Accepted Manuscript

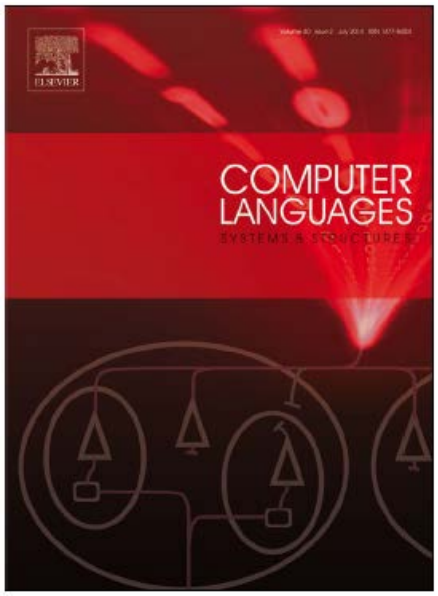
A model driven architecture for the development of smart card software

Hidayet Burak Saritas, Geylani Kardas

DOI: 10.1016/j.cl.2014.02.001

To appear in: *Computer Languages, Systems & Structures*

Published online: 17 February 2014



Please cite this article as: Hidayet Burak Saritas, Geylani Kardas, A model driven architecture for the development of smart card software, Computer Languages, Systems & Structures (2014), doi: 10.1016/j.cl.2014.02.001

A Model Driven Architecture for the Development of Smart Card Software

Hidayet Burak Saritas and Geylani Kardas[1]

Ege University International Computer Institute, 35100, Bornova, Izmir, Turkey
saritasburak@gmail.com, geylani.kardas@ege.edu.tr

Abstract: Smart cards are portable integrated devices that store and process data. Speed, security and portability properties enable smart cards to have a widespread usage in various fields including telecommunication, transportation and the credit card industry. However, the development of smart card applications is a difficult task due to hardware and software constraints. The necessity of the knowledge of both a very low-level communication protocol and a specific hardware causes smart card software development to be a big challenge for the developers. Written codes tend to be error-prone and hard to debug because of the limited memory resources. Hence, in this study, we introduce a model driven architecture which aims to facilitate smart card software development by both providing an easy design of smart card systems and automatic generation of the required smart card software from the system models. Differentiating from the previous work, the study in here contributes to the field by both providing various smart card metamodels in different abstraction layers and defines model-to-model transformations between the instances of these metamodels in order to support the realization of the same system on different smart card platforms. Applicability of the proposed methodology is shown for rapid and efficient application development in two major smart card frameworks: Java Card and ZeitControl Basic Card. Lessons learned during the industrial usage of the architecture are also reported in the paper. Finally, we discuss how the components of the architecture can be integrated in order to provide a domain-specific language for smart card software.

Keywords: Model driven architecture, metamodel, model transformation, smart card, Java Card, Basic Card.

[1] Corresponding author. Addr: Ege Universitesi Uluslararasi Bilgisayar Enstitusu, Bornova Metro Duragi Karsisi, 35100, Bornova, Izmir Turkey. Tel.: +90-232-3113223; Fax.: +90-232-3887230

1. Introduction

Smart cards are portable, integrated circuit devices that securely store and process data [1]. These tiny computers with their own memories and processors have a widespread usage in various fields including telecommunication, transportation, banking and healthcare. For instance the operation of a cellular phone is directly based on a smart card which carries an identification number unique to the owner, stores personal data and prevents operation if removed. Also most of today's credit cards are in fact smart cards that store account information for a bank customer and provide authorization and authentication for electronic money transfers.

Since hardware and software capabilities of a smart card are very limited compared to a desktop personal computer, the development of smart card applications is a difficult task. The necessity of having a deep knowledge of both a very low-level smart card communication protocol [2] and a specific hardware causes smart card software development to be a big challenge for the developers. Limited memory resources force developers to deal with very primitive data structures. The design of the software for the process of incoming hexadecimal data packages and preparation of the outgoing data packages byte-by-byte, again in hexadecimal format, are difficult and time-consuming jobs for the developers. Development environments dedicated to the smart cards are mostly incapable of code debugging and this makes written codes tend to be error-prone.

Like other smart card developers, we also experienced above difficulties in the design and implementation of various smart card software which were developed during our academic research (e.g. [3], [4], [5]) or for commercial purposes. Based on our experience, we may conclude that working in a higher abstraction level different from the code level is mandatory

for an efficient design and implementation of smart card software systems. Within this context, Model Driven Engineering (MDE) (or Model Driven Development (MDD)) [6], which aims to change the focus of software development from code to models, may also provide easy and efficient production of smart card software. In such an MDE environment, card developers can graphically design their system models conforming to metamodel(s) at various abstraction layers which in fact present entities and their relations needed for a smart card system, and then software codes needed for the designed system are automatically generated as the result of a model to text transformation. Hence, in this paper, we introduce a MDD process which aims to facilitate the smart card software development by both providing an easy design of smart card systems and automatic generation of smart card programs. We use Model Driven Architecture (MDA) ([7], [8]) which is one of the realizations of MDD to support the relations between platform independent and various platform dependent smart card entities to develop software for smart cards.

Following the derivation of smart card metamodels in different abstraction layers, definition and implementation of model-to-model (M2M) transformations for the instances of these derived metamodels were performed. Finally, model-to-text (M2T) transformations were provided to automatically generate required software codes.

Apart from the related work (e.g. [9], [10], [11], [12]) which mainly considers model driven smart card software development only specific for the Java Card framework [13], the study presented in this paper introduces a platform independent metamodel (PIMM) for smart card systems in order to free the developer from taking into consideration the specific needs of different smart card platforms such as Java Card. Furthermore, the proposed PIMM enables the developer to model the smart card system conforming to data transmission and data

storage standards brought by ISO/IEC 7816 standards family [2]. This standards family includes various standards for smart card development such as the physical characteristics, electrical interface, transmission between smart cards and host devices and personal verification.

On the other hand, transformability from the general smart card PIMM to the dedicated smart card platforms is also presented in this study. Metamodels of two major smart card frameworks, Java Card [13] and ZeitControl Basic Card [14], are defined as the platform specific smart card metamodels (PSMM). Hence, applied M2M transformations and following M2T transformations provide the implementation of the same smart card system on different execution platforms.

The rest of the paper is organized as follows: A brief discussion on smart card technology is given in Section 2. Section 3 includes the proposed smart card PIMM and related modeling environment. Platform specific metamodels and modeling tools for the MDD of Java Card and Basic Card applications are given in Section 4 and 5 respectively. Defined model transformations between general smart card models and Java Card and Basic Card models are discussed in Section 6. Section 7 covers the automatic code generation from platform specific card instance models. Evaluation of the study by considering the lessons learned during the industrial usage of the architecture is reported in Section 8. Related work is given in Section 9. Section 10 concludes the paper.

2. Smart Card Technology

Smart cards are tiny computers with their own processor and memory. For instance, the integrated circuit on a bank credit card is in fact a smart card. Also, subscriber identity

modules (SIM), used inside our cellular phones, are smart cards that store subscriber information to use the phones properly. A smart card includes a micro-processor, read-only memory (ROM), random access memory (RAM) and electrical erasable programmable ROM (EEPROM). Operating system of the card is stored in ROM. Similar to the main memories of our desktop PCs, applications run on RAM. Finally, EEPROM stores applications and data while the card is unpowered. Connection points on the card support the input/output and communication with the host computers [1], [15]. Fundamental functionality (e.g. commands for interchange, structure of transmitted data packages) and characteristics (such as electrical interface or contact type) for smart cards are defined with ISO/IEC 7816 standards [2].

Depending on the usage type, smart cards may be classified as *contact* or *contacless*. Contact cards should be inserted in a card reader which is directly connected to a host computer whereas contactless cards do not need to be inserted or contacted physically for operation. Such kind of cards communicates with and is powered by the reader through Radio Frequency (RF) induction technology.

