr

Stm = Expr ;
Expr = Locvardecl | Assignment | CreateExpr | MethodCall
      | BinaryExpr | UnaryExpr | LiteralExpr | FieldAccess
      | Name | ( Expr ) | else
ExprList = ε | Expr[, Expr]*

Locvardecl = Identifier : Type | Identifier : Type := Expr
Assignment = Expr := Expr
CreateExpr = create Identifier ( ExprList )
MethodCall = Identifier ( ExprList )
             | Expr . Identifier ( ExprList )
BinaryExpr = Expr Binop Expr
UnaryExpr = Unop Expr | Expr Unop
LiteralExpr = true | false | NumberLiteral | StringLiteral
FieldAccess = Expr . Identifier
Name = Identifier | Name . Identifier

Identifier = Legal Java identifier (JLS 3.8)
Type = Name
Binop = == | != | < | > | <= | >= | + | -- | * | / | %
       | and | or | via
Unop = + | -- | ++ | -- | not | #
NumberLiteral = Legal Java integer literal (JLS 3.10.1)
StringLiteral = Legal Java string literal (JLS 3.10.5)
```

Fig. 13. The MEL language used in activity diagrams

Fig. 13, technically not an expression in Java) has a UML-like syntax, similarly an assignment (**:=** instead of simply =). Logical operations must be written as **and**, **or**, **not** instead of **&&**, ||, **!**. A new prefix operation is **#** that denotes the length of a list or the size of a set. Another new operation is **via** that may only be used on top-level in send and accept actions and specifies the communication paths over which a message is sent or received.

After parsing a MEL expression an annotated abstract syntax tree in the form of a model is created in the same manner as by a Java compiler. Annotating MEL requires a context (the classes of the class diagram), and a current class (the swim lane of the activity diagram), and must be done in sequential order following the control flow of the activity diagram to capture the scope of local variables. Identifier are classified as either local variables, fields, classes etc. and for every method call a suitable method declaration must exist (either in the class diagram, or in the predefined types, or in a sub activity diagram), and so on.

An `AcceptEventAction` must be used as an entry point into a swim lane. It must contain a method call of the form *Classname(id1,id2,...)*, optionally followed by a **via** *Identifier*. The *Classname* must name a message class, and the identifier *id1*, *id2*, ... are interpreted as local variables with the types of the attributes and associations of *Classname*. For example, `StartFrom(val,pd)` means that a `StartFrom` message is received. `val` becomes a local variable of type `Number` that is initialized with the `value` attribute, and `pd` becomes a local variable of type `PurseData` containing `value.dataTo` (see the class diagram in Fig. 9). The scope of a local variable ends at the border of a swim lane.

MEL has a do-what-I-mean flavor that is very convenient for modeling. This can be considered as syntactical sugar. For example, the static members of a class can be accessed without a classname: The name resolution will interpret `state == IDLE` (see Fig. 10) as `state == State.IDLE`. Furthermore, MEL ignores object identities. In a communication scenario with cryptographic protocols objects are almost never identical, because messages treat objects as data. Therefore, == can be used to compare objects, and is interpreted as an equals test that compares attributes.

The annotated abstract syntax tree is essential for error checking as well as for the correct generation of code (e.g. == may become an `equals` method call). The idea is to make the MEL language easy to use for a modeler, but still as precise as a programming language. In the future, MEL can be extended if it is useful, for example with OCL-like constructs for collections. However, control flow should be modeled with activity edges.

VIII. Generation of Code

Smart cards are small, secure computers with a size of 1 $\times$ 1 centimeters and a thickness of less than 1 millimeter. For example, the subscriber identity module (SIM) of mobile phones is a smart card, the new electronic passports contain a contactless smart card, and smart cards are used as payment cards, health cards, for access control. Java Card [26], [27] is a version of Java [28] tailored to smart cards. More than 3.5 billion Java smart cards have been issued up to now [29].

Java Card has the same syntax and semantics as Java, but the programming style is usually very different from ‘normal’ Java programs. The reason for this are the severe resource restrictions (memory and speed) of smart cards. Java Card has no Strings, no floating point arithmetic, and no Integers. Furthermore, threads and garbage collection are not supported. The missing garbage collection means that the programmer must be very careful when he creates objects or arrays because the allocated memory will never be freed.

The communication with a smart card is realized by using APDUs [30] (application protocol data units), essentially sequences of bytes in a predefined format. The Java Card API for the communication works with byte arrays. The missing garbage collection and the communication API induce a programming style that is usually not object-oriented. Typically, Java syntax is used to manipulate byte arrays directly omitting object-oriented paradigms like modularization and encapsulation. Examples can be found in [31] that contains two different Mondex implementations based on byte arrays.

