# Security requirements formalized with OCL in a model-driven approach

**Marian Borek, Nina Moebius, Kurt Stenzel, Wolfgang Reif**

**Angaben zur Veröffentlichung / Publication details:**

Borek, Marian, Nina Moebius, Kurt Stenzel, and Wolfgang Reif. 2013. "Security requirements formalized with OCL in a model-driven approach." In *2013 3rd International Workshop on Model-Driven Requirements Engineering (MoDRE), 15 July 2013, Rio de Janeiro, Brazil*, edited by Ana Moreira, Gunter Mussbacher, João Araújo, Nelly Bencomo, and Pablo Sánchez, 65–73. Piscataway, NJ: IEEE. https://doi.org/10.1109/modre.2013.6597265.



# Security Requirements Formalized with OCL in a Model-Driven Approach

Marian Borek, Nina Moebius, Kurt Stenzel, Wolfgang Reif
Department of Software Engineering,
University of Augsburg, Germany
{borek,moebius,stenzel,reif}@informatik.uni-augsburg.de

***Abstract*—Security requirements are properties that have to be guaranteed for an application. Such guarantees can be given using verification. But there is a huge gap between security requirements expressed with human language and formal security properties that can be verified. This paper presents the use of OCL to formalize security requirements in a model-driven approach for security-critical applications. SecureMDD is such a model-driven approach. It uses UML to model the application and OCL to specify the security requirements. From the application model and the contained OCL constraints, a formal specification of the application including the security properties is generated automatically. This specification is used to verify application-specific security properties that matches a lot of security requirements much better than application-independent security properties like secrecy, integrity and confidentiality. We demonstrate how to concretize security requirements as well as the use of OCL constraints to specify security requirements, the transformation from OCL constraints into algebraic specifications and the use of those specifications to verify the security requirements using an electronic ticketing system as a case study.**

## I. Introduction

Requirements describe how a system should behave and which properties it has to fulfill. They are mostly given informally by stakeholders during the analysis phase of the software development process. This can be done systematically by employing use cases, user stories or other documentation but the requirements still remains informal. Security requirements are important for every system which handles assets (e.g., valuable or private data). An example for such a system is an electronic ticketing system with the security requirements "fare dodgers will be detected" and "a user receives his paid tickets". This system and its requirements are explained in detail in this paper. Other examples for security-critical applications are an online banking system which has to ensure that "no money can be lost", an E-Voting system which has to guarantee that "votings can not be associated with their voters" or a digital currency system where "money can not be duplicated". These security requirements are application-specific (like many other security requirements too) and do not match existing application-independent security properties like secrecy, integrity, confidentiality or even role-based access control. Of course, an instantiation of an application-independent security property is application-specific but an "application-specific property" is not a part of an application-independent class of properties like role-based access control. In our opinion there is an infinite number of security requirements that can not be expressed by a finite number of high-level keywords. OCL (Object Constraint Language) [1] is a suitable language for formalizing such security requirements if the static view of the application is specified with UML (Unified Modeling Language) class diagrams. Such diagrams must include all relevant information that is necessary to express the properties. To verify application-specific properties the whole application has to be specified and formalized. This means that the entire system behavior, including operations is completely specified. With our model-driven approach called SecureMDD [2][3] it is possible to model the complete application including its full behavior. This paper focuses on the formalization of security requirements with OCL and their transformation into algebraic specifications to verify application-specific security properties. Because many security requirements are application-specific, this is a major improvement compared to other model driven approaches [4][5][6] that only consider application-independent security properties.

This paper is structured as follows. Section 2 gives an overview of our model-driven approach. Section 3 illustrates the requirements of an electronic ticketing system and section 4 concretizes these requirements. Section 5 shows the formalization of security requirements with OCL, section 6 describes the transformation into algebraic specifications and section 7 shows how they are used for verification. Section 8 discusses related work and section 9 concludes this paper and outlines future work.

## II. The SecureMDD Approach

SecureMDD is a model-driven approach for security-critical systems. It focuses on application domains like e-Commerce and e-Government and makes it possible to develop secure applications by specifying them with extended UML and verifying their application-specific properties.

Fig. 1 illustrates the SecureMDD approach in detail. From a platform-independent UML model of an application runnable code as well as a formal specification for interactive verification with the theorem prover KIV [7] and an ASLan++ specification to find security flaws using model checking are generated automatically.

The platform-independent model uses a UML profile as well as a platform-independent and easy to use modeling

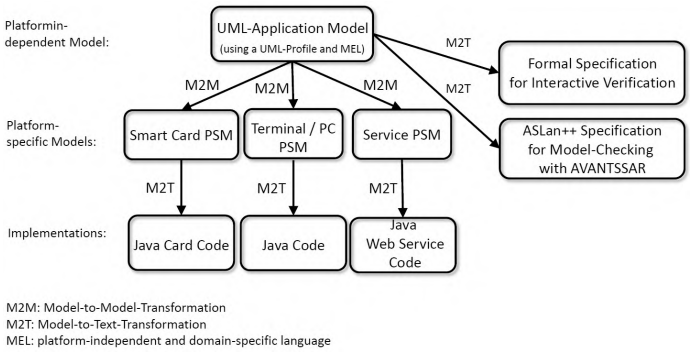


Fig. 1. SecureMDD Approach

language MEL [8] [3] to model security-critical applications. This happens in two steps. Firstly, three platform-specific models are generated from the application model, one for each supported platform (Java Card, Java and Java for Web Services). These models are used as documentation of the gradual code generation. Secondly, the application code is generated from the platform-specific models.