Like Transmission Control Protocol/Internet Protocol (TCP/IP) and related data package transmission used in computer networks, there also exists a standard communication infrastructure defined for smart cards again in ISO/IEC 7816 Standards family. Whole communication between a smart card and a host computer (terminal) is provided by the exchange of Application Protocol Data Unit (APDU) packages [2]. A host terminal application sends *Command APDU* packages and receives *Response APDU* packages from the smart card. Based on the received command APDU, related application residing in the card is chosen by the smart card operating system and the chosen application now begins to process the received command APDU packages and sends back responses in response APDU

packages to the terminal. A command APDU package has a 4-byte mandatory header with the following fields: CLA, INS, P1, P2. In addition to these fields, it may also include data up to 256 bytes. Table 1 lists command APDU fields and their descriptions.

Table 1: Command APDU fields

Field	**Description**
CLA	Instruction class indicating the type of command
INS	Instruction code indicating the specific command (e.g. get data or write data)
P1	Instruction parameter for the command
P2	Instruction parameter for the command
Lc	Length of data being sent
Data	Data being sent
Le	Length of the expected response

A response APDU package is sent by the card to the reader. The package contains a mandatory 2-byte status word and up to 256 bytes of data (Table 2). Status word (SW1 and SW2) denotes whether the requested operation successfully performed or not. For example, hexadecimal 0x9000 value for SW1 and SW2 bytes indicates that the operation has been successfully performed.

Table 2: Response APDU fields

Field	**Description**
Data	Response data (It can be empty in case of a failure or intentionally no response data is requested).
SW1	Command processing status
SW2	Command processing status

3. A PIMM for Smart Cards

Both programming interface and structure of the programs stored in the smart cards vary despite the above discussed unique hardware and standard communication protocol. Some of the cards support file systems and allows file-oriented data processing. On the other hand, a group of smart cards includes middlewares that enhance actual read/write operations over different abstractions such as object-orientation. In this section, we discuss a PIMM for smart card applications which enables software developers to model their card software conforming to common smart card data transmission and data storage standards without considering the specifications of programming for different smart card platforms (e.g. Java Card [13]). The preliminary version of the metamodel is introduced in [16].

Derivation of the metamodel entities and their relations was performed as the result of a feature-oriented domain analysis. With the collaboration of smart card software developers, both features of smart card software and dependencies among those features were determined. While studying on the features, we also specified the system constraints. Further, the analysis also enabled us to determine mandatory and optional features. For instance, a smart card application should support the APDU protocol and own the required properties for the convenient communication between the host computers. On the other hand, it would be better to have authorization features for accessing the data stored on the card in most of the situations. However it is not a must for all smart card applications. Obtained feature model supported the creation of the smart card PIMM in question.

Figure 1 depicts the proposed common smart card metamodel which is encoded according to the well-known Ecore meta-metamodel included in the Eclipse Modeling Framework (EMF) [17]. Similar to Unified Modeling Language (UML) class diagrams, boxes with compartments

represent meta-entities while lines show the associations like inheritance and composition between the card entities. Some of the meta-entity attributes are suppressed in the figure to provide clear reading.

The key entity of the metamodel is the *Application* which represents a smart card program. The platform independent modeling of a smart card software starts with the inclusion of an Application instance. As will be discussed later, platform dependent counterpart of this entity can be an applet for Java Cards or a file definition for Basic Card applications. At least one instance of the Application entity should be included in a smart card application model. Notice that the implementation of such constraints required for the metamodel entities and their relations was provided with using the well-known Object Constraint Language (OCL) [18], [19]. Special OCL rules were written for describing those constraints which also constitute the static semantics for the models.

Figure 1 approximately here

Figure 1: The metamodel for smart card applications

Another important meta-entity is the *APDU*. Communication between smart cards and terminals is realized over APDU packages. Each communication in an interaction model can be represented with an APDU instance. The metamodel includes two specialization of this meta-entity as expected: *CommandApdu* and *ResponseApdu*. Comforming to ISO/IEC 7816 specifications, these APDU types cover proper attributes in our metamodel. For instance, instruction bytes (*INS*) in a *CommandApdu* entity manages the instructions that will be executed in the card application. Data and/or notification on a card process are returned by

ResponseApdu instances. A *ResponseApdu* includes a group of *SWCondition*s which define status word types of ISO/IEC 7816 APDU responses.

For a smart card application, all needed *CommandApdu* and *ResponseApdu* instances are encapsulated within an *APDUOperation* meta-entity. That entity provides the definition of data types and variables needed during the execution of commands or card operations. The relation between a modeled *Application* and every added *APDUOperation* is established over an *Including* entity.

Almost all smart card software requires a user authorization via the input of a valid personal identification number (PIN). Hence a meta-entity called *PIN* is included in the metamodel. Before the establishment of the connection between the smart card and the host, value of the PIN should be verified. Terminal sends an APDU including an entered PIN value and the application in the smart card checks that value. Validation of the PIN is mandatory for the execution of successor commands. An *Access_key* entity in a smart card model associates *PIN* instance(s) with a *Application*.

Constant and *Data* entities are used for the definition of information processed in the card application. Each data owns name, type and value attributes. Some fundamental smart card data types are already defined in the model with *SCDataTypes* entity. These types are "*number*", "*numberArray*", "*string*", "*boolean*" and "*byte*". User defined data types can be included in the models via *DataUnit* entities. Hence, new data structures can be constructed from the collection of existing data types (e.g. *SCDataTypes* or again *DataUnit*s). Association of the new *DataUnit* instances with an *Application* is realized over an *Instance*.

Now let us consider the modeling of smart card applications according to above discussed PIMM since the MDE of software systems naturally requires a phase in which the modeling of the system is realized. Models are the main artifacts of the development process and software developers should be supported with appropriate modeling tools for the production of these artifacts. Hence, in this study, a set of modeling tools were designed and implemented to support our approach proposed for the MDE of smart card systems.

The modeling tools introduced in this paper were developed on Eclipse platform [20] by using the Graphical Modeling Framework (GMF). GMF [21] is a framework for building graphical modeling tools for various domains. In our study, we (1) provide domain models for various smart card environments (including both platform independent and platform specific) as Ecore metamodels, (2) prepare graphical elements representing the smart card domain elements and their relations, and (3) map smart card components with the graphical nodes to generate smart card modeling tools / editors according to GMF specifications.

Considering the generation of the modeling tool for our general smart card models, above introduced metamodel naturally presented a base on which the related modeling editor is built. Since we already derived the Ecore encoding of the PIMM, remaining work just covered the tasks given in (2) and (3).

The screenshot in Figure 2 shows a fragment from the modeling environment provided by the platform independent smart card modeling tool. Developers can visually model their smart card systems by using the editor palette shown in the right side of Figure 2. The palette includes the nodes and links that can be used for preparing a smart card application instance model. Each node or link is a graphical element which represents a unique meta-entity of our

PIMM. The editor environment also supports developers in model consistency and prevents wrong relation establishments between smart card model elements. Further, constraints on the model entities and their relations (e.g. compartment constraint, number of relations constraints, relationship source and destination constraint, entity-relation consistency constraint) are also checked automatically within the editor environment. As previously mentioned, such constraints were implemented by using OCL [18]. For this purpose, built-in features of the GMF were utilized. The "gmfmap"s were prepared for the components of each Ecore metamodel (both PIMM and PSMMs) in our study. Inside each "gmfmap" description file, OCL constraints can be inserted e.g. for the compartment setting between the top node and the child nodes or creating the link mappings such as association or aggregation. OCL constraints were also written for the determination of the relation types inside Feature-Value expressions.

A developer simply drag-and-drops a smart card software component from the palette to the design area, sets the attribute values for the component (if applicable) and associates it with already existing smart card elements according to the above metamodel definitions and constraints. Modeling of a smart card software is started with the inclusion of an *Application* instance (shown in the upper left of Figure 2). Successor instances (such as APDUs, PIN, data or smart card operation types) are added into the model by connecting them with proper association links (*Instance*, *Including* or *Access_key*) again based on the metamodel specifications.

Figure 2 approximately here

Figure 2: Modeling enviroment for the development of platform independent smart card software

As can be seen from the Figure 2, some elements such as *Application* and *APDUOperation* instances own a set of compartments in order to encapsulate other elements. Hence, an appropriate settlement of the elements is achieved. Furthermore, metamodel constraints for each model element are checked automatically by the editor. These constraints include unique settlement of the elements (e.g. *CommandApdu* and *ResponseApdu* elements can only be inserted into an *APDUOperation*) or the use of the correct association links (e.g. *Access_key* relation can only be established between a smart card *Application* and a *PIN* element). The modeling editor also controls the derivation and use of data variables and constants only inside the suitable elements such as *Application* and *DataUnit*.

