Developing Java Card Applications with B

Article *in* Electronic Notes in Theoretical Computer Science · July 2007

DOI: 10.1016/j.entcs.2007.03.016 · Source: DBLP

3 authors, including:

Anamaria Martins Moreira
Federal University of Rio de Janeiro
62 PUBLICATIONS 222 CITATIONS
SEE PROFILE

David Déharbe
ClearSy System Engineering
114 PUBLICATIONS 940 CITATIONS
SEE PROFILE

Developing Java Card Applications with B[*]

Bruno Emerson Gurgel Gomes[1], **Anamaria Martins Moreira**[1], **David Déharbe**[1]

[1]Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN)
Lagoa Nova
59072-970 – Natal – RN – Brazil

{bruno, anamaria, david}@consiste.dimap.ufrn.br

Abstract. *This work proposes a methodology for the rigorous development of Java Card smart card applications, using the B Method. Its main feature is to abstract the particularities of Java Card and smart card aware applications from the specifier as much as possible. In the proposed approach, the specification of the aplication logic needs not be preoccupied with the specific aspects of the Java Card platform (in particular, communication between the card acceptance device and the smart card itself). Platform-specific code can then be automatically generated during the refinement and code generation process. An interesting side-effect of this approach is that the specification may be reused with any other platform of implementation.*

1. Introduction

The smart card support is a component of the IT infrastructure in a growing number of sectors [11]: banking, mobile and non-mobile communications, ID/access, leisure, retail and loyalty, transport, healthcare, government, multimedia, etc. Most of the applications require a high-degree of reliability, and make smart card aware software design a suitable application for formal methods.

Java Card [4] is one of the leading technologies in this sector as it provides significant features: multiple applications, portability, compatibility with a leading programming language technology (Java). The strategic importance of this market is a strong motivation to address the problem of providing rigorous software development processes for smart-card aware applications based on the Java Card technology.

The B methodology [1] is a good candidate for such process. Based on the experience gathered from the development of formal specification languages such as VDM and Z, B has been one of the foremost formal methods with a strong industrial support and has been applied in the development of safety-critical applications.

In this paper, we propose a specialization of the B methodology that aims at improving the productivity in the development of Java Card software. Previous work [16] showed the possibility to automatically generate Java Card code from B modules. However, this work is limited to the translation of the language aspects and ignores some important aspects of the Java Card platform, such as the *communication* between the host application and the applet running on the smart card, leaving its specification and implementation as an additional burden to the designer. The goal of the research presented in

[*]The work presented in this paper has been partially financed by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), a Brazilian governmental agency for the development of science and technology.

this paper is to provide B design guidelines specific for the Java Card framework which make it possible to automate part of the specification and implementation needed to build the communication protocol between the card applet and the host application.

The paper is organized as follows. The fundaments are presented in Section 2. (overviewing smart cards and Java Card) and 3. (introducing the B method). The core contribution of the paper is in Section 4., where the application of the B methodology to design Java Card components is presented. Section 5. concludes the paper with related work and some final remarks.

2. Smart Cards and Java Card

A smart card is a plastic card that looks like a common magnetic-stripe card, but that has embedded in it an integrated circuit with a microprocessor and memory. This kind of card offers several advantages compared with magnetic-stripe cards [10]. The most important advantage is the security in data processing and storage provided by the smart card environment composed by operating system, microprocessor and applications. In application level, personal identification numbers (PIN) and cryptographic algorithms can be used to improve the security. Other advantages are storage capacity, millions of times greater than the capacity of magnetic-stripe cards; and the remote database query independence, once every data the card needs is found in its memories or is supplied by an external application.

Many applications can benefit from the use of smart cards. Telecommunication, financial and transportation industries and the health care sector are good examples. The secure transaction mechanisms and the elements for secure user identification and data storage, besides the mobility of the cards, make them an ideal platform for these applications.

Java Card technology consists in a Java language subset. It allows memory-constrained devices, like smart cards, to run applications in a secure and inter-operable way. Security is obtained through Java elements, like its secure execution environment, which controls, for instance, the level of access to all methods and attributes; and the applet separation by a resource named applet firewall [4]. Inter-operability is the characteristic that allows the execution of a Java Card application in any smart card that follows the Java Card specifications, independently of hardware and software manufacturers, without or with few code modifications.