The platform-specific models are tailored to their target platforms, e.g., Java Card for a smart card, Java for a terminal or a PC, and Java based Web services for a service. A smart card is a secure device that can be accessed only via a predefined interface and is tamper-proof: nobody has access to the operating system or the internal memory directly. A terminal (e.g., an ATM) is also a secure device that receives instructions from a user and can have an interface for the communication with a smart card. A PC is a personal computer where the user has access to internal storage. A service will be deployed on servers that are assumed to be secure devices. Services can be connected to by terminals or other services over a network. To perform tasks, a service can orchestrate other services, or several autonomous services can collaborate together. Services can only be accessed via their specified interfaces.

To specify the system we use class diagrams, activity diagrams, sequence diagrams and a deployment diagram. The class diagrams are used to describe the static part of the system and define classes with attributes which can be annotated with stereotypes from our UML profile. Those classes represent transferred messages or agents like smart cards, terminals, PCs and services. The activity diagrams in combination with MEL are used to specify the dynamic behavior of the system. They model the cryptographic protocols used by the system agents as well as the agents' behavior after receiving a message. The sequence diagrams represent a higher abstraction of this dynamic behavior for documentation purposes, while the deployment diagram is used to describe the communication structure, attacker abilities and connection security.

In order to read and transform an application model we use the modelling tools provided by the Eclipse[1] project. The formal specification is generated using the framework oAW[2] and the three platform-specific models are generated using model-to-model (M2M) transformations written in QVT [9]. Model-to-text (M2T) transformations based on oAW are used to generate the executable code from the platform-specific models (PSMs). Each PSM is transformed to one or more Java packages that contain the full source code for each agent type. The programming language for terminals, PCs and services is Java, while smart card code is written in Java Card [10], [11], a version of Java tailored for smart cards with their severe resource limitations.

Due to our model-driven approach which allows us to model the complete application including its full behavior, we are able to demonstrate the formalization of application-specific security requirements inside an application model and their transformation into algebraic specification to verify these security requirements.

## III. Electronic Ticket Example

ETicket describes an electronic ticketing system for traveling with trains. The functional system requirements are:

- The system has to allow its users to buy tickets over the Internet by using a personal computer.
- A bought ticket is stored on the user's smart card and can be managed (e.g., displayed and deleted) using a personal computer with a smart card reader.
- The tickets can only be validated and "punched" by an inspector with a valid inspector device.

Beside the functional requirements which describe the functional system behavior, the non-functional requirements are:

- The entrance and exit areas of the trains are not permanently controlled and do not have turnstiles.
- A Dolev-Yao [12] attacker is assumed. (An Dolev-Yao attacker is able to read, send and suppress messages on every connection and at any time)
- "Fare dodgers" will be detected.
- A user pays only for tickets he bought.
- A user receives his paid tickets.

The last three non-functional requirements are security requirements that describe security properties that have to hold for the system.

## IV. Concretization of Security Requirements

Expressing correct and precise security requirements is not a trivial task. The problem is, that informal security properties often are too general. Hence, the security requirements have to be made concrete. Inspired by attack trees [13] and the work of Lamsweerde [14] [15] we use the following procedure to systematically state security requirements more precisely. At the beginning, the root of the tree contains a security requirement. The nodes of the tree contain attacks to break the security requirement and the leaves contain techniques to prevent an attack. If no suitable technique to prevent an attack can be found the security requirement on the root of

[1] http://www.eclipse.org/

[2] openArchitectureWare: http://www.openarchitectureware.org/

the tree has to be restricted. The attack trace can remain in the tree but should be marked as no longer possible. Because the security requirement is restricted, the other already considered traces with suitable security measures can not suddenly lead to attacks. The goal is to find all attacks with a security measure for each attack. If no security measure can be found the security requirement has to be restricted.