Each model designed in the editor environment is in fact an instance of our PIMM. Figure 2 also shows such an instance model for a classic e-purse smart card system. Suppose a smart card software is required for an e-purse (wallet) application. In this application, customers use their smart cards during online payment operations. The electronic money transfer is realized based on the customer account information stored on the smart card. The smart card application instance (seen at the left of Figure 2) provides the storage of the customer account information and user authentication (over a PIN) and includes appropriate smart card operations (modeled as APDUOperations) for e-money transfer (credit / debit) and balance inquiry (getBalance APDUOperation in the model).

4. Modeling Java Card Applications

Today Java Card [13] is perhaps the most preferred type of smart cards and its application programming interface, called Java Card API, is one of the widely chosen software libraries for the development of smart card software. The Java Card technology provides application development for smart cards and other memory-limited devices by using the features of the

Java programming language. However only a subset of the Java programming language can be used due to the resource limitations of a smart card. For example, only *boolean*, *short* and *byte* can be used as primitive data types. Integers, characters and the String class can not be used. Furthermore multidimensional arrays, dynamic class loading, garbage collection, threads, object serialization and cloning are also not supported in Java Card.

A card program written for a Java Card is named as the card *applet*. Java Card Framework (JCF) supplies the API to develop smart card applications that conform to the ISO/IEC 7816 standards [2]. After a card applet is prepared using the Java Card API, a Converted Applet (CAP) file is formed from the written applet and any other on-card class files needed by this applet. Converter loads this CAP file to the smart card and the interpreter installs the applet encapsulated in the CAP file. After the installation, all the permanent Java objects are created on the card and are ready to use [22].

Software developers use the Java packages and classes distributed within the Java Card API for the implementation of smart card applications. However, above discussed restrictions of the API make the development process difficult and time-consuming. In order to cope with those deficiencies, a MDE methodology for the development of Java Card programs is defined in this study. The methodology includes the visual modeling of Java Card programs and then automatic code generation from the prepared system models. For this purpose, we first need a metamodel for describing Java Card entities and their relations. The definition of a M2T transformation originating from this metamodel enables us to automatically generate Java Card software codes. In the following, we discuss the related metamodel and visual modeling of the Java Card software based on this metamodel. M2T transformations defined for the Java Card programs will be later discussed.

Based on the specifications of the Java Card API, we derived a metamodel for the Java Card programs. The metamodel includes Java Card components and their associations. According to our approach, the derived metamodel is considered as a PSMM for smart cards within the MDA perspective [7]. We provided a transformation mechanism from the platform-independent smart card metamodel (introduced in Section 3) to this PSMM in order to obtain Java Card platform-specific counterparts of the models conforming to the platform-independent smart card metamodel. Related transformation mechanism will be discussed later in Section 6 of this paper.

Like the PIMM discussed in the previous section, derived Java Card metamodel is encoded again according to the Ecore meta-metamodel [17]. That encoding provides the XML Metadata Interchange (XMI) serialization of our Java Card metamodel and hence the metamodel can be employed in various M2M transformations as source or target metamodels. Figure 3 depicts the derived Java Card metamodel. Some of the entity attributes are omitted for clarity.

The key entity in the metamodel is the *Applet*. An applet is a Java Card program residing on a smart card. It receives incoming requests from the host, processes the request and responses back to the host program. The *APDU* meta-entity in our metamodel represents packets of data that conform to the specifications of the ISO/IEC 7816. As discussed before, an APDU can be a command or a response APDU. In Java Card technology, the host application sends a command APDU and a Java Card applet responds with a response APDU. In fact, a Java Card applet remains idle until it receives a command APDU.

Figure 3 approximately here

Figure 3: The Java Card metamodel

The *PIN*, as its name denotes, provides an interface for declaring passwords for the authorized access to the smart card programs. The *OwnerPIN* is a concrete implementation of the PIN interface. It maintains the PIN value, the maximum length of PIN allowed, the maximum number of times an incorrect PIN can be presented before the PIN is blocked and the remaining number of times an incorrect PIN presentation is permitted before the PIN becomes blocked. Java Card API provides a ready-to-use class for this PIN implementation with the same name. So, the OwnerPIN is also represented with a meta-entity in the proposed Java Card metamodel. The traditional way is to provide an attribute with type OwnerPIN for each Java Card applet. Properties (e.g. maximum length and maximum try limit) of this PIN are set when the applet is installed into the smart card. Before opening a communication session between the host application and the Java Card applet, the verification of the entered PIN is performed inside the smart card: The host application sends an APDU containing a PIN value; the applet processes that PIN value and checks whether the PIN is valid. The session is not opened until the PIN is verified.

The upper part of the Figure 3 consists of the meta-entities representing the base of the JCF. It resembles the metamodel of the core of the Java programming language with the definition of classes, parameter types, fields and methods. The relations between the Java Card classes (*Association, Aggregation, Generalization* and *Implements*) are also given in the metamodel. However, we organize the metamodel as it reflects the restrictions originating from the use of just a subset of the Java programming language for the Java Card. For instance, *JClass* represents the Java class entity for the JCF which is the base class for all Java Card classes

(e.g. Applet, APDU and any user-defined Java Card classes). We introduce JClass meta-entity as a limited version of a classic Java Class with inabilities such as it can not be cloned or serialized. Likewise, primitive data types in Java Card API (*JCDataTypes*) are restricted with only three primitive data types: byte, short and boolean. These primitive types are represented in the metamodel with *JCByte*, *JCShort* and *JCBoolean* meta-entities. Representations of the exception types specific to the Java Card (e.g. *PINException*, *APDUException*) are also included in the metamodel.

The derivation of the Java Card metamodel has allowed us to develop another modeling tool for smart card software development. By following the development steps discussed in Section 3, a GMF-based tool has been produced. The screenshot given in Figure 4 shows the modeling environment presented by this tool. Again a developer can simply drag-and-drop a Java Card software component from the palette to the design area, sets the attribute values for the component and associates it with already existing smart card elements according to the Java Card metamodel definitions and constraints. All attributes of the smart card components can be set by using the properties tab of the editor (shown at the bottom of Figure 4).

Figure 4 approximately here

Figure 4: Modeling environment for the MDE of Java Card Programs

A huge number of attributes can also be set or altered graphically on the model (e.g. type and cardinality of association links). Associations and attribute settings that violate the specifications and the restrictions of the Java Card metamodel are detected and prevented by the editor. For instance, if a *ParamType* or a *Field* instance is in array type, more than one dimension is not allowed for this defined array since multi-dimensional arrays are not allowed

in Java Card programs. Other examples for the violation check can be listed as follows: *Field* and *Method* instances can only be inserted into the instances of *JClass*, *Applet* and *OwnerPIN*. The initial value of an *OwnerPIN* instance's attribute for the remaining number of incorrect PIN trials should be equal to the value of the same instance's attribute for the maximum number of incorrect PIN entries.

Figure 4 also includes the Java Card instance model for our sample purse application introduced in section 2. The Java Card applet ("PurseApplet"), seen at the center of Figure 4, provides secure storage of the customer account information and includes appropriate methods for user authentication and e-money transfer (debit / credit). Other required Java Card components for the program (e.g. Purse, pursePIN) and their relations with the main applet class are also included in the model. Purse object, that will be stored in the smart card, includes balance information for a user. Instance fields and methods for the Purse object are encapsulated in proper compartments during the system modeling in the editor. Based on the received command APDUs, PurseApplet calls proper getter/setter methods of the Purse object and sends response APDUs to inform host application for the result of the operations. To keep simplicity, the complete definition of applet methods required for processing all command APDUs is not given in the current instance model. Also Java Card components needed for the exception mechanism are omitted in the model.

5. Modeling ZeitControl Basic Card Programs

ZeitControl Basic Card [14] (hereafter referred to as Basic Card), is another programmable smart card. It fully supports the smart card communication protocol defined in ISO/IEC 7816 standards [2]. The operating system of the Basic Card consists of a directory-based, PC-like file system. The built-in chip of the Basic Card has lower memory resources comparing to

other smart card technologies (such as Java Card). However, this also causes Basic Card to be cheaper than other cards.