The use of this technology brings many improvements for the developer of smart card applications. The ease of programming in Java, that abstracts the low level details of the smart card system; and Java development tools (like IDEs, simulators and emulators) allow a rapid application build, test and installation, reducing the time and the cost of software production. Moreover, other benefits are the possibility of multiple applications to coexist in a same card and the ample compatibility with smart card international standards, like ISO 7816.

2.1. Smart Card system and communication model

A smart card system is composed of hardware and software components. These components are: Support software, software for communication with the card acceptance device (CAD), the CAD itself and the smart cards and their applications.

User-CAD communication software (host application) This software is responsible for the communication between an external application, called “host application”, and the one inside the card. It sends commands for the smart card application and receives the responses to these commands. This software can be included in a desktop computer, in a cell phone or in a security subsystem.

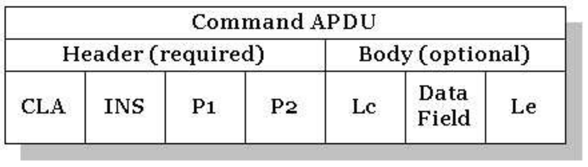
Card Acceptance Device (CAD) A CAD is the device localized between the host application and the smart card. It supplies power to the card and is the means of communication between the host application and the application inside the card. A CAD can be connected to a desktop or a terminal, such as an electronic payment terminal.

Smart Cards and their applications Inside the card’s memories are installed the applications. This can be done when the card is being manufactured, installing applications in its ROM memory, or later, installing the applications in the card’s non-volatile and writable memory. Languages like C, the assembly language of the card and Java Card can be used to develop these applications. Today, Java Card is supported in more than 95% [14] of the cards and is considered the best choice when productivity and security are the main requirements.

Support software This kind of software provides services to a smart card application. For instance, we could have an application that allows the applet to access a credit card operator service in a secure way.

The communication between these environment items is performed through a half-duplexed protocol called *Application Protocol Data Unit (APDU)*. An APDU message has the form of a data package exchanged between the host application and the application in the card in a master-slave architecture. The host application sends commands to the card application, that, in turn, sends back a response. The *command* and the *response APDU* are the two protocol structures used to send these messages. In this protocol, a command APDU is always paired with a *response APDU* [4].

A *command APDU* has the structure shown in figure 1 below:

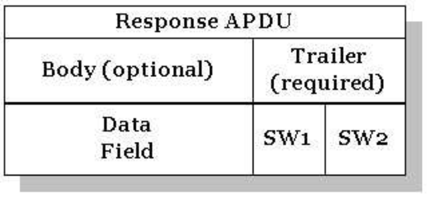


Command APDU						
Header (required)				Body (optional)		
CLA	INS	P1	P2	Lc	Data Field	Le

Figure 1. Command APDU structure. SOURCE: [9]

The command header is obligatory and is composed by four 1 byte fieds: *CLA*, *INS*, *P1* and *P2*. The *CLA* field identifies a class of *command* and *response APDU* . The *INS* field corresponds to a instruction inside a *CLA*. These instructions can be, for instance, method calls. *P1* and *P2* are parameters that can be used to supply some additional information to the *INS* instruction. The command body is required only for extra data sending or receiving. The data is sent in the *DATA* field and has its length specified in *Lc*. If a response with data is expected, its length has to be informed in the *Le* field. The Lc and Le fields have 1 byte of length.

A *response APDU* has a simple structure, which is shown in the figure 2 below:



Response APDU		
Body (optional)	Trailer (required)	
Data Field	SW1	SW2

Figure 2. Response APDU structure. SOURCE: [9]

The response is formed by the optional body, that contains the *Data* field, with the data returned to the host application and the trailer, which contains the fields *SW1* and *SW2* that together inform the *command APDU* processing status.