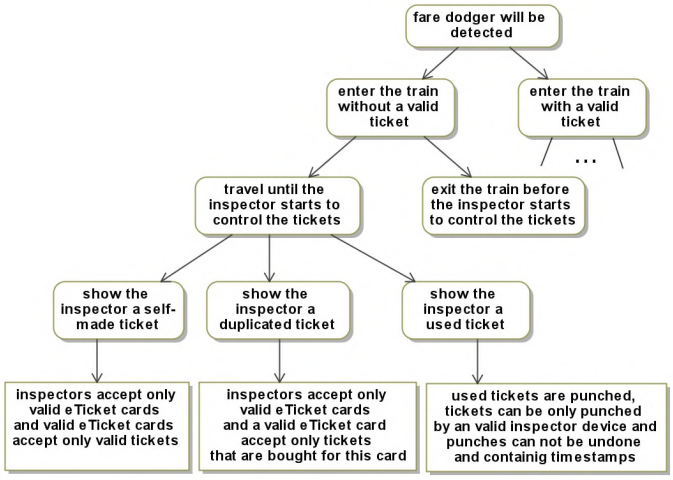


Fig. 2. Alternative of an attack tree

Fig. 2 depicts such an (incomplete) tree. The root contains the security requirement "fare dodgers will be detected". This requirement is tried to be broken by an attack. From the root node there are two options: to *enter the train with a valid ticket* or to *enter the train without a valid ticket* (someone travels more than he has paid for). To enter the train with a valid ticket is a legal option and can not be forbidden. To enter the train without a valid ticket could be ensured by a turnstile. But this goes against the requirement "the entrance and exit areas of the trains are not permanently controlled and do not have turnstiles". So, both options could lead to an attack. After entering a train without a valid ticket there are also two options. To *travel until the inspector starts to control the tickets* or to *exit the train before the inspector starts to control the tickets*. In the first case there are three possible options. *show the inspector a self-made ticket*, *show the inspector a duplicated ticket* or *show the inspector a used ticket*. For each option there are suitable security measures. Self-made tickets are not possible because an inspector accepts only valid eTicket cards (the smart card containing the ticket) and a valid eTicket card has to accept only valid tickets. Duplicated tickets are not possible because an inspector accepts also only valid eTicket cards and a valid eTicket card has to accept only valid tickets that are bought for this card. And the last attack, to show the inspector a used ticket, is detectable because used tickets are punched, tickets can only be punched by a valid inspector device and punches can not be undone and contain timestamps. The leaves of the tree describe the security measures overlap with some requirements (e.g., tickets can only be punched by an valid inspector device) and describe new one (e.g., valid eTicket cards accept only valid tickets). If the new requirements are security requirements, they can be used as helper security requirements to prove the actually needed security requirements. The attack tree shows that the first case, travel until the inspector starts to control the tickets, is secure. But in the second case the traveler exits the train at the next station before the inspector starts to control the tickets. This can not be forbidden and describes a real attack. With the knowledge of that attack trace, the security requirement can be restricted. The attack trace has shown that the only security measure to identify fare dodgers is an inspector. So, the security requirement is changed into "fare dodgers will be detected if they are controlled by an inspector". The previously found attack trace should be marked as no longer possible and the process to find attacks has to be continued with the current leaves that are not marked as impossible and which do not represent security measures.

As a result, the more concrete security requirement "fare dodgers will be detected if they are controlled" and some helpful additional requirements (e.g., valid eTicket cards accept only valid tickets) are obtained. Similarly, the mentioned process applied on the security requirement "a user receives his paid tickets" restricts the requirement into "a paid ticket is stored on the server until it has been received by the card". This is because of the requirement "a Dolev-Yao attacker is assumed". The concretization of security requirements is done during the analysis phase of the development process. In the design phase the informal security requirements can be formalized by OCL constraints, which need a UML model of the application. From the resulting OCL constraints provable security properties can be generated automatically.

## V. Security Requirements formalized with OCL Constraints

To formalize informal security requirements, OCL is a suitable language. OCL makes it possible to express not only application-independent properties (e.g., secrecy) but also application-specific properties. This is very important because many security requirements are application-specific (e.g., "a paid ticket is received by a card or it is ready to be retrieved"). The considered application-specific properties are all properties that can be defined as invariants over the systems states. But beside the main security requirements it can be helpful to specify initial assumptions and helper properties. Such assumptions and properties are also application-specific and expressible with OCL. To define OCL constraints, the application model is needed. More precisely, only the classes of the system participants and their attributes are necessary.

Fig. 3 shows part of the eTicket class diagram that is relevant for the OCL constraints. It depicts the class *ApplicationConstraints* that is annotated with the same-named stereotype. This class contains three security-critical OCL constraints and two OCL operations annotated with the stereotype ≪OCLOperation≫ which can be used inside the OCL constraints. The system is mainly described by the system participants that are defined as classes with the stereo-