The Basic Card programs are written in a special programming language called ZC-Basic Language [14], a dialect of the Basic language. That language is naturally not object oriented and each written program for Basic Cards is usually made up of a single code file in which the whole functionality needed for all card processing and host communication operations is defined as a set of Basic procedures. Similar to the Java Card's CAP conversion mechanism, a Basic Card program is converted to an image file (with .IMG extension) and uploaded to the Basic Card.

To provide MDE of Basic Card software in the defined MDA, again we need a metamodel for Basic Card programs and provide a development environment in which both visual modeling of Basic Card software and auto-generation of program codes are easily performed. In the following, we discuss these ingredients for the proposed methodology.

First of all, we derived the required metamodel from the structure of Basic Card programs. The metamodel describes the building blocks (data types, functions, subroutines, etc.) of a Basic Card program and relations between these blocks. Figure 5 depicts the Ecore encoded Basic Card metamodel. Entity attributes are not shown here again for the sake of simplicity.

Similar to the Java Card metamodel, the metamodel for Basic Card is considered as another PSMM in our MDE approach. Hence another set of model transformations were defined from the platform-independent smart card metamodel (introduced in Section 3) to this PSMM in order to obtain Basic Card platform-specific counterparts of the models conforming to the

platform-independent smart card metamodel. Related transformation mechanism will be later discussed in Section 6.

The *ZCardProgram* meta-entity (shown in the bottom-left of the Figure 5) is the representation of a Basic Card source program. A ZCardProgram is composed of program attributes, card initialization code, card operation procedures and references to external file(s) and definition file(s). The card initialization is the first block of code that is not contained in a procedure definition.

A *BasicMember* in the model represents any unit of a Basic Card program (e.g. a command, a function or a parameter). A Basic Card procedure definition can be a *Function*, a *Subroutine* or a *Command*. A function returns a value to the caller whereas a subroutine does not return a value but carries data through its arguments. A Command is defined like a subroutine however two ID bytes should be specified as well. These ID bytes represent the command that will be invoked later by the card.

Figure 5 approximately here

Figure 5: The Basic Card metamodel

Data types for variable and parameter declarations can be String, floating-point, single (4 byte single precision number), long, integer and byte in ZC-Basic Language. Hence, in the proposed metamodel, we define corresponding meta-entities (e.g. *BasicString* and *BasicFPoint*) and an enumeration for these types, called *DataTypes*. Constants for the declaration of command types are encapsulated in a *DefinitionFile*. Such definition files are included at the beginning of a Basic Card program. Also other source files, represented with

the *IncludeZCardProgram* entity in the metamodel, can be included by the main card program.

The *Persistence* meta-entity denotes the storage type and access rights of a data field in a Basic Card. Four flags (EEPROM, public, private and static) are defined in ZC-Basic Language for this purpose and any Persistence instance conforming to our metamodel stores values of each of these flags for a specific data field.

As expected, another visual modeling editor, based on the above metamodel of Basic Cards, has been developed for the Basic Card programmers (Figure 6). Figure 6 also includes the Basic Card instance model for our sample purse application introduced in Section 3. The elements in the model constitute a ZCardProgram which stores the related customer account information and includes appropriate Basic Card commands, subroutines and functions designed for electronic money transfer. The program has the same functionality with the previously discussed Purse applet. Instances for command definitions and required program attributes of the ZCardProgram are also included in the model.

Figure 6 approximately here

Figure 6: Modeling environment for the MDE of Basic Card Programs

6. Model Transformations between Platform Independent and Platform Specific Smart Card Metamodels

Sendall and Kozaczynski [23] describe model transformation as the heart and soul of model driven software development. Indeed, definition of metamodels is required but not sufficient for a complete MDE process. We have to define transformations between those metamodels

to obtain the main artifacts of the process: target models. We define transformations between the aforementioned platform independent and platform specific smart card metamodels and apply those transformations where generic smart card models conforming to the PIMM discussed in Section 3 are accepted as the source models and their platform specific counterparts (target models) conforming to the specifications of Java Card or Basic Card metamodels are automatically achieved.

In fact, benefits of the definition and application of the model transformations on generic smart card models are twofold. 1) We provide an operational semantics for the generic smart card models designed according to the PIMM since the models can be transformed into the models of card execution platforms such as JCF or Basic Card environment. 2) Developers model their card applications by just concentrating on the smart card domain without dealing with the specifications of various card platforms and later they obtain real implementations of their designed models by first the application of the model transformations and then code generation.

Entity mappings between our smart card PIMM and Java Card and Basic Card PSMMs pave the way for the definition and implementation of the model transformations that are applied on platform independent smart card model instances at runtime in order to obtain their counterparts in real smart card infrastructures. Mappings, which we determine between smart card PIMM and PSMM entities, are in *n-to-m* manner. That means *n* number of PIMM entities can be mapped to *m* number of Java Card PSMM entities (or *k* number of Basic Card PSMM entities). Table 3 lists some of these mappings. For instance, *Application* entity of the PIMM is mapped to *Applet* in Java Card PSMM and *DefinitionFile* in Basic Card PSMM. On

the other hand, *APDUOperation*, *CommandApdu* and *ResponseApdu* of PIMM are mapped to *Method* of Java Card PSMM and *Command* of Basic Card PSMM.

Table 3: Some of the entity mappings between smart card PIMM, Java Card PSMM and Basic Card PSMM

Smart card PIMM entity	**Java Card PSMM entity**	**Basic Card PSMM entity**
SCProject	JCProject	ZCardProgram
Application	Applet	DefinitionFile
APDUOperation CommandApdu ResponseApdu	Method	Command
PIN	OwnerPIN	Attribute
Constant	Field	Constant
Condition	CodeBlock	CodeBlock

After determination of the entity mappings between PIMM and above discussed target PSMMs, we need to provide model transformation rules which are applied at runtime on platform independent instances to generate platform specific counterparts of these instances. For that purpose, transformation rules should be formally defined and written according to a model transformation language. To this end, many languages are proposed (e.g. [24], [25], [26], [27]). In this study, we prefer to use ATL Transformation Language (ATL) to define required model transformations. ATL [27] is one of the well-known model transformation languages which is specified as both a metamodel and a textual concrete syntax. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. Besides,

ATL can define an additional model querying facility which enables to specify requests onto models [28]. ATL has a transformation engine and an integrated development environment (IDE) that can be used as a plug-in on Eclipse platform [29]. Finally, ATL can be used on the metamodels that conform to Eclipse Ecore meta-metamodel. Those features of ATL caused us to prefer ATL as the implementation language for the transformations between our platform independent and specific smart card models.

To give some flavor of the written transformations, some of the defined rules are discussed in here. For instance, Listing 1 includes a fragment of the ATL rule written for the transformation of platform independent smart card *application* instances into Java Card *applet*s. While transformation rules are being defined, source and target metamodels must be indicated in ATL code as shown in lines 2-3 of Listing 1. This information is also given in the properties of the created ATL project on Eclipse platform. As shown in Listing 1, "SmartCard.ecore" file is the input metamodel for transformation rules (denoted with “IN” keyword in line 4) and "Javacard.ecore" file is the output metamodel (denoted with “OUT” keyword in line 4).

```
01  module SmartCardtoJavaCard;
02  -- @path MM=/SmartCard/model/PIMModel/SmartCard.ecore
03  -- @path MM1=/SmartCard/model/PSMModels/JavaCard/Javacard.ecore
04  create OUT : MM1 from IN : MM;
05
06  rule Application2Applet{
07      from
08          appl : MM!Application
09      to
10          applet : MM1!Applet(
11              name  <- appl.name,
12              fields <- Sequence{appl.constants, appl.datas},
13              methods <- appl.getAssociations())
14     ...
15  }
```

Listing 1: A fragment from the ATL rule for the transformation of smart card application instances into Java Card applets.