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 transforming the object-oriented view of the application into a program consisting of byte array representations for each object resp. class. Thus, the purse class implementing the protocol steps of the cryptographic protocol operates on the data types defined in the platform-independent model by the developer.

However, the communication is still based on byte arrays. This means, to transmit data between a smart card and a terminal the message objects as well as associated objects must be converted into byte arrays and back again. The easiest way to do so is to serialize each message object before sending it and after receiving a byte array message to convert it into the corresponding message object. This is done using an encoding similar to a TLV encoding [32], [33]. This encoding is highly application dependent because Java Card does not support reflection. Therefore it is ideally suited for automatic code generation.

Another challenge is the missing garbage collection. The required objects cannot be created during the protocol runs but must be allocated once beforehand and reused. In our approach we generate code for an object store that allocates the required objects and manages them, i.e. if an object is needed it is requested from the store. More details on the code generation can be found in [25].

IX. Generation of a Formal Model for Verification

To prove the security of the system under development we automatically generate a formal model based on algebraic specifications and abstract state machines suitable for our interactive theorem prover KIV. The static aspects of the modeled application are defined by algebraic specifications whereas the dynamic part of the system is translated into an abstract state machine (ASM) [2]. The formal model uses the application-dependent data types which are defined in the class diagram, i.e. specifications exist for the messages, plain data and so on. We use application-dependent types instead of a generic type as used in [34] [12]. Since the formal model is used for interactive verification, it is very helpful to have a formal model that is close to the UML models.

To model the attacker we define the attacker knowledge which contains all (relevant) data known by the attacker during a protocol run, similar to [34] and [35]. The attacker knowledge contains all data that is part of a message and can be analyzed by the attacker. In the Mondex example this includes the encrypted content of the `Req`, `Val` and `Ack` messages. If the attacker does not know the key he cannot decrypt the content, but with an insecure protocol he may later learn the key, and then decrypt the data. All non-security critical data such as the amount to load is not explicitly stored in the attacker knowledge because this data is not secret and assumed to be known by the attacker.

In the formal model the components of the systems are defined as different *agents* that communicate by exchanging messages. The formal model captures the behavior of the real world that is related to the application. In the real world, many Mondex cards exist. To model the transfer of money, at least two cards (agents in the formal model) are needed. Indeed, it may be possible that there exists an attack on the protocol that needs three or more cards, and does not work with only two cards. In this case a formal model with only two cards would be grossly flawed, because the proofs of the security properties would succeed for a protocol that is in reality insecure. Therefore the formal model has an arbitrary, but finite, number of cards (more precisely: instances for each agent type). To represent the communication we explicitly model the possible connections between two agents. Since more than one communication path between two agents may exist, we additionally use ports to distinguish the paths. The information about communication paths and ports is taken from the deployment diagram. To model the sending and receiving of messages in the formal model we use inboxes (essentially queues) for each component and port. An inbox is of type message list and contains all messages that were received by an agent but not yet processed.

The dynamic part of the system is modeled as an abstract state machine (ASM). The state of the ASM consists of the states of all agents. In the Mondex example the state of the purse consists of the values of the attributes and associations of the `Purse` class. A step of the ASM applies one ASM rule and transforms the state. A run of the ASM is a sequence of single steps and creates a trace, i.e. a sequence of its states. A trace models arbitrary protocol runs that could happen in the real world. Since many different events occur in the real world (e.g. the attacker may choose to interfere with a communication or not) an adequate formal model is the set of all possible traces. If the protocol is secure for all possible traces we assume that the protocol is secure in the real world. Therefore the ASM must allow the same choices that are possible in the real world, i.e. the ASM must be indeterministic. We model the real world by defining an ASM rule that nondeterministically chooses an agent which – if possible – executes a protocol step. If for example the `Purse` agent is chosen, it is checked whether the inbox (of the connection to the terminal) is non-empty. If so, the first message is taken and processed. If the inbox is empty, another agent is chosen by the ASM. If the first message is of type `StartFrom`, the ASM rule describing the processing of a `StartFrom` message is executed. This rule is shown in listing 1. To generate the ASM rule, the activity diagrams are used as input (see Fig. 10).

It is not the purpose of this paper to describe the syntax and semantics of the ASM rules as they are used in the KIV system. Therefore, we give just an informal overview of the example rule. The content of the `StartFrom` message, i.e. the `value` and the `PurseData` of the `to` purse, are stored in local variables (lines 2 and 3 in listing 1). Next, it is checked

```
 1 STARTFROM#
 2 let value = inmsg.value,
 3     dataTo = inmsg.dataTo in
 4  if exlogcounter(ag) < LOGLENGTH
 5  then
 6   if state(ag) = IDLE
 7   then
 8    ...