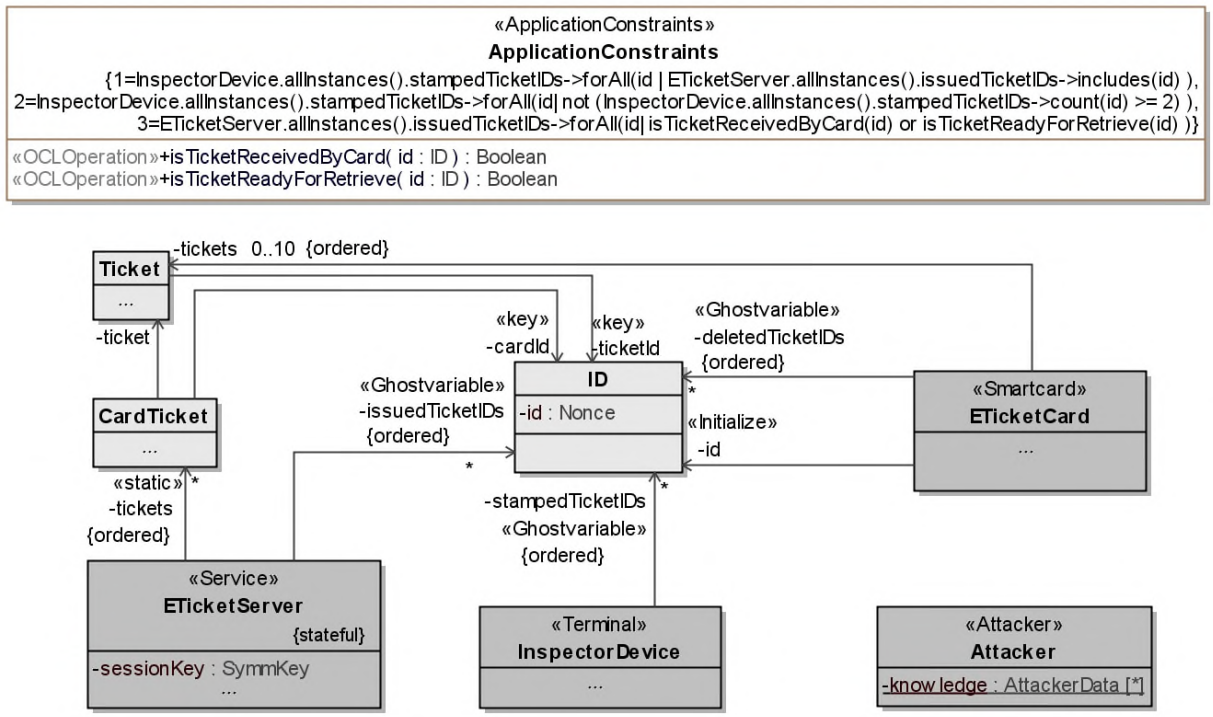


Fig. 3. Part of eTicket class diagram

type ≪Service≫, ≪Terminal≫, ≪PC≫, ≪Smartcard≫, ≪User≫ or ≪Attacker≫. The application model describes the system participants *ETicketServer*, *InspectorDevice*, *ETicketCard* and the *Attacker*. They are colored dark gray and their attributes are light gray. The complete SecureMDD model is available on our website[3]. The security requirements were defined in the analysis phase and it can be possible that they have no relation to the application model that results in the design phase. Hence, the first step is to relate the informal security requirements to the application model. A mentioned security requirement is "fare dodgers will be detected if they are controlled". The model specifies punched/stamped tickets with the *stampedTicketIDs* attribute of an *InspectorDevice*. But the model does not define if a user is a "fare dodger". Hence, the security requirement can not be directly expressed with this application model. The solution is the previously mentioned attack tree (see Fig. 2). The security requirement is divided by several attack nodes. If one node can be described as a security property (e.g., attack is not possible) with OCL on the application model, the children of that node are also included. Otherwise, if all children of a node can be described, the union of the constraints describe the constraint for the parent node. The full attack tree describes only three attacks that occur several times: using a self-made ticket, duplicating a ticket or using a ticket twice. Using a self-made ticket can be prevented with the security requirement "only tickets issued by an *eTicketServer* will be punched". Duplicating a ticket and using a ticket twice can be prevented with the security requirement "a ticket will not be punched multiple times".

The first OCL constraint formalizes the aforementioned security requirement "only tickets issued by an *eTicketServer* will be punched". Therefore, only the inspector device (*InspectorDevice*) which is used to validate and "punch" tickets as well as the ticket issue server (*ETicketServer*) are involved. Each class that represents a system participant (excluding the attacker) has an arbitrary number of instances if no further restrictions apply. Hence, the modeler can invoke the predefined OCL operation *allInstances* on those classes to get a *set* of their instances. A *set* is a collection and there are three different kinds of collections considered in this paper: *sets* (not ordered and duplicate free), *bags* (not ordered and with duplicates) and *sequences* (*lists*) (ordered and with duplicates). The dot operator, which usually is used to access properties like class attributes from single class instances can also be used on collections. This is an abbreviated form to access each element from the collection. Hence, the expression *InspectorDevice.allInstances().stampedTicketIDs* returns a bag of all *stampedTicketIDs* from all *InspectorDevices*, whereby *stampedTicketIDs* is a class attribute of *InspectorDevice*. With the arrow operator, functions for collections like *forAll*, *includes*, *excludes*, *count*, etc. can be invoked. *forAll* means that for each collection element the following expression has to be true and *includes* checks if an element is contained in the collection. This way, the informal security requirement "only tickets issued by an *eTicketServer* will be punched" is formalized using the OCL constraint depicted in List. 1.

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

```
inv:
InspectorDevice.allInstances().stampedTicketIDs->
forAll(id | ETicketServer.allInstances().
  issuedTicketIDs->includes(id)
)
```

Listing 1. Only tickets issued by an *eTicketServer* will be punched

It states that all *stampedTicketIDs* from all *InspectorDevices* are contained in a *bag* of all *issuedTicketIDs* from all *ETicketServers*. The constraint is very similar to the informal requirement which shows the big advantage of OCL when formalizing security requirements. But to understand why the informal requirement and the OCL constraint are equal it is necessary to understand the process of issuing and "punching" tickets. When a ticket is issued, the *ETicketServer* creates a new and unique ticket id of type *ID* and adds it to the *issuedTicketIDs* list. Then it creates a ticket as an instance of class *Ticket* containing this id and sends it to an *ETicketCard* that stores the ticket in the list *tickets* modeled as a class attribute, which can store up to ten tickets. To "punch" a ticket, it is marked as "punched" on the smart card and the whole ticket including its id is sent to an *InspectorDevice* which stores the id of the ticket in its list *stampedTicketIDs*. Because the uniqueness of a ticket is realized with ids, only the ids have to be stored. The lists *issuedTicketIDs* and *stampedTicketIDs* are ghost variables. This means that they are only necessary for the specification and verification of security properties and are therefore accessed by OCL constraints but they are not needed in the running system. In particular, a smart card (*ETicketCard*) has limited resources and could not store a list with an arbitrary number of elements. Hence, the ghost variables are not used to ensure properties, but to show that the properties hold.