"Application2Applet" rule will execute on a platform independent smart card model and for each Application instance (Listing 1, line 8), it will generate a corresponding Java Card Applet (line 10). In order to provide that generation, data and constants of the application are transformed into instance fields of an applet object by executing some inner rules (line 12). For example, during transformation of each data in the application into a field in the applet, "Data2Field" rule, shown in Listing 2, is executed. Every data attribute is converted into a field attribute (lines 6-11 of Listing 2). Default values for the unmapped attributes are also given (e.g. "static" and "final" is specific to the Java Card so just default values are given (lines 7 and 9 of Listing 2) instead of a transformation since there is no counterpart in the PIMM for these attributes).

```
01  rule Data2Field{
02      from
03          dts : MM!Data
04      to
05          flds : MM1!Field(
06              name <- dts.name,
07              static <- true,
08              comment <- dts.comment,
09              final <- false,
10              dataType <- dts.getDataType(),
11              IsArray <- dts.IsArray())
12  }
```

Listing 2: Rule for the transformation of application data into Java Card applet fields

It is worth noting that some helper rules are used during entity transformations. These helpers are the realization of the constraints to query the source models. The constraints in ATL are specified with using OCL [18], [19]. Same helper rules and constraint repetitions may be required both for other rules in the same target model transformation or other platform specific model transformations (e.g. for Basic Card). Hence this kind of rule decomposition makes the definitions easier. The helpers correspond to the constraint part of the related rules. There are two types of helpers in our transformations. The first type helpers are used to check

if the smart card model element is the part of the required pattern or not. The second type helpers are used to select the smart card elements for creating relations between target elements. For example, as a second type helper, the execution of "getDataType" helper rule in line 10 of "Data2Field" rule (in Listing 2) provides the determination of the appropriate Java Card data type for the source data. A fragment from that helper rule is given in Listing 3. Likewise, execution of "getAssociations" helper rule in line 13 of Listing 1 creates Java Card applet methods for APDU operations modeled in the platform independent smart card model while "IsArray" helper rule returns boolean true value when it encounters a string or a number array in the smart card application model.

```
01  helper context MM!Data def : getDataType() : String =
02        if (self.type = #number)then 'JCShort'
03        else if (self.type = #string)then 'JCByte' endif
04        else if (self.type = #boolean)then 'JCBoolean' endif
05        else if (self.type = #byte) then 'JCByte' endif
06        ...
07        endif;
```

Listing 3: A fragment from getDataType helper rule

In order to transform platform independent smart card software models into file-oriented Basic Card program models, another group of transformations are defined between the entities of PIMM and Basic Card PSMM and they are written in again by using ATL. For example, "SmartCard2BasicCard" rule given in Listing 4, creates a Basic Card program with all required components based on the transformation from a platform independent *SCProject* into a Basic Card *ZCardProgram*. Basic Card commands corresponding to APDUs, DefinitionFile(s) for each smart card Application instances and other remaining attributes are all determined and set by processing the source model (lines 12-16 in Listing 4).

```
01  module SmartCardtoBasicCard;
02  -- @path MM=/SmartCard/model/PIMModel/SmartCard.ecore
03  -- @path MM1=/SmartCard/model/PSMModels/BasicCard/BasicCard.ecore
04  create OUT : MM1 from IN : MM;
05
06  rule SmartCard2BasicCard{
07      from
08          smartCard : MM!SCProject
09      to
10          basicCard : MM1!ZCardProgram(
11              name <- smartCard.title,
12              commands <- Set{MM!APDUOperation.allInstances()->
13                   select(a|a.oclIsKindOf(MM ! CommandApdu) = false)},
14              attributes <- Set{MM!PIN.allInstances()->
15                 select(a|a.oclIsKindOf(MM ! APDUOperation) = false)},
16              defFile <- Set{MM!Application.allInstances()}
17        )
18  }
```

Listing 4: The ATL rule for the transformation of platform independent smart card software instances into Basic Card programs.

7. Automatic Code Generation from Platform Specific Smart Card Instance Models

Although both the graphical modeling and M2M transformations may facilitate the development of smart card software systems, it is not sufficient for real life implementations of such systems. Proposed software development methodologies should provide a step in order to assist developers for code generation. Within this perspective, various M2T transformations for each platform specific smart card framework are designed and implemented in this study.

We implement the related transformations by using the MOFScript [30]. MOFScript is a language specifically designed for the transformation of models into text files and it deals directly with metamodel descriptions as input. Also, it provides a tool as an Eclipse plug-in and hence MOFScript transformations can be written and directly executed inside the Eclipse environment. Taking into account all of these advantages, we chose MOFScript as the

implementation language for the M2T transformations that produce program codes for smart card applications.

The output smart card system models, achieved as the result of applying M2M transformations discussed in the previous section, now become the inputs of the defined M2T transformations. When a developer completes the visual modeling of a platform independent smart card system and executes above discussed M2M transformations to obtain that model's platform specific (Java Card or Basic Card) counterparts, the Ecore representation of the output model is stored in a file. The MOFScript engine applies our M2T transformations on this file and produces smart card program codes for the related application. It is worth noting that a developer may prefer to work just in the platform specific level and visually model a platform specific card application (e.g. a Java Card applet) by using the appropriate platform specific modeling editor(s) (introduced in Sections 4 and 5 of this paper) and finally that model can directly be accepted as an input for the M2T transformations to generate codes.

An excerpt from the prepared MOFScript transformation for Java Card programs is given in Figure 7. This text transformation uses the metamodel of the Java Card discussed in Section 4. The transformation, in here, reads a Java Card system model, determines each Applet instance and generates codes for each Applet class. Codes for each Java Card component are generated according to the rules and constraints of the Java Card programming language. These auto-generated program codes are ready to be both compiled and converted to the CAP format.

Transformation script, given in Figure 7, first controls whether the applet instance implements any interface and generates required Java Card codes. Later instance fields of the applet are determined and codes for their visibility, modifiability and type are generated based on the

input model. For each attribute, their type (object reference, array, JCByte, JCShort, etc.) are set. However amount of scripts required for that transformation is too big to be included here and hence it is not completely shown in Figure 7. In the next step, generation of the *install* method for the applet takes place. As its name already denotes, codes required for the installation of the software on the smart card are generated in here (see middle part of Figure 7). Scripts for APDU command detection and process selection are shown at the lower part of Figure 7. Reaction of the applet for each received command is determined in the *process* method of the applet. So, related transformation provides automatic generation of the *process* method's body. At the bottom part of Figure 7, code generation scripts for remaining methods of the applet are shown. However, full listing of scripts prepared for the generation of both *process* and other methods needs much more space and hence not discussed in this paper.

Figure 7 approximately here

Figure 7: An excerpt from the MOFScript transformation for Java Card programs

Figure 8 includes an excerpt from the auto-generated code of the example Purse applet previously discussed in Section 4. The transformation engine applies the transformation on the Java Card model of the e-purse application and Java class files belonging to each model component are achieved. For instance, excerpt in Figure 8 includes some of the auto-generated codes of Purse applet for hexadecimal APDU processing and packet control. Application of the transformation, shown in Figure 7, outputs the JavaCard applet body including applet instance fields, constructor, *install*, *process* and remaining APDU methods.

Figure 8 approximately here

Figure 8: An excerpt from the auto-generated code of the example Purse applet

In this study, another M2T transformation was defined and written in MOFScript for the generation of Basic Card programs according to the ZC-Basic Language. The serialized Basic Card models (in Ecore) are processed by the transformation engine and written rules are applied on these models to generate ZC-Basic codes. In Figure 9 (a), an excerpt from the MOFScript transformation for Basic Card is given.

Figure 9 (a) and (b) approximately here

Figure 9: (a) An excerpt from the MOFScript transformation for Basic Card programs. (b) An excerpt from the auto-generated ZC-Basic code of the example Purse program

In the above script (Figure 9 (a)), external definition files and card command procedures for a ZCardProgram are determined in a Basic Card model. ZC-Basic codes for each determined model element are generated. For instance, ZC-Basic code of our example Purse program is automatically generated when the model depicted in Figure 6 of this paper is given into this transformation. Figure 9 (b) includes an excerpt from the auto-generated ZC-Basic code of our example Purse program. Generated codes also include subroutines and functions but they are not shown in the figure to keep simplicity.