 9    pdAuth(ag) .from := data(ag);
10    pdAuth(ag) .to := dataTo;
11    pdAuth(ag) .value := value;
12    data(ag) .sequenceNo := data(ag)
13                    .sequenceNo + 1;
14    state(ag) := EPR;
15
16    let encmess = mkMsgcontent(
17            STARTTO,pdAuth(ag)) in
18    let enc = encrypt(
19              sesskey(ag),encmess) in
20     outmsg(ag) := mkStartTo(enc);
21   else ABORT#
22  else ABORT#
```

Listing 1. ASM rule of processing a StartFrom message

if the exception log has free entries (line 4). The expression `exlogcounter(ag)` is specific for the formal model. `ag` is a variable for a `Purse` agent. As mentioned previously, the formal model contains an arbitrary number of `Purse` agents, and `ag` is the agent chosen in this ASM step. Agents are modeled with dynamic functions in the formal model, i.e. `exlogcounter` is a function that maps a `Purse` agent to the value of its `exlogcounter` attribute. It can be read as `ag.exlogcounter`. Similar functions exist for all attributes and associations of `Purse` (`pdAuth(ag)`, `state(ag)`, ...). Then the ASM rule checks whether the state of the card is set to `IDLE` (line 6) and performs some additional checks. If all tests succeed, several attributes and associations of the considered agent `ag`, in this case the purse agent, are updated (lines 9 - 14). An update means that the corresponding dynamic function is modified (therefore the function is called 'dynamic'). In a next step, a local variable `encmess` of type `Msgcontent` is created with the `msgflag` `STARTTO` that indicates a `StartTo` message and the current pay details (line 16). Then, this variable is encrypted by using a predefined encrypt function (line 18). The dynamic function `outmsg` that is generated automatically for each agent stores the message that is going to be sent after termination of the ASM rule for processing a `StartFrom` message. In our case, a `StartTo` message is sent next and stored `outmsg` (line 20). If one of the checks made in the beginning fails, the protocol aborts (line 21 and 22). The abortion is defined in a separate ASM rule called ABORT#. It can be seen that the structure of the ASM rule follows the structure of the activity diagram, but uses a different syntax, and has a semantics that is similar to MEL (e.g. copy semantics), but not identical (dynamic functions and inboxes are not part of MEL).

One relevant security property for Mondex is that the sum of money stored on all Mondex cards plus the sum of money stored in all (valid) exception logs does not increase or decrease over the time. This implies that no money is lost or created during a transfer of money, even in the presence of an attacker. This property can be formulated as a theorem in the formal model and proved with our theorem prover KIV. Of course, since a card may be recharged, this holds only for the use case 'Person-to-Person Payment'.
In previous work Haneberg [12] [36] developed a formal model based on ASMs and verification techniques to prove the security of an abstract model. This approach was successfully used in several case studies. The formal model introduced in this section is based on the one by Haneberg but uses application-dependent data types instead of a generic data format.

X. Related Work

Basin et al. [37] [38] 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.

Another approach that is related to ours is UMLSec developed by Jan Jürjens [8]. As in our approach he proposes to use UML for the development of security-critical applications. UMLSec defines a UML profile which adds security-relevant information to the UML diagrams. Security properties are expressed by using stereotypes. Jürjens provides tool support for verifying properties by linking the UML tool to a model checker resp. automated theorem provers. By doing so, the security properties mainly addressed are those that are expressed by the predefined stereotypes. The relevant formal model reflects an abstracted view of parts of the entire system. In our approach we concentrate on a transformation process that generates a formal model of the entire application which can be used for interactive verification of all system aspects. Based on the generated formal model, we can express and prove application dependent security properties such as "No money can be created within the Mondex application". In contrast to UMLSec we additionally focus on the generation of running Java Card code as well as the proof that this code is a refinement of the formal model.

In [39] 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 (OCL). The specified 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.

Alam et al. [40] present a model-driven security engineering framework for B2B-workflows. They introduce a domain-specific language for specifying access control policies which is used in the context of UML models. Furthermore, a UML profile for trust management is defined. After modeling a B2B application with UML, it is then translated into low-level web service artefacts using model-to-model and model-to-text transformations.

Deubler et al. present a method to develop security-critical service-based systems [41]. For modeling and verification the tool AutoFocus [42] 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.

XI. Conclusion

We presented our SecureMDD approach for the modeling of security-critical systems, especially smart card applications, with UML. Using this model-driven method UML models can be automatically translated into a formal model that is used to verify the security of our models. Furthermore, executable code can be generated automatically. In this paper we focused on the modeling with UML, i.e. the use of our UML profile which is tailored to security-critical applications and our Model Extension Language that we use in activity diagrams to describe cryptographic protocols. We propose a modeling technique that is easy to learn and abstracts from specifics regarding the formal specification or implementation. One disadvantage of UML is that it is only semi-formally defined. Since in our approach the UML models are translated into abstract state machines, we give them a formal semantics. We do not define a semantics for UML in general but only consider those parts that are used in our approach and which are interpreted in the context of security-critical applications. Our technique has evolved over several case studies. E.g. we have analyzed an application where a smart card is used as a copycard for a library [35]. Another case study deals with an application to buy cinema tickets using a mobile phone [12].