2.2. Java Card applets

A Java Card applet is a class written in the Java Card subset of the Java language that inherits the *javacard.framework.Applet* class. This class is a blueprint that defines some variables and methods of an applet [4]. It makes, for instance, the implementation of the *install* and *process* methods obligatory. The install method creates the applet by invoking its constructor method and registers it in the *Java Card Runtime Environment (JCRE)*, by invoking the register method. The *process* method receives the APDU messages of the host application, does the initial processing of these messages, and invokes a method, passing to it the APDU object as a parameter.

Example 1 *A short applet example summarizing the affirmations made above is:*

```
import javacard.framework.*;
public class Transport extends Applet {
    //The current amount of credit
    private short balance;
    //The INS code of addCredit method
    public static final ADD_CREDIT = 0x01;

    /** Constructor method */
    private Transport(byte[] bArray, short bOffset, byte bLength) {
        balance = 0;
        register();
    }
    /** Invoked by the JCRE, install is the applet entry point. It
        creates an applet instance and registers it in the
        JCRE through the invocation of the applet constructor method. */
    public static void install(byte[] bArray, short bOffset, byte bLength) {
        new Transport(bArray, bOffset, bLength);
    }
    /** process receives an APDU object and selects the instruction
        specified in its INS filed. */
    public void process (APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        switch (buffer[ISO7816.OFFSET_INS]) {
            case ADD_CREDIT:
                addCredit(apdu);
    }
    /** The method addCredit adds some data to the balance attribute. */
    public final void addCredit(APDU apdu) {
        byte[] buffer = apdu.getBuffer();
        byte bytesRead = (byte) apdu.setIncomingAndReceive();
        balance = (short) (balance + buffer[ISO7816.OFFSET_CDATA]);
    }
}
```

3. Software development with B

The B method for software development [17, 1] is based on the B *Abstract Machine Notation* (AMN) and the use of formally proved refinements up to a specification sufficiently concrete that programming code can be automatically generated from it.

Its mathematical basis consists of first order logic, integer arithmetic and set theory, and its basic constructs concerning them are very similar to those of the Z notation [12]. Its structuring constructs are however stricter and more closely related to imperative modular programming language constructs, with the intention of being more easily understood and used outside the academic world. Also, its more restrictive constructs simplify the job of support tools. Industrial tools for the development of B based projects have been available for a while now [5, 2], with specification and verification support as well as some project management tasks and support for team work. Its modular structure and characteristics make it adequate for the specification of Application Programming Interfaces (APIs) or other software components.

3.1. The B methodology

A B specification is structured in modules which are labeled according to their abstraction level: *MACHINE*, *REFINEMENT* or *IMPLEMENTATION*, from the most abstract to the most concrete. The development process starts with one or more MACHINEs, which may be refined into REFINEMENTs (optional) and then into IMPLEMENTATIONs. The original abstract MACHINEs are to be proved consistent with respect to some specified properties (particularly, the INVARIANT of each MACHINE) and then, each refinement step has to be proved correct with respect to the corresponding machine. The IMPLEMENTATIONs are then checked for compliance with the code generator for a particular language and, if it is the case, programming code may be generated. Assuming the correctness of the code generator, the generated code can be guaranteed to satisfy the stated properties of the abstract specification (the MACHINEs).

3.2. The B notation

Although we concentrate our introduction to the B notation on the more abstract specification (i.e. MACHINE), similar comments apply to the remaining levels. A B module contains two main parts: a state space definition and the available operations. It may additionally contain auxiliary clauses in many forms (parameters, constants, assertions), but those, essentially for practical purposes (i.e. to promote modularity, reuse, etc.), and do not extend the expressive power of the notation. In the remainder, we will restrict our discussion to the core clauses of the module specification.

The specification of the state components appears in the VARIABLES and INVARIANT clauses. The former enumerates the state components, and the latter defines restrictions on the possible values they can take. Essentially, if V denotes the state variables of a machine, the invariant is a predicate on V. Let us denote INV such invariant predicate. All verifications carried out throughout the development process have the intention of checking that no invalid state will ever be reached as long as the operations of the machine are used as specified.

For the specification of the initialisation as well as the operations, B offers a set of so-called *substitutions*. These are “imperative-like” constructions with translation rules that define their semantics as the effect they have on the values of any (global or local) variables to which they are applied. The semantics of the substitutions is defined by the *substitution calculus*, a set of rules stating how the different substitution forms rewrite to formulas in first-order logic. Let S denote a substitution, E an expression, then $[S]E$ denotes the result of applying S to E.