8. Evaluation

In order to evaluate applicability of our proposed MDA and practicality of the introduced methodology and modeling toolkits, we needed feedback from the smart card developers. For this purpose, the participation of the smart card software developers from the Kentkart Company has been provided. Kentkart[2] is one of the important IT companies in Turkey which manufactures smart card hardware and produces smart card based information systems. Main

[2] Kentkart Automatic Fare Collection & Vehicle Tracking Systems: http://www.kentkart.com/en (last access: December 2013)

expertise of the company lies within automatic fare collection, passenger information services and vehicle tracking systems. Currently, smart card based mass-transit systems of the Kentkart are being used in more than 15 cities of Turkey and some other locations in Europe and Middle East.

Software developers, willing to participate in this evaluation, were asked to test the modeling environment introduced in this study. We paid attention to gain feedback from a group of participants with varying experience from 2 years to 10 years on smart card software development. The assessments of the participants were retrieved by making an interview with each participant individually.

Graphical interface of the modeling tools was generally approved by the developers. The whole environment was found user friendly and easy-to-use. Almost all of the developers agreed that the design environment based on the derived metamodels fully supports related smart card programming constructs.

Capability of both modeling in general and without dealing with the specifications of different smart card platforms by employing the platform independent smart card metamodel and related modeling environment got mainly two different responses. All of the evaluators (developers in the company) encountered such a platform independent smart card software development environment for the first time. In fact, for many of them, it is the first time to use an IDE for smart card software development with visual modeling feature and automatic code generation. Most of the developers indicated that it is really a major benefit of our IDE not to be dependent on the specific smart card program constructs and provide a higher abstraction for modeling software. We have to note that those developers, who welcomed the use of

platform independent components of the proposed modeling environment, are mostly engineers with little experience on smart card development. However, that attraction contradicts a bit with the feedback gained from the other group of developers with substantial experience on software development for Java Card. Instead of the general smart card modeling environment, they favored the use of platform specific components and code generation. They advised to improve capabilities of the modeling environment by inserting the built-in support for CAP or IMG conversions for different types of Java or Basic smart cards. That needs the modeling environment to be specialized for every smart card type manufactured by different vendors. Inclusion of some template models (e.g. for simple purse applet or mass-transit card) inside the modeling environments was also suggested. Hence, a developer can open one of these application template models inside the related editor environment, visually make specializations for the desired system and then automatically obtain codes for the card software.

Two important modifications were made to the environment according to the common suggestions of the participants. Ability to include some code blocks during the visual modeling was strongly suggested. Sometimes card developers prefer to note an algorithm or just write a trivial (mostly not working) code segment for a model element at the design time. For this purpose, a meta-entity called *CodeBlock* was inserted with its associations with other meta-entities into the metamodels of both Java Card and Basic Card as previously discussed in Section 4 and 5 (shown in Figure 3 and 5). Related editor palettes include corresponding drag and drop elements and hence developers may add some notes or code fragments (e.g. as shown at the center of Figure 4) and associate them with the desired model instances. The content of a CodeBlock element can also be accepted as an annotation for a model element in some situations. During M2T transformation, content of a CodeBlock instance is directly

inserted into the generated code of the model instance associated with this CodeBlock. The content is inserted into the generated code as comment line(s).

Second modification is the automatic inclusion of the command detection and process selection structure for the Java Card applets. In a Java Card applet, once a command APDU is received, the type of the command is determined and the related method for processing the APDU packet is selected for execution. Traditionally, Java Card developers write codes for this command detection and selection structure in the process method of an Applet class. Developers advised to directly add template codes for this structure into the generated Java Card applet codes. Current MOFScript M2T transformation for Java Card programs supports automatic insertion of codes for this detection and selection structure. If each type of the command APDU and related processing methods is modeled for an application, auto-generated codes for this structure become complete and do not need any extra intervention of the developer. Considering the assessment of the evaluators, automatic inclusion of the command detection and process selection structure of the proposed environment is perhaps the best acknowledged feature.

Within the evaluation, we also took into account the code generation capability of the proposed MDA. Since it is less useful and in fact not appropriate to just measure the generated codes and give a quantitative result (such as a ratio between the number of lines of the generated code and lines of code pertaining to the full implementation), we preferred to determine which parts of a smart card application can or can not be produced completely just after the code generation. Table 4 summarizes the feedback on the assessment of the code generation for Java Card.

Table 4: Java Card code components which are fully or partially generated via automatic code generation

Code Component	**Assessment of the automatically generated code**
Fundamental Java Card methods (e.g. "process", "install" and PIN validation)	Methods can be fully created. Related codes can be executed on smart cards directly without any addition.
User defined methods	All method signatures are generated. However methods need to be modified/completed before execution on the smart card.
Constant data types	All of them can be fully generated.
Attributes and Method fields	All of them (both system and user-defined ones and arrays) can be fully generated.
Exception mechanism	Codes for handling two types of exceptions: "Wrong Length" exception and "PIN Verification Required" exception, defined in Java Card Platform Specification [31], can be fully generated. Handlers for remaining exceptions need further intervention.

Critical code components (listed at the left column of Table 4) for Java Card applications were determined first and developers were asked to examine generated codes for these components. Assessment result for each component is listed next to the related component in Table 4. Likewise, code components for a Basic Card application were determined and generated codes for these components were evaluated. Results for this evaluation are listed in Table 5.

Table 5: Basic Card code components which are fully or partially generated via automatic code generation

Code Component	**Assessment of the automatically generated code**
"Command" procedures (for APDU packet exchange)	Procedure signatures, parameters, return values and condition statements can be fully created.
User defined "subroutine"s and "function"s	Only signatures can be generated.
Constant data types	All of them can be fully generated.
Variable data types	All of them can be fully generated.

Finally, it is worth reporting the effort needed for the development of the proposed MDA with its supporting components and tools. Despite using appropriate MDE tools ([17], [19], [20], [21], [29], [30]) and techniques ([8], [32]), the whole development was accomplished over 12 month period. Three metamodels covering more than 60 entities with inner and inter-relations were required to be constructed inside the Eclipse platform. More than 400 lines of code (LOC) were produced as the result of the metamodel creation. Related process provided the production of graphical modeling editors and toolkits based on the Eclipse GMF. Further, 17 ATL rules for Java Card platform and 7 ATL rules for Basic Card platform were written for the implementation of M2M transformations. Approximately 550 LOC were written for these M2M transformations. Automatic code generation from the platform-specific smart card models needed the implementation of M2T transformations as MOFScript rules with more than 1000 LOC.

9. Related Work

The challenge of smart card software development naturally causes some researchers to study on new approaches and define new methodologies for easy development. Since JCF is the most available and open framework, related work almost covers just the development of Java Cards. For instance, the independent certification mechanism introduced in [9] includes a generator and checker to develop Java Card applets with high assurance. The checker takes an applet specification, generated code and a proof, and returns an answer, depending on whether the proof is the valid evidence of the correctness of the code with respect to the specification. However, only the shallow embedding of a subset of Java Card specification is considered in the study. The formal specification of the approach is discussed in [12]. An approach to correctness, in which a generator generates checkable proofs from the transformations that it performs, is proposed in the paper. The approach is exemplified with the description of a generator of Java Card applets. Our study differs in supporting the checking of models via metamodels and generation of codes from the system models.

Bonnet et al. [33] propose a framework for personalizing on-card software relying on the MDE and software product lines. The adaptation of this framework to the context of smart card configuration is further detailed in [34]. Since configuring a smart card is a multi-level process involving actors such as customers, marketers or engineers, the customization level ranges from clients (e.g. a bank) to individual card holders. The architecture of the proposed model driven software product line consists of modeling the core software artifacts that define product families and marking these models with variability-specific annotations [33]. Within this context, Bonnet et al.'s study deals with the layered configuration process for smart cards and mostly involves card production issues while our study considers the MDD of software regardless of the card configuration.

SecureMDD, introduced in [35], is a model driven software development method which intends to facilitate the development of security-critical applications that are based on the cryptographic protocols. The applications are first modeled using a UML profile which is tailored to model security-relevant aspects and extend UML activity diagrams. As the result of a series of some transformations, the implementation of the system is realized. However, the way of implementing model transformation and code generation is not included in [35]. Also, specific application of the proposed methodology is investigated again only for the Java Card code generation. Furthermore, instead of concentrating on providing an MDD for smart card development, main aim of the work is to formally prove the correctness and security of the generated code for security-critical distributed applications in general. Only Java Card platform is chosen for the exemplification purposes.