Another security requirement is that "a ticket will not be punched twice".

```
inv:
InspectorDevice.allInstances().stampedTicketIDs->
forAll(id| not (InspectorDevice.allInstances().
          stampedTicketIDs->count(id) >= 2)
)
```

Listing 2. A ticket will not be punched twice

This is formalized by the second OCL constraint in the *ApplicationConstraints* class and is additionally illustrated in List. 2. It checks whether the collection of all *stampedTicketIDs* in all *InspectorDevices* is duplicate free. Therefore, each *punched* ticket id must not occur two times or more in the union of all lists of *stampedTicketIDs*.

```
inv:
ETicketServer.allInstances().issuedTicketIDs->
forAll(id| isTicketReceivedByCard(id) or
          isTicketReadyForRetrieve(id)
)
```

Listing 3. A paid ticket is received by a card or it is ready to be retrieved

List. 3 shows the third OCL constraint, which formalizes the security requirement "a paid ticket is received by a card or it is ready to be retrieved". This requirement can be expressed with the application model without customization. The constraint checks for each *ETicketServer* whether each issued ticket is either received by a card or that the ticket is still in the process of being retrieved. Because the ticket id makes each ticket unique it is sufficient to check only the ids. The checks happen by the invocation of the OCL operations *isTicketReceivedByCard* and *isTicketReadyForRetrieve*. OCL operations are class operations without side effects. To achieve this, the UML operation property *isQuery* has to be true. This is set automatically if an operation applies the stereotype ≪OCLOperation≫. The behavior of such operations is specified with OCL in the body condition of the operation. For example, the body condition of the operation *isTicketReceivedByCard* is *ETicketCard.allInstances().tickets.id→includes(id) or ETicketCard.allInstances().deletedTicketIDs→includes(id)*. This means, that a ticket is received on a card if the ticket is stored on the card or has already been deleted from it. More precisely, it means that a ticket with a unique ticket id is received by a card only if the list of stored tickets (see the *ETicketCard* attribute *tickets*) contains a ticket with the same ticket id or if the ticket id is contained in the *deletedTicketIDs* list. OCL operations are useful to split large OCL expressions and to be reused in several OCL constraints.

The main security requirements are now formalized and can be transformed automatically into algebraic specifications. But for verification initial assumptions and helper statements are also needed. Some attributes in Fig. 3 have the stereotype ≪Initialize≫. These attributes have to be initialized during the deployment. But it has to be specified which values are valid. Hence, this can be described in OCL in a class annotated with the stereotype ≪InitializeConstraints≫ (see Fig. 4).

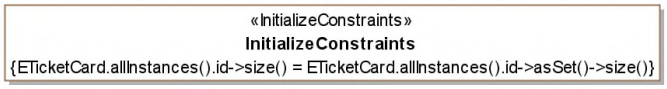


Fig. 4. Initialize Constraints

The initialize constraint shown in Fig 4 formalizes the assumption that all *ETicketCards* have initially different ids. More precisely, it means that the size of all ids (containing duplicates) is equal to the size of all ids without duplicates. The elimination of duplicates is done by converting the bag of ids into a set with the OCL operation *asSet*. Such initialize constraints are important for the verification and for the correct deployment. Hence, it is security-critical that all *ETicketCards* have different ids. If two cards had the same id, it could not be guaranteed that "a ticket will not be punched twice" because a ticket, stored on a server to be retrieved, could be overwritten by a new ticket bought with another card with the same id. This depends on the fact that an *ETicketServer* stores the tickets that are ready to be retrieved in the attribute *tickets* that is a container which stores key-value pairs whereby a key can occur only one time in the container. Hence, it overwrites elements with the same key. In this case the key is the card id which is defined by the stereotype ≪key≫ that is applied on the id attribute of *CardTicket* (see Fig. 3).

The last kind of constraints are helper constraints. They are further statements that must hold true for the system.

Such statements can be helpful during the verification of the actual security properties. They are very useful especially if the application developer and the verifier are different people. The application developer has had some thoughts during development of the protocols; Those thoughts are security-relevant and should be documented.

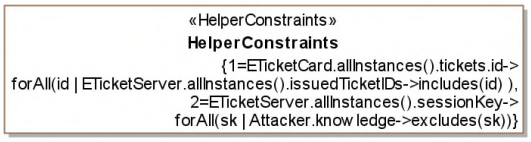


Fig. 5. Helper Constraints

Fig. 5 depicts a class with the stereotype ≪HelperConstraints≫. It contains two helpful constraints. The first helper constraint results during the concretization of the security requirements and describes that all tickets stored on a valid card have to be issued by a valid server. More precisely, all ticket ids from all *ETicketCards* are contained in the collection of all *issuedTicketIDs* from all *ETicketServers*. This is quite similar to the security requirement "only tickets issued by an *eTicketServer* will be punched". But, the mentioned helper constraint has to hold so that the security requirement can hold. Helper constraints can also include initialize constraints. The aforementioned assumption that all *ETicketCards* have different ids should hold initially but it is obvious that it should also hold at any time after the initialization. Therefore, it should also occur as a helper constraint. However, not all initialize constraints are also helper constraints (e.g., "all lists of stamped and issued tickets have to be empty" is an initialize but no helper constraint).

The second helper constraint describes secrecy, an application-independent security property. In our approach it is described as: an exchanged session key that is stored on an *ETicketServer* has to be secret. More precisely, the *sessionKey* attributes from all *ETicketServer* must not be part of the attacker's knowledge. The attacker is represented by the class *Attacker* with the applied stereotype ≪Attacker≫. We assume a Dolev-Yao attacker, which means that the attacker is able to read, send and suppress messages on every connection and at any time. Hence, we assume only one attacker and define the attacker knowledge as a static attribute of the attacker class. The attacker knowledge is a set of *AttackerData*, which can be any security data type. Security data types are predefined and contain a data type for symmetric keys (*SymmKey*), asymmetric keys (*PublicKey*, *PrivateKey*), an arbitrary number used only once in a cryptographic communication (*Nonce*), secrets (*Secret*), hashes (*HashedData*), etc.