The basic substitution, denoted $v := E(V)$, where E is an expression on variables V, states that, when the operation completes, the value of variable v is $E(V)$, where the values of the variables appearing in this expression are taken when the operation initiates. For instance, an operation that would incrementing a counter variable cnt would be specified as $cnt := cnt + 1$. Indeed, the basic substitution is very similar to the side-effect free assignment construct found in imperative programming languages. Applying such substitution to an expression consists in substituting the target variable v with the source expression. For instance, $[cnt := cnt + 1]cnt \geq 0 = cnt + 1 \geq 0$. The B notation provides conditional, non-deterministic, parallel, and other substitution constructs.

One very particular substitution is the PRE-THEN-END, which can be used to specify any pre-condition that the definition of the operation assumes in order to "work properly". For instance, a partial operation that increments our counter variable only up to a certain value max would be specified as $cnt := PRE\ cnt < max\ THEN\ cnt := cnt + 1\ END$. This construct offers the full expressive power of first-order logic to specify the domain of an operation. It is therefore very useful to specify the bounds of application of an operation, within which one expects that the machine will not reach any invalid state.

Example 2 *A very simple example of a B machine is:*

```
MACHINE Transport
VARIABLES
    balance
INVARIANT
    balance : NAT
INITIALISATION
    balance := 0
OPERATIONS
    addCredit (cr) =
      PRE
        cr : NAT &
        cr > 0 &
        balance + cr <= MAX_DATA
      THEN
        balance := balance + cr
      END
END
```

This example was extracted and simplified from the *Transport* machine, the main machine of our sample transport application (Section 4.2.5.). In this piece of code, we can see the machine name definition in the *MACHINE* clause. The state variable, named *balance*, is typed in the *INVARIANT* clause and initialized in the *INITIALISATION* clause. An operation, named *addCredit*, is used to add some positive natural value to the *balance* variable.

3.3. Proof obligations

To guarantee the correctness of a B module, proof obligations must be generated from the initialization clause and the operations definition, establishing that:

1. the initialization actions take the machine into a valid state, i.e. the initialization substitution S establishes the invariant: $[S]INV$.
2. the machine will not be taken from a valid state into an invalid one when any of the machine's operations is executed as long as the user provided parameters and the machine variables are such that the pre-condition PRE for application of the substitution S corresponding to this operation evaluates to true: $PRE \wedge INV \Rightarrow [S]INV$.

4. Applying the B method to Java Card development

Smart cards store software components that are used by client applications, also called *host applications*, that communicate with the card via card acceptance device. Due to obvious restrictions, the code embedded in smart cards has a simple structure. In particular, Java Card imposes stringent restrictions on the Java language, e.g. excluding complex data types and multi-threading. This is one of the scenarios for which the B notation is well adapted.

The B method is used to specify the functionality of the card-side components. However, this would require from the specifier to get into details of Java Card specific communication and security issues. It is possible nevertheless to develop a completely portable B specification of the application, as if it would execute on the same host as the client application. From that specification, we propose to generate the "glue" code responsible for hiding from the host all the complexity of implementing the APDU-based communication protocol to access the card. Assuming that the specified component can be implemented as a JAVA class with an interface I, our approach results in the generation of the following components:

- A host-side component C_h with interface I, responsible for performing remote method invocation on the smart card by means of the standard protocol. Each method in this class is thus responsible for communicating with the smart card, by encoding, sending, receiving and decoding the APDU buffer carrying the method calls, exceptions and return value.
- A smart card-side component C_s, responsible for reading the APDU buffer contents set by C_h, decoding them and dispatching the parameters to the code implementing the logic of the corresponding specification operation, coding the result into a response APDU packet and storing it into the APDU buffer.