A methodology based on the B Method [36] is introduced in [11] for the development of Java Card applications. The B method is used to specify the functionality of the card-side components. Platform-specific code can then be automatically obtained by the refinement and code generation process. This work is based on a previous study [10] and aims to provide automated support to generate Java Card methods from B specifications. In Tatibouet et al.'s work [10], the generated code needs to be manually modified to combine the communication and code aspects particular to the Java Card platform. Gomes et al.'s effort [11] is important with bridging the above mentioned gap by proposing to generate Java Card platform-specific codes automatically during the introduced methodology. However, their study just covers ideas. A tool, which implements all the identified steps, needs to be developed as already admitted by the authors.

Similar to works in [9], [10] and [11], our previous work [37] also deals with the automatic generation of Java Card applications. Modeling smart card software according to Java Card specifications and code generation from the designed models are guided with a graphical tool. However, neither the assessment of the proposed methodology nor platform independent modeling of the smart card software is considered in that study.

The work introduced in [38] considers platform independent and platform specific modeling of smart card applications according to MDA. However, instead of constructing metamodels, and creating instance models conforming to those metamodels, authors propose the construction of system models just based on UML class diagrams. In fact, every platform independent model (PIM) is a class model (only covering instance fields) of a system intended to be built. PIMs only conform to UML and do not include any specifications for the smart card domain. Besides, every platform specific model (PSM) is just the improved version of a PIM in which previous classes now encapsulate the signatures of some new methods. Hence, considering any smart card execution platform, the generated PSM is in the form of a general and ordinary class diagram so it is still a PIM. We can conclude that Nikseresht and Ziarati [38] propose a framework for the development of only file-oriented smart card applications.

Taking into account all of the above discussed related work, we believe that our MDA-based methodology contributes to the noteworthy studies of those researchers by enabling both abstract and generic smart card modeling based on a PIMM, supporting more than Java Card framework in the platform specific level and also providing convenient modeling tools that are required for the application of a model driven smart card software development.

Finally, it is worth indicating that there exist various successful MDD/MDE applications on different domains. For instance, Jimenez et al. [39] introduce a new model-driven methodology with its supporting domain-specific language for home automation system design. Heijstek and Chaudron [40] discuss the impact of MDE for the implementation of a system for supporting sales of mortgages in a large financial institution. Fister Jr. et al. [41] discuss the MDD of software required for the measurement of time in sporting competitions and present a domain-specific language for this purpose. MDE practices on three different domains, imaging system manufacturing, car manufacturing and telecommunication respectively, are reported in [42]. Kos et al. [43] introduce a domain-specific language, called Sequencer, for modeling data acquisition and measurement process control and discuss the application of Sequencer on the automotive industry. Moreover, the most recent studies introduce the use of MDE for the development of air traffic control systems [44], control command software in nuclear power plants [45] and model extraction tool for healthcare data annotation [46]. Those examples may signify the expectation of similar achievements in the smart card domain as the result of fruitful MDA application.

10. Conclusion

An MDA for the development of smart card software was designed and implemented in this study. Metamodels for smart card systems both in platform independent and platform specific levels were derived and M2M transformations were defined and applied for the instances of these metamodels residing on different abstraction levels. Furthermore, M2T transformations based on the introduced smart card PSMMs were constructed for the automatic generation of the card software. The engineering methodology based on our proposed MDA is supported with integrated development environments in which developers can easily model smart card

software conforming to the specifications and restrictions of the related smart card frameworks and finally obtain auto-generated, ready-to-compile program codes.

Based on the feedback gained from the smart card software developers, we believe that the application of the method and use of the modeling environments provide easy and efficient development of resource-restricted smart card software and save the developers from the tedious and error-prone work. Within this context, main advantages of the approach can be listed as follows: 1) Easy modeling that enables automatic preparation of smart card software components. For instance, incoming and outgoing hexadecimal data packages for smart cards can be visually designed instead of hard coding. Hence, there is no need to prepare byte-by-byte message preparation. 2) Graphical design and automatic code generation of the process methods for smart card applications. 3) An integrated development environment for rapid code generation. That is especially welcomed by card programmers during our evaluation.

Again based on the feedback gained as the result of the developers' assessments, we can state that the proposed MDA and supporting modeling tools have the potential of fulfilling the requirements and/or expectations of the smart card software developers within a wide range. Specifically, we determined that the unexperienced developers tend to start from scratch and hence use platform independent card modeling environment first and then apply M2M transformations to work for the details of the specific card platforms (such as Java Card or Basic Card). However, experienced developers mostly prefer employing the platform specific modeling and M2T transformations in order to directly achieve codes for the dedicated smart card frameworks as the main artifacts.

Finally, we can also add that the introduced PIMM and defined transformation in this study may pave the way for the derivation of a Domain-specific Language (DSL), especially a Domain-specific Modeling Language (DSML) for smart card software. DSLs ([32], [47], [48], [49]) have notations and constructs tailored toward a particular application domain (e.g. smart cards). The end-users of DSLs have the knowledge from the observed problem domain [50], but usually they have little programming experience. Domain-specific modeling languages (DSMLs) further raise the abstraction level, expressiveness and ease of use, since models are specified in a visual manner and they represent the main artifacts instead of software codes [51]. The development of a DSML is usually driven with the language model definition [52]. That is, concepts and abstractions from the domain need to be defined to reflect the target domain (language model). Then, relations between language concepts need to be defined. Both form an abstract syntax of modeling language and usually, language model is defined with a metamodel. Within this context, the PIMM we introduced in this paper can naturally enable us to achieve the abstract syntax of a smart card DSML. Furthermore, the GMF-based representations of the meta-entities discussed here may provide a visual concrete syntax for the desired DSML. It is not sufficient to complete a DSML definition by only specifying the notions and their representations. The complete DSML definition requires the language semantics. One way of fulfilling this requirement is to derive an operational semantics in which the semantics of the language concepts is provided in terms of other concepts whose meaning is already established. In our case, we can achieve the semantics over the model transformations between the smart card PIMM and PSMMs of the dedicated smart card environments. However, the challenges for specifying DSML semantics may still remain in this approach. As discussed in [53], restrictive well-formedness constraints may prevent the construction of valid models. Even if such models are constructed, that does not mean they generate acceptable behaviors. Further, composition, verification and

reusability of semantics may also be challenging. Finally, the semantics definition of a DSML should be clear and comprehensible for all of its users. But, it would be probably difficult to support that comprehension at the same level both for the language designers and the domain experts who are supposed to benefit from the DSML in question [53].

As the future work, we can consider the enrichment of current MDA's platform-specific support; such that smart card software, designed according to the PIM specifications introduced in this study, can be also implemented and executed in other smart card platforms not covered in here (e.g. Microsoft .Net based smart card framework of Gemalto Inc. [54]). Similar to the generation of platform specific modeling environments for Java Card or Basic Card, we first need to derive the metamodel of such smart card frameworks. Upon completion of the metamodel(s) creation, it is straightforward to build up M2M transformations from our smart card PIMM to those platform's models and finally define M2T transformations to gather smart card executables for those platforms as discussed in this paper.

Acknowledgments

We would like to thank software developers from Kentkart Automatic Fare Collection and Vehicle Tracking Systems for their cooperation and valuable feedbacks.

References

[1] Rankl W., Effing W. Smart Card Handbook. 4th ed. John Wiley & Sons: 2010.

[2] ISO/IEC 7816 Standards. ISO/IEC 7816 Standards family for Identification cards - Integrated circuit cards. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45144 (last access: December 2013).

[3] Erdur RC, Kardas G. Personalized Access to Semantic Web Agents Using Smart Cards. Lecture Notes in Computer Science 2005; 3648: 1110-9.

[4] Kardas G, Tunali ET. Design and Implementation of a Smart Card Based Healthcare Information System. Computer Methods and Programs in Biomedicine 2006; 81(1): 66-78.

[5] Kardas G, Celikel E. A Smart Card Mediated Mobile Platform for Secure E-Mail Communication. In: 4th International Conference on Information Technology: New Generations (ITNG 2007), Las Vegas, USA: IEEE Computer Society Press 2007: p. 925-6.

[6] Schmidt DC. Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer 2006; 39(2): 25-31.