## VI. From OCL Constraints to Algebraic Specifications

After the application model is created and the security requirements are formalized with OCL, the formal model is generated automatically. It describes a world in which system participants (called agents) exchange messages according to the protocols, and an attacker tries to break the security. The participants behavior is described with Abstract State Machines (ASMs) [16] and the class diagrams, attacker abilities and OCL constraints are specified with algebraic specifications. The translation of OCL constraints into algebraic specifications is done automatically. For parsing of OCL constraints the Eclipse OCL[4] is used. It also validates the OCL constraints against the model and builds up an abstract syntax tree which allows us to transform the OCL constraints into algebraic specifications.

```
InspectorDevice.allInstances()->
  collect(temp1| temp1.stampedTicketIDs)->
  forAll(id|
    ETicketServer.allInstances()->
    collect(temp2| temp2.issuedTicketIDs)->includes(id)
  )
```

Listing 4. Extended OCL constraint of the OCL constraint shown in List. 1

List. 4 depicts an extended representation of the OCL constraint shown in List. 1. It formalizes the security requirement "only tickets issued by an *eTicketServer* will be punched" and results through the building of the abstract syntax tree from the original OCL constraint. OCL constraints use short cuts to access a property (e.g., an attribute, association, or operation) for each element of a collection. This is done by a dot operator on a collection followed by the property. This kind of short cuts is replaced with a *collect* operation by the Eclipse OCL framework.

```
onlyIssuedTicketsCanBePunched :
|-  onlyIssuedTicketsCanBePunched(
      InspectorDevice-stampedTicketIDs,
      ETicketServer-issuedTicketIDs
    )
   <->
    forAll1(ETicketServer-issuedTicketIDs,
            collect1(InspectorDevice-stampedTicketIDs,
              InspectorDeviceAllInstances)
    )
```

Listing 5. Algebraic specification of the OCL constraint depicted in List. 4

List. 5 illustrates the axiom that describes the behavior of the predicate *onlyIssuedTicketsCanBePunched* that represents the mentioned OCL constraint. A predicate is a function with boolean as return type. *onlyIssuedTicketsCanBePunched* needs two parameters, *InspectorDevice-stampedTicketIDs* and *ETicketServer-issuedTicketIDs*. They are dynamic functions that map an agent to a list of ids. *InspectorDevice-stampedTicketIDs* stores stamped ticket ids for inspector devices and *ETicketServer-issuedTicketIDs* stores issued ticket ids for eTicket server. These dynamic functions are initially empty and are filled during the execution of the protocol. More details about dynamic functions and our formal specification can be found in [17]. *onlyIssuedTicketsCanBePunched* is specified by the call of another predicate called *forAll1*. It also has two parameters, *ETicketServer-issuedTicketIDs* and the result of the function *collect1*. The used predicates like *forAll1* and *collect1* are numerated because those functions are not generic like the OCL *collect* or *forAll* operation. This means, that for

[4]http://www.eclipse.org/modeling/mdt/?project=ocl

each concrete invocation another function has to be generated. The extended representation of the OCL constraint shown in List. 4 is very similar to its algebraic representation depicted in List. 5. First, the OCL constraint collects all inspector devices using *InspectorDevice.allInstances()*. The algebraic specification for the constraint uses the constant *InspectorDeviceAllInstances*. This means that it does not have to be a parameter of the predicates *onlyIssuedTicketsCanBePunched* and *forAll1* but can be used inside these predicates. After that, all stamped ticket ids from all inspector devices are collected by the *collect* operation. This is done in the algebraic specification by the *collect1* function.

```
collect1_emp :
|-  collect1(InspectorDevice-stampedTicketIDs,[]) = [];

collect1_rec :
|-  collect1(InspectorDevice-stampedTicketIDs,ag ' +agents)
    = InspectorDevice-stampedTicketIDs(ag) +
      collect1(InspectorDevice-stampedTicketIDs,agents);
```

Listing 6. Specification of collect1

*collect1* (see List. 6) is defined recursively and consists of two parts. The first part describes the end of the recursion (*collect1_emp*). This means that if the list of inspector devices is empty the return value is an empty list. The other part (*collect1_rec*) describes the recursion. It returns a list of stamped ticket ids for an inspector device (*ag*) appended by the result of the next invocation of *collect1* but without the already considered inspector device (*ag*). *ag* ' means that *ag* is converted into a list with *ag* as an element. To invoke this function the parameters *InspectorDevice-stampedTicketIDs* and *InspectorDeviceAllInstances* are needed (see List. 5).

```
forAll1_emp :
|-  forAll1(ETicketServer-issuedTicketIDs,[]) = true;

forAll1_rec :
|-  forAll1(ETicketServer-issuedTicketIDs,a_ID ' +a_listofID)
    = includes1(
        collect2(ETicketServer-issuedTicketIDs,
          ETicketServerAllInstances), a_ID
      )
      and forAll1(ETicketServer-issuedTicketIDs,a_listofID);
```

Listing 7. Specification of forAll1

List. 7 shows the algebraic representation of the OCL operation *forAll* for the mentioned OCL constraint. It is also defined recursively and checks for all stamped ticket ids if they occur in the list of all issued ticket ids from all *ETicketServers*. The end of the recursion (*forAll1_emp*) is described similarly to that for the collect operation. If the list of stamped ticket ids is empty it returns true. The main behavior is defined by (*forAll1_rec*). It checks for one stamped ticket id (*a_ID*) if it is contained in the list of all issued ticket ids from all *ETicketServer* by using the functions *includes1* and *forAll1* recursively.

```
includes1:
|-  includes1(a_listofID,a_ID) = (a_ID \in a_listofID);
```

Listing 8. Specification of includes1

List. 8 depicts the *includes1* function. It takes a list of issued ticket ids and a ticket id and checks if the list contains the id using the predefined *\in* operator. The includes operator *\in* could be also directly used instead of the *includes1* function. But *\in* is an infix operator and *includes* is an operation. Hence, for automatic generation it is more systematical to define an algebraic function for each OCL operation.