4.1. Guidelines for the specification development

Our approach imposes few constraints on the specification of the smart card component. As advocated by the proponents of aspect oriented programming (AOP) [8], the basic functionality of a component should remain separated from other concerns (in our case, the necessity to handle calls, exceptions and return values as APDU buffer contents). Indeed, we see the Java Card specific part of the code as an aspect of the code transversal to the application itself, or, as called in the AOP community, a *crosscutting concern*. In AOP, however, the idea is that the source code of the aspect will still be separated from the code of the application. Here, we do not try to do this, mainly because the management of aspect could create execution resources overhead that smart cards cannot afford and also because the user of the methodology shall not have to work at the source code level. Thus, Java Card "aspects" remain separated at the specification level, while the generated code is already *weaved*, i.e., aspect related code is already included in the application specific code.

But better, from the specifier's point of view, is that our approach relieves him from the duty of specifying the aspect. Indeed, the particular characteristics of Java Card make it possible to automatically generate most of the Java Card aspect code from the original, Java Card free, application specific B specification.

For full automation of the process, however, some restrictions on the B machine apply. They are:

1. The B integer variables must be restricted to the range of some Java Card integer type, that are: *byte, short*. The *int* type is not supported in all Java Card implementations, for this reason it should not be used in a B specification. We provide the specifier with (B specification and implementation of) Java Card compliant data types.
2. The Java Card limitations must be respected for the B implementation machine. For instance, a generated class can have at most 256 public or protected static fields [13] and, consequently, a B machine must obey this restriction.

As long as the restrictions outlined here are respected, it is possible to automatically generate Java code from the B specification of the API. But then, only a generic specification which does not reflect its Java Card aspects would be available. Generation is then executed in two steps:

1. in a first step, a Java Card extended version of the original specification is generated as well as some auxiliary specifications to relate Java Card aspects and application specific aspects (all of them B machines)
2. in a second step, Java Card code is generated.

4.2. The development process

The process proposed in this paper starts from a initial B specification (a B MACHINE) of the desired API. Let us call it `API.mch`. This machine is then submitted to two parallel refinement/implementation sequences:

1. To develop the Smart Card implementation of the API, a refinement is usually applied to make it a full-function machine, i.e., all operations have minimum preconditions (only typing restrictions) and deal internally (through exceptions) with all the potential problems of invalid data and actions. We call this refinement `API_FF.ref`. Normally, an additional machine dealing with exceptions data (say, `API_Exceptions.mch`) is also generated. More information on how to derive this refinement and the corresponding `API_Exceptions.mch` from the original specification is given in section 4.2.1., below.
2. The second line of development takes care of the implementation of the API on the host side and is fully automatable. This implementation (say, `API_Host_imp.imp`) translates the actions specified in each operation of `API.mch` into data that is used by the Java Card Runtime Environment (JCRE) to create APDU commands. It can be viewed as a wrapper, such that the host application remains unaware of the Java Card remote implementation of the machine's attributes and methods. A more detailed description of this implementation process is given in Section 4.2.4..

The relations among these specifications is shown in Figure 3.