[7] Object Management Group. Model Driven Architecture. http://www.omg.org/mda/ (last access: December 2013).

[8] Frankel DS. Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley; 2003.

[9] Coglio A. Code generation for high-assurance Java Card applets. In: 3rd NSA Conference on High Confidence Software and Systems 2003: p. 85-93.

[10] Tatibouet B, Requet A, Voisinet JC, Hammad A. Java Card code generation from B specifications. Lecture Notes in Computer Science 2003; 2885: 306-18.

[11] Gomes BEG, Moreira AM, Deharbe D. Developing Java Card Applications with B. Electronic Notes in Theoretical Computer Science 2007; 184: 81-96.

[12] Coglio A, Green C. A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Applets. Lecture Notes in Computer Science 2008; 4171: 57-63.

[13] Sun Microsystems. Java Card Technology. http://java.sun.com/javacard/ (last access: December 2013).

[14] ZeitControl Card Systems GmbH. Basic Card. http://www.basiccard.com/ (last access: December 2013).

[15] Hansmann U, Nicklous MS, Schack T, Seliger F. Smart Card Application Development using Java, Springer: 2000.

[16] Saritas HB, Kardas G. Model Driven Development of Smartcard Software. In: 3rd Turkish Software Architecture Conference (UYMK 2010), Ankara, Turkey, 2010: p. 34-44 (in Turkish).

[17] Eclipse Community. Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/ (last access: December 2013).

[18] Warmer J., Kleppe A. The Object Constraint Language: Getting Your Models Ready for MDA. 2nd ed. Addison-Wesley Professional: 2003.

[19] Object Management Group. Object Constraint Language (OCL). http://www.omg.org/spec/OCL/2.3.1/ (last access: December 2013).

[20] Eclipse Community. Eclipse Platform. http://www.eclipse.org/ (last access: December 2013).

[21] Eclipse Community. Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/ (last access: December 2013).

[22] Chen Z. Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Massachusetts, USA: Addison-Wesley; 2000.

[23] Sendall S, Kozaczynski, W. Model transformation - the heart and soul of model driven software development. IEEE Software 2003; 20: 42–5.

[24] Duddy K, Gerber A, Lawley M, Raymond K, Steel J. Model Transformation: A declarative, reusable patterns approach. In: 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), Brisbane, Queensland, Australia; 2003: 174-85.

[25] Kalnins A, Barzdins J, Celms E. Model Transformation Language MOLA. Lecture Notes in Computer Science 2005; 3599: 62-76.

[26] Agrawal A, Karsai G, Neema S, Shi F, Vizhanyo A. The design of a language for model transformation. Software and Systems Modeling 2006; 5(3): 261-88.

[27] Jouault F, Allilaire F, Bezivin J, Kurtev I. ATL: A model transformation tool. Science of Computer Programming 2008; 72(1-2): 31-9.

[28] ATLAS Group. ATL User Manual. http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf (last access: December 2013).

[29] Eclipse Community. ATL Model Transformation Language and Toolkit. http://www.eclipse.org/atl/ (last access: December 2013).

[30] Oldevik J, Neple T, Gronmo R, Aagedal J, Berre AJ. Toward Standardised Model to Text Transformations. Lecture Notes in Computer Science 2005; 3748: 239-53.

[31] Oracle Corporation. Java Card 3.0.1 Platform Specification. http://java.sun.com/javacard/3.0.1/specs.jsp (last access: December 2013).

[32] Mernik M, Heering J, Sloane A. When and how to develop domain-specific languages. ACM Computing Surveys 2005; 37(4): 316-44.

[33] Bonnet S, Potonniee O, Marvie R, Geib, J-M. A Model-Driven Approach for Smart Card Configuration. Lecture Notes in Computer Science 2004; 3286: 416-35.

[34] Bonnet S, Marvie R, Geib J-M. Putting Concern-Oriented Modeling into Practice. In: 2nd Nordic Workshop on UML, Modeling, Methods and Tools, Turku, Finland; 2004.

[35] Moebius N, Stenzel K, Grandy H, Reif W. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In: 4th International Conference on Availability, Reliability and Security, IEEE Computer Society Press 2009; p. 841-6.

[36] Abrial J-R. The B-Book: Assigning Programs to Meanings. Cambridge University Press: 1996.

[37] Saritas HB, Kardas G. Model Driven Development of Java Card Software. Turkish Informatics Foundation Journal of Computer Science and Engineering 2011; 4: 19-28 (in Turkish).

[38] Nikseresht A, Ziarati K. MDA Based Framework for the Development of Smart Card Based Application. In: 2011 International MultiConference of Engineers and Computer Scientist, Hong Kong; 2011; p. 1-6.

[39] Jimenez M., Rosique F., Sanchez P., Alvarez B., Iborra A. Habitation: A Domain Specific Language for Home Automation. IEEE Software 2009; 26(4): 30-38.

[40] Heijstek W., Chaudron M.R.V. Empirical investigations of model size, complexity and effort in a large scale, distributed model driven development process. In: 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009), IEEE Computer Society Press 2009, p. 113-20.

[41] Fister Jr. I., Fister I., Mernik M., Brest J. Design and implementation of domain-specific language easytime. Computer Languages, Systems & Structures 2011; 37: 151-167.

[42] Hutchinson J., Rouncefield M., Whittle J. Model-driven engineering practices in industry. In: 33rd International Conference on Software Engineering (ICSE 2011), ACM Press 2011; p. 633-42.

[43] Kos T., Kosar T., Mernik M. Development of data acquisition systems by using a domain-specific modeling language. Computers in Industry 2012; 63(3): 181-192.

[44] Carrozza G., Faella M., Fucci F., Pietrantuono R., Russo S. Engineering Air Traffic Control Systems with a Model-Driven Approach. IEEE Software 2013; 30(3): 42-48.

[45] Ceret E., Calvary G., Dupuy-Chessa S. Flexibility in MDE for scaling up from simple applications to real case studies: illustration on a Nuclear Power Plant. In: 25ème conférence francophone sur l'Interaction Homme-Machine (IHM 2013) 2013, p. 1-10.

[46] Van Gorp P., Vanderfeesten I., Dalinghaus W., Mengerink J., van der Sanden B., Kubben P. Towards Generic MDE Support for Extracting Purpose-Specific Healthcare Models from Annotated, Unstructured Texts. Lecture Notes in Computer Science 2013; 7789: 213-221.

[47] van Deursen A, Klint P, Visser J. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices 2000; 35(6): 26-36.

[48] Mernik M, Zumer V. Incremental programming language development. Computer Languages, Systems & Structures 2005; 31: 1-16.

[49] Fowler M. Domain-specific Languages. Addison-Wesley Professional: 2011.

[50] Sprinkle J, Mernik M, Tolvanen J-P, Spinellis D. Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer?. IEEE Software 2009; 26(4): 15-8.

[51] Gray J, Tolvanen J-P, Kelly S, Gokhale A, Neema S, Sprinkle J. Domain-Specific Modeling. In Fishwick PA, editor. Handbook of Dynamic System Modeling: CRC Press; 2007, p. 1-7.

[52] Strembeck M, Zdun U. An approach for the systematic development of domain-specific languages. Software: Practice and Experience 2009; 39(15): 1253-92.

[53] Bryant B.R., Gray J., Mernik M., Clarke P.J., France R.B., Karsai G. Challenges and directions in formalizing the semantics of modeling languages. Computer Science and Information Systems 2011; 8(2): 225-253.

[54] Gemalto Inc. Gemalto .NET Smart Card Framework. http://www.gemalto.com/products/dotnet_card/dotnet_framework.html (last access: December 2013).

Hidayet Burak Saritas received his B.Sc in Mathematics (Computer Science division) and M.Sc in Information Technologies from Ege University in 2007 and 2011 respectively. He is currently working as a senior software developer at Kentkart Ege Electronics Company. His main research interests are model-driven development, smartcards and embedded systems.

Geylani Kardas received his B.Sc. in computer engineering and both M.Sc., and Ph.D. degrees in information technologies from Ege University in 2001, 2003 and 2008 respectively. He is currently an assistant professor at Ege University, International Computer Institute. His research interests include model-driven software development, domain-specific (modeling) languages, agent-oriented software engineering and smartcard systems. He is a member of the ACM.