The *collect2* function represents the second invocation of the OCL operation *collect* inside the actual discussed constraint. It is specified similarly as the *collect1* function except that *collect2* returns a list of all issued ticket ids from all *eTicketServer* instead of all stamped ticket ids from all inspector devices.

OCL invariants and initial constraints are greatly translatable into our formal model and are used for verification of security requirements.

## VII. Verification of Security Requirements

In SecureMDD the algebraic specifications generated from OCL expressions are used to verify security requirements. Our approach differentiates between three kinds of constraints: application constraints, initialize constraints and helper constraints. The application and helper constraints are formalized with OCL as invariants and have to hold at all times. This requires that they also hold initially. But many constraints do not hold for an arbitrary initialize state. For example, in our eTicket case study the application constraint "a ticket can not be punched twice" can not hold if the initialize constraint "all *ETicketCards* have initially different ids" does not hold. Therefore, it is important to formalize the initial state with initialize constraints. Usually, the application modeler does not know all initialize assumptions that are needed for verification. Hence, some additional initialize assumptions have to be specified by the verifier manually.

$$InitCons \wedge AddInitAssum \vdash AppCons \wedge HelpCons \quad (1)$$

Theorem 1 should be initially true and describes that the application constraints (AppCons) and helper constraints (HelpCons) have to follow from the initialize constraints (InitCons) and some additional initialize assumptions (AddInitAssum). But the application constraints and the helper constraints also have to hold before and after each protocol step (see theorem 2).

$$AppCons \wedge HelpCons \xrightarrow{STEP} AppCons \wedge HelpCons \quad (2)$$

A protocol describes system participants exchanging messages and a protocol step defines the behavior of a system participant from receiving a message until sending the next one.

With the two mentioned theorems and a formal model that describes the behavior of the application, the security requirements that are represented by the application constraints can be verified. Verification with our approach is described in [18].

## VIII. Related Work

Security requirements are necessary when developing secure applications. They describe security properties that have to hold for the system. If a possible security hazard is not considered by the security requirements, the system is probably not secure. Security requirements can be application-independent but also application-specific. Application-specific requirements are defined on the application-model and do not match to an application-independent class of properties like confidential, replay-protected or role-based access control. In the past, there have been many model-driven approaches which consider only application-independent security. However, in this paper we focus on application-specific security requirements.

SecureSOA developed by Menzel et al. [6] fosters a model-driven approach based on the modeling of security requirements in system design models. It uses UML stereotypes like ≪User Authentication≫, ≪Non-Repudiation≫ and ≪Trust≫ to express security requirements. They are translated into an instance of a meta-model for security policies from which enforceable WS-SecurityPolicies are generated. The authors mention that SecureSOA is not limited to the illustrated constraints. It supports custom enhancements but only by extending the approach with more application-independent security requirements.

Hatebur et al. [19] describe the development of UMLsec [5] design models based on security requirements. They describe the whole system (including security requirements) with OCL pre- and postconditions. This results in very large and hard to understand OCL specifications that are detailed enough to generate the whole application. From those OCL specifications an UMLsec model is generated. In contrast, we model the application with UML and extend the model by OCL constraints that describe security requirements that have to be proved. Furthermore, because UMLsec can only express application-independent security properties the approach from Hatebur et al. is also restricted to application-independent security properties.

Saleem et al. [20] introduce a domain-specific modeling language for security objectives. They use UML and the standard stereotypes ≪confidentiality≫, ≪integrity≫ and ≪availability≫ to model the security requirements of security-critical applications. They do not generate any further artifacts.

The aforementioned projects use standard security keywords to express application-independent security requirements. Other projects focus on access control that usually need more application-specific input for instantiation but in our opinion such properties are still not application-specific properties. SecureUML developed by Basin et al. [4] is such an approach. It is model-driven and uses OCL to describe access control for security critical systems. The application model including the access control constraints is transformed into access control information for Enterprise JavaBeans.

Montrieux et al. [21] is also a model-driven approach for specification and verification of role-based access control properties. They use UML stereotypes (e.g., ≪granted≫ and ≪forbidden≫) and OCL for specifying access control.

The approach developed by Deubler et al. [22] also considers a model-driven approach and focus on secure service-based systems. It uses the tool AUTOFOCUS, which is similar to UML, to model an application and specification patterns to express authorization predicates. Those patterns as well as their transformation into temporal logic (CTL) are supported by AUTOFOCUS. The resulting CTL formulas are model checked.

There are also other approaches that are not restricted to application-independent security properties. Pavlidis et al. [23] describe such an approach for modeling security requirements. They make requirements engineering systematic but the resulting documents are too imprecise to verify security properties.

Another approach is described by Haley et al. [24]. They describe a framework for security requirements elicitation and analysis. For an existing application a context has to be specified manually and the requirements are formalized on that context. Our approach is not to specify the context but to model the whole application including an platform-independent implementation of the participants behavior. From that model, a formal specification as well as runnable code is generated. This causes that the framework and not the system modeler has to ensure that the context is correct and that the implementation does not introduce conflicting behavior. Furthermore, they do not use OCL or UML and do not generate a formal specification that includes the whole system behavior and the requirements that have to be proved.