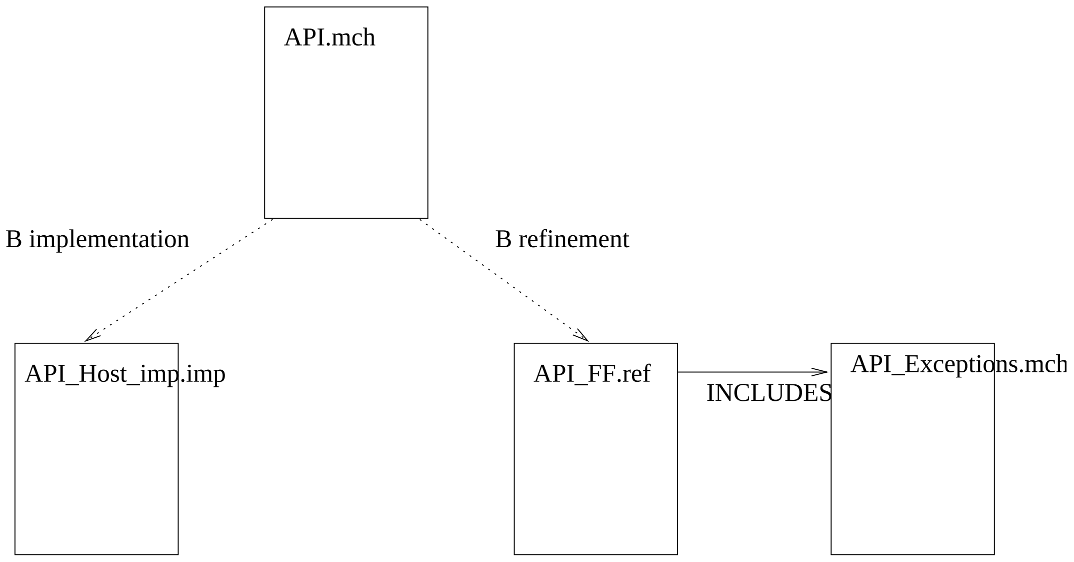


Figure 3. The two lines of development for an API machine

4.2.1. The full function API

The full-function machine `API_FF.ref` refinement is required in order to have a more robust specification for the API. The original abstract machine `API.mch` may only define the main behaviour of each API operation, stating operation's preconditions that have to be satisfied in order to have a correct (invariant preserving) execution of the machine. This kind of non-robust behaviour is in general not allowed in Smart Card applications, and is certainly not the standard style of programming used by Java Card developpers.

Java Card programming style includes internally validating all parameter data for each operation, and exception raising each time a non-conformance is detected. Thus, the specification developer may need to include these exceptions. For that, he must define the `API_Exceptions` machine and include it in the refinement. The `API_Exceptions` machine must contain all the exception names in a enumerated set (called `EXCEPTIONS` by convention). At the `API_Exceptions` implementation level, these constants will be implemented by natural values and, when translated into a Java Card class, a constant declaration will be generated for each exception element of the EXCEPTIONS set.

Example 3 *If we consider the specification of Example 2, its refinement would look like:*

```
REFINEMENT Transport_FF_ref
REFINES
    Transport
INCLUDES
    Exceptions
VARIABLES and INITIALISATION : as in the original specification
OPERATIONS
    addCredit (cr) =
      BEGIN
        IF cr > 0 THEN
          IF balance + cr <= MAX_DATA THEN
            balance := balance + cr
          ELSE
            setException(data_overflow)
          END
        ELSE
          setException(credit_not_positive)
        END
      END
END
```

We provide a template for the API_Exceptions machine, containing a variable to represent the last raised exception, an operation to raise an exception (setException, used in API_FF.ref) and the set EXCEPTIONS. It is the responsibility of the designer to define this set in this machine, or in the refinement API_FF.ref (in this case the machine API_Exceptions needs not be modified).

4.2.2. Generation of a Java Card version of the full function API

The full function machine obtained in the refinement above is not formally refinable (in the sense of the B method) into a regular Java Card implementation of the API: indeed, Java Card requirements and programming style enforce that method have the APDU as formal parameter, which is therefore incompatible with the profile of operations in API.mch.

A translation step has then to be performed to convert the interfaces of operations to the Java Card style. This translation step takes as input API_FF.ref and generates a new B machine that we call API_JC.mch. Note that here we do not strictly follow the B development process as the result of the translation is a new B abstract machine. This machine is then refined and implemented, following the rules of B, into the B implementation corresponding to the code that is to be embedded in the card. This new machine is also imported by the implementation of the API on the host side to implement the variables of API.mch, as shown in Section 4.2.4.. It uses a pre-defined machine, called JCRE.mch, specifying parts of the APDU protocol and of the Java Card Runtime Environment, to directly manipulate the APDU buffer.

This translation step is schematized in Figure 4. Because it only adapts the operations interfaces, it is fully automatable. In Java Card, the method receives the parameters and returns its result via the APDU buffer. Additionally, the values passed through the buffer need to be converted to the actual data types. The generated B machine includes these necessary steps around the functional logic of the operation. This step is similar to the AOP concept of weaving aspect with a class and is completely automatable.

Note that we cannot check immediately that the result of the translation is a refinement, as the interface is modified. However as the resulting module will be nested within a wrapper that reestablishes the original interface, it will be possible to carry out the refinement proof at this level.

4.2.3. Generating auxiliary machines of the Java Card platform

The full function machine API_FF.ref is also the source for the generation of two auxiliary machines needed to specify the logic of the