At the moment our approach is tailored to smart card applications but we are going to extend it, e.g. to service-oriented architectures, in the future. For example, the german electronic health card which consists of smart card parts as well as services that are realized as SOA, would fit into this domain. Another focus of future research is to build in the expression of security properties on the level of platform-independent modeling, for example by supporting the use of OCL expressions.

References

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

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

[3] Y. Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, Ed. Oxford Univ. Press, 1995, pp. 9 – 36.

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

[5] H. Grandy, K. Stenzel, and W. Reif, "A Refinement Method for Java Programs," in *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, ser. LNCS, vol. 4468. Springer, 2007.

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

[7] K. Stenzel, "Verification of Java Card Programs," Ph.D. dissertation, Universität Augsburg, Fakultät für Angewandte Informatik,URL: http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/,or http://www.informatik.uni-augsburg.de/forschung/dissertations/, 2005.

[8] J. Jürjens, *Secure Systems Development with UML*. Springer, 2005.

[9] 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 (ICSEA) 2007*. IEEE Press, 2007.

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

[11] 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*, vol. 20, no. 1, January 2008.

[12] H. Grandy, D. Haneberg, W. Reif, and K. Stenzel, "Developing Provably Secure M-Commerce Applications," in *Emerging Trends in Information and Communication Security (ETRICS)*, ser. LNCS, G. Müller, Ed., vol. 3995. Springer, 2006, pp. 115–129.

[13] "Eclipse Modeling Project," http://www.eclipse.org/modeling/.

[14] "Open Architecture Ware," http://www.openarchitectureware.org/.

[15] Object Management Group (OMG), "The unified modeling language," 2006. [Online]. Available: www.uml.org/

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

[17] C. Jones and J. Woodcock, Eds., *Formal Aspects of Computing*. Springer, January 2008, vol. 20 (1).

[18] 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*, U. G. J.-R. Abrial, Ed. Springer LNCS 5115, 2008.

[19] G. Schellhorn, H. Grandy, D. Haneberg, and W. Reif, "The Mondex Challenge: Machine Checked Proofs for an Electronic Purse," in *Formal Methods 2006, Proceedings*, ser. LNCS, J. Misra, T. Nipkow, and E. Sekerinski, Eds., vol. 4085. Springer, 2006, pp. 16–31.

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

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

[22] 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)*. Springer LNCS 1055, 1996, pp. 147–166.

[23] 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, pp. 350–357.

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

[25] N. Moebius, K. Stenzel, H. Grandy, and W. Reif, "Model-Driven Code Generation for Secure Smart Card Applications," in *20th Australian Software Engineering Conference*. IEEE Press, 2009.

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

[27] Sun Microsystems, “Java Card 3.0 Platform Specification,” http://java.sun.com/javacard/3.0/specs.jsp, 2008.
[28] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java (tm) Language Specification, Third Edition*. Addison-Wesley, 2005.
[29] Sun Microsystems, “Press release,” April 22 2008. [Online]. Available: http://www.sun.com/aboutsun/pr/2008-04/sunflash.20080422.1.xml
[30] *ISO 7816-4 – Identification Cards – Integrated cicuit(s) cards with contacts – Part 4: Organization, security and commands for interchange*, International Standards Organization, 1995.
[31] H. Grandy, N. Moebius, M. Bischof, D. Haneberg, G. Schellhorn, K. Stenzel, and W. Reif, “The Mondex Case Study: From Specifications to Code,” University of Augsburg, Technical Report 2006-31, December 2006, uRL: http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/.
[32] 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.
[33] O. Dubuisson, *ASN.1 - Communication Between Heterogeneous Systems*. Elsevier-Morgan Kaufmann, 2000.
[34] L. C. Paulson, “The Inductive Approach to Verifying Cryptographic Protocols,” *Journal of Computer Security*, vol. 6, pp. 85–128, 1998.
[35] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn, “Verifying Smart Card Applications: An ASM Approach,” in *International Conference on integrated Formal Methods (iFM) 2007*, ser. LNCS, vol. 4591. Springer, 2007.
[36] D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn, “Verifying Security Protocols: An ASM Approach,” in *12th Int. Workshop on Abstract State Machines, ASM 05*, D. Beauquier, E. Börger, and A. Slissenko, Eds. University Paris 12 – Val de Marne, Créteil, France, March 2005.
[37] D. Basin, J. Doser, and T. Lodderstedt, “Model Driven Security: From UML Models to Access Control Infrastructures,” *ACM Transactions on Software Engineering and Methodology*, pp. 39–91, 2006.
[38] T. Lodderstedt, D. A. Basin, and J. Doser, “SecureUML: A UML-Based Modeling Language for Model-Driven Security,” in *UML 2002 - The Unified Modeling Language, 5th International Conference*, 2002, pp. 426–441.
[39] M. Kuhlmann and M. Gogolla, “Modeling and validating Mondex scenarios described in UML and OCL with USE,” *Formal Aspects of Computing*, vol. 20, no. 1, pp. 79–100, January 2008.
[40] M. Alam, R. Breu, and M. Hafner, “Model-Driven Security Engineering for Trust Management in SECTET,” *JSW*, vol. 2, no. 1, pp. 47–59, 2007.
[41] 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*. ACM, 2004, pp. 115–124.
[42] M. Broy, F. Huber, and B. Schätz, “AutoFocus - Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme,” *Informatik, Forschung und Entwicklung*, vol. 14, no. 3, pp. 121–134, 1999.

APPENDIX

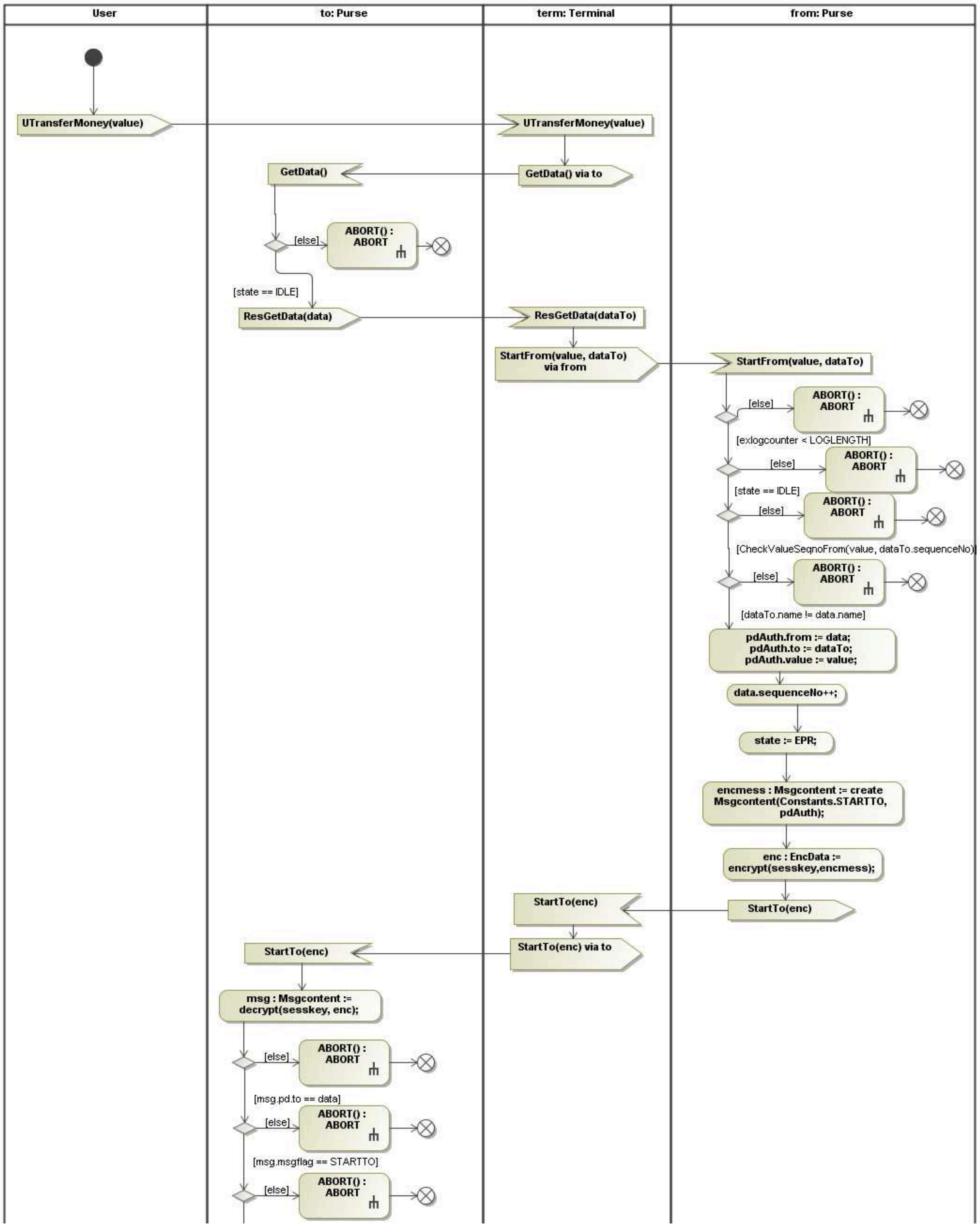


Fig. 14. Mondex Activity Diagram for Transferring Money, Part 1

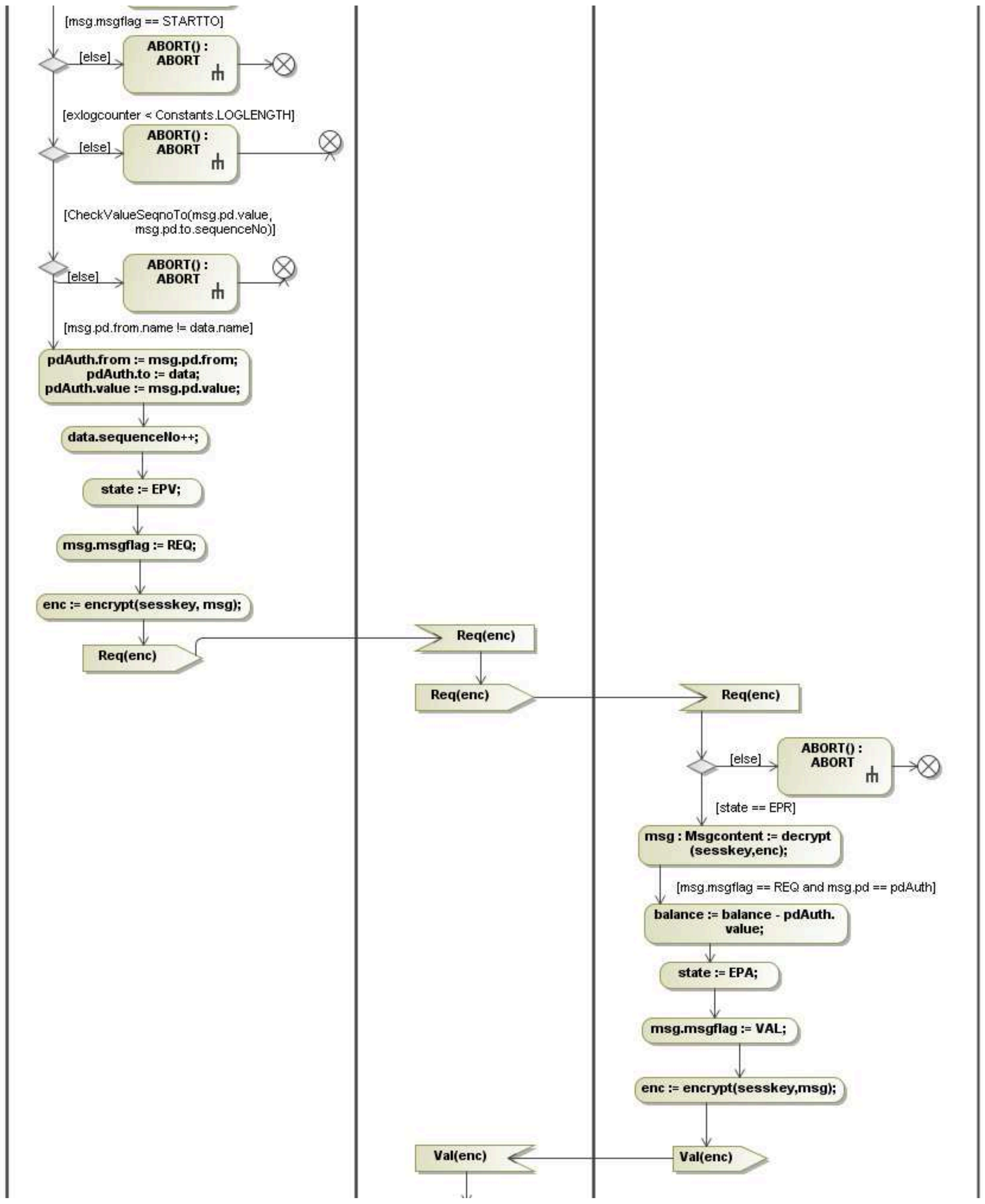

[msg.msgflag == STARTTO]
[else]
ABORT() : ABORT
[exlogcounter < Constants.LOGLENGTH]
[else]
ABORT() : ABORT
[CheckValueSeqnoTo(msg.pd.value, msg.pd.to.sequenceNo)]
[else]
ABORT() : ABORT
[msg.pd.from.name != data.name]
pdAuth.from := msg.pd.from; pdAuth.to := data; pdAuth.value := msg.pd.value;
data.sequenceNo++;
state := EPV;
msg.msgflag := REQ;
enc := encrypt(sesskey, msg);
Req(enc)
Req(enc)
Req(enc)
Req(enc)
[else]
ABORT() : ABORT
[state == EPR]
msg : Msgcontent := decrypt (sesskey,enc);
[msg.msgflag == REQ and msg.pd == pdAuth]
balance := balance - pdAuth. value;
state := EPA;
msg.msgflag := VAL;
enc := encrypt(sesskey,msg);
Val(enc)
Val(enc)

Fig. 15. Mondex Activity Diagram for Transferring Money, Part 2

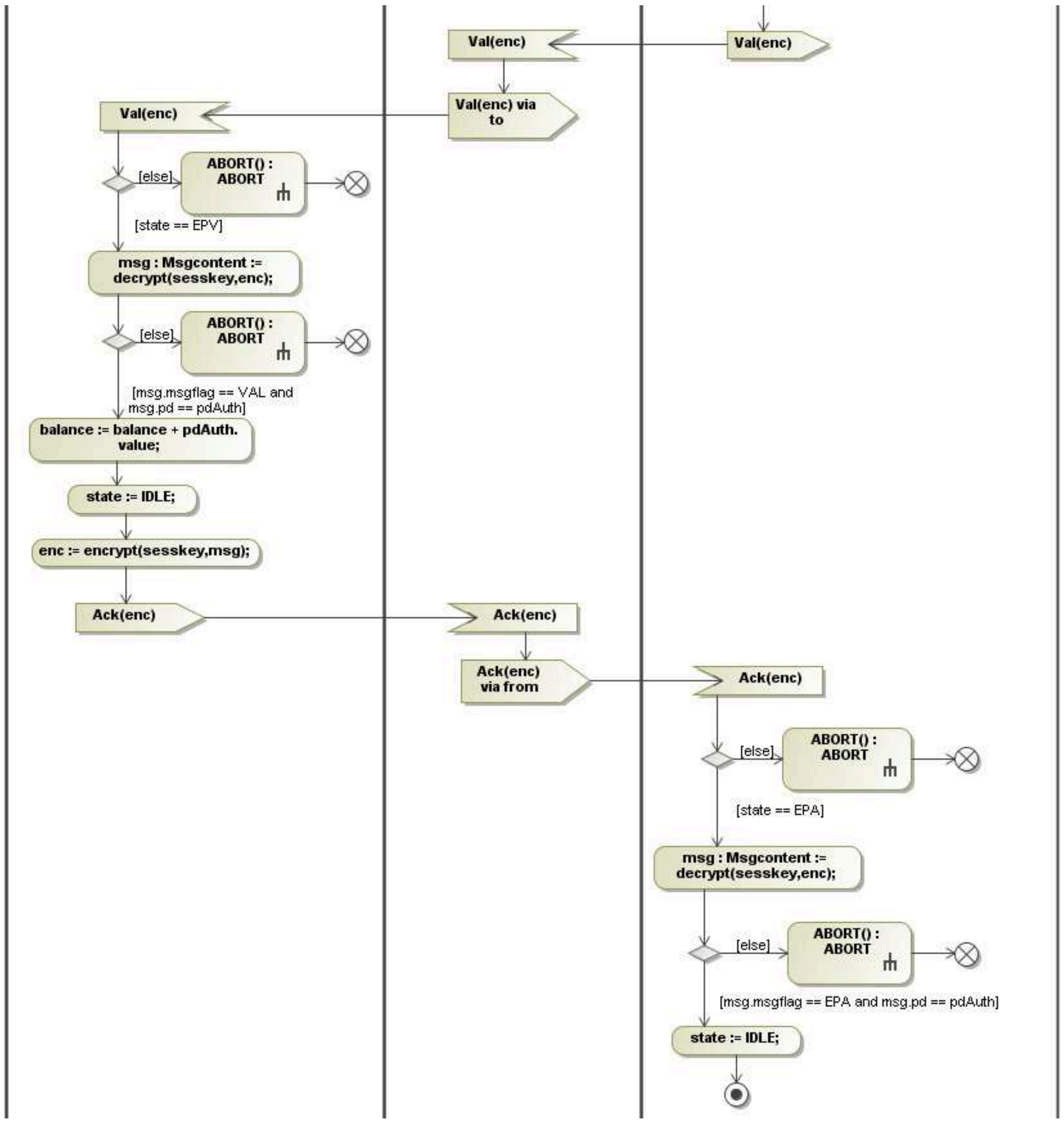

Val(enc)
Val(enc)
Val(enc) via to
Val(enc)
[else]
ABORT() : ABORT
[state == EPV]
msg : Msgcontent := decrypt(sesskey,enc);
[else]
ABORT() : ABORT
[msg.msgflag == VAL and msg.pd == pdAuth]
balance := balance + pdAuth. value;
state := IDLE;
enc := encrypt(sesskey,msg);
Ack(enc)
Ack(enc)
Ack(enc) via from
Ack(enc)
[else]
ABORT() : ABORT
[state == EPA]
msg : Msgcontent := decrypt(sesskey,enc);
[else]
ABORT() : ABORT
[msg.msgflag == EPA and msg.pd == pdAuth]
state := IDLE;

Fig. 16. Mondex Activity Diagram for Transferring Money, Part 3

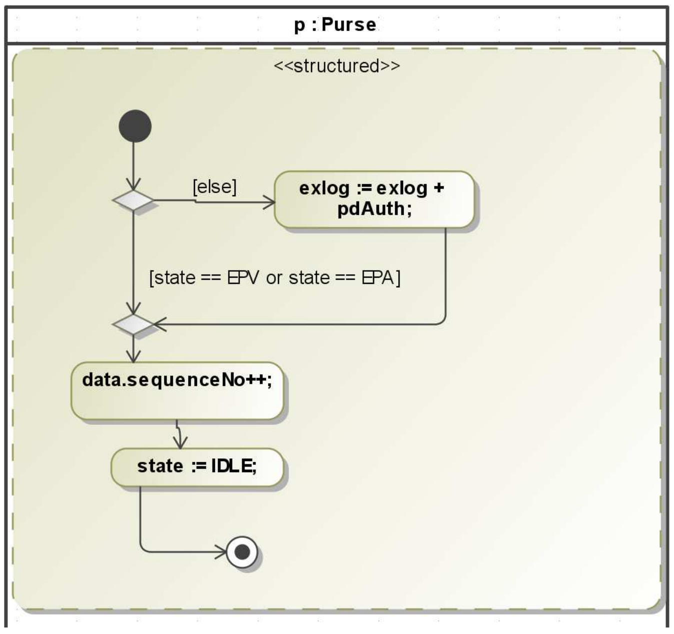


Fig. 17. Mondex Activity Diagram for Subactivity Abort()

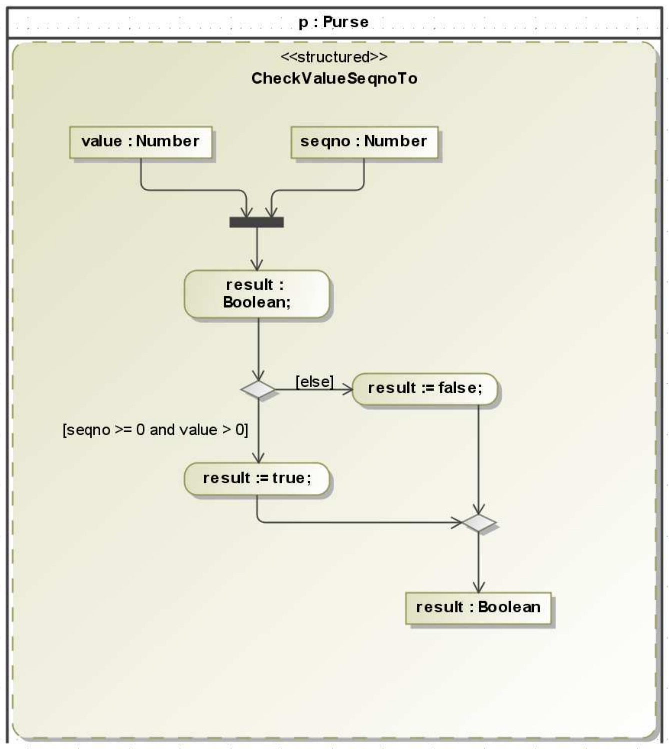


Fig. 18. Mondex Activity Diagram for Subactivity CheckValueSeqnoTo(value : Number, seqno : Number): Boolean

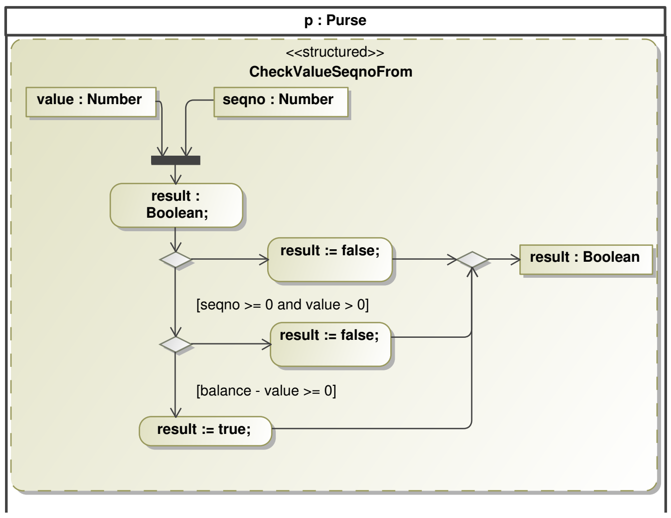


Fig. 19. Mondex Activity Diagram for Subactivity CheckValueSeqnoFrom(value : Number, seqno : Number): Boolean