Paja et al. [25] [26] [27] describe modeling and analysis of security requirements. They focus on socio-technical security and consider standard security (e.g., integrity, non-repudiation, need-to-know), authorization and trust. Additionally, they are able to detect conflicting security requirements as well as conflicts between security requirements. But they do not specify the full dynamic system behavior so they are not able to verify security requirements like “the sum of all account balances plus the amount of all the money that has been withdrawn from cash machines is unchanged”.

OCL constraints are of course also used in non-model-driven approaches. The tool KeY, which was developed by Beckert et al. [28] uses OCL to specify security requirements for object-oriented code verification. With this approach it is possible to specify application-specific properties with OCL to verify them. But this approach uses object-oriented source code as foundation to define OCL constraints and does not solve the question how application-specific properties are expressible on UML models. However, model-driven approaches simplify the development of security-critical applications by abstracting from implementation details. Hence, source code verification is much more difficult.

## IX. Conclusion and Future Work

Usually, security requirements are given in an informal language. To verify such requirements they have to be formalized.

In this paper, we have shown that a lot of security requirements are applications-specific. We have also illustrated how such informal and application-specific security requirements are concreted and expressed with OCL constraints on the application model. Furthermore, we have depicted the automatic translation from the resulting OCL constraints into algebraic specifications and have explained how the specifications are used in association with the formal model to verify the mentioned security requirements.

Future work is the support of OCL pre- and postconditions for operations. If some operations have to be implemented very efficiently, they should be usually realized with the source code programming language. This can be done in our model-driven approach by operations marked as *manual*. Those operations can be used inside the application model and have to be implemented manually in the generated source code. But for verification it is necessary to know some details about the behavior of the operation. Those details can be specified by OCL pre- and postconditions.

## REFERENCES


[1] *Object Constraint Language, Version 2.3.1*, Object Management Group (OMG), 2012.

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

[3] M. Borek, N. Moebius, K. Stenzel, and W. Reif, "Model-driven development of secure service applications," in *Software Engineering Workshop (SEW), 2012 35th Annual IEEE*. IEEE, 2012, pp. 62–71. [Online]. Available: http://dx.doi.org/10.1109/SEW.2012.13

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

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

[6] M. Menzel and C. Meinel, "Securesoa modelling security requirements for service-oriented architectures," in *Services Computing (SCC), 2010 IEEE International Conference on*. IEEE, 2010, pp. 146–153.

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

[8] N. Moebius, K. Stenzel, and W. Reif, "Modeling Security-Critical Applications with UML in the SecureMDD Approach," *International Journal On Advances in Software*, vol. 1, no. 1, 2008.

[9] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, Object Management Group (OMG), 2011.

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

[11] Z. Chen, *Java card technology for smart cards: architecture and programmer's guide*. Prentice Hall PTR, 2000.

[12] 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.

[13] B. Schneier, "Attack trees," *Dr. Dobbs journal*, vol. 24, no. 12, pp. 21–29, 1999.

[14] A. Van Lamsweerde, "Goal-oriented requirements engineering: A guided tour," in *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. IEEE, 2001, pp. 249–262.

[15] A. van Lamsweerde, "Elaborating security requirements by construction of intentional anti-models," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 148–157. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999421

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

[17] N. Moebius, K. Stenzel, and W. Reif, "Generating formal specifications for security-critical applications - a model-driven approach," in *ICSE 2009 Workshop: International Workshop on Software Engineering for Secure Systems (SESS'09)*. IEEE/ACM Digital Libary, 2009.

[18] ——, "Formal verification of application-specific security properties in a model-driven approach," in *Proceedings of ESSoS 2010 - International Symposium on Engineering Secure Software and Systems*. Springer LNCS 5965, 2010.

[19] D. Hatebur, M. Heisel, J. Jürjens, and H. Schmidt, "Systematic development of umlsec design models based on security requirements," in *Fundamental Approaches to Software Engineering*. Springer, 2011, pp. 232–246.

[20] M. Saleem, J. Jaafar, and M. Hassan, "A domain-specific language for modelling security objectives in a business process models of soa applications," *AISS*, vol. 4, no. 1, pp. 353–362, 2012.

[21] L. Montrieux, M. Wermelinger, and Y. Yu, "Tool support for uml-based specification and verification of role-based access control properties," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 456–459.

[22] 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.

[23] M. Pavlidis, S. Islam, and H. Mouratidis, "A case tool to support automated modelling and analysis of security requirements, based on secure tropos," in *IS Olympics: Information Systems in a Diverse World*. Springer, 2012, pp. 95–109.

[24] C. B. Haley, R. Laney, J. D. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *Software Engineering, IEEE Transactions on*, vol. 34, no. 1, pp. 133–153, 2008.

[25] E. Paja, A. K. Chopra, and P. Giorgini, "Trust-based specification of sociotechnical systems," *Data & Knowledge Engineering*, 2012.

[26] E. Paja, F. Dalpiaz, and P. Giorgini, "Identifying conflicts in security requirements with sts-ml," 2012.

[27] F. Dalpiaz, E. Paja, and P. Giorgini, "Security requirements engineering via commitments," in *Socio-Technical Aspects in Security and Trust (STAST), 2011 1st Workshop on*. IEEE, 2011, pp. 1–8.

[28] B. Beckert, R. Hähnle, and P. H. Schmitt, *Verification of object-oriented software: The KeY approach*. Berlin, Heidelberg: Springer-Verlag, 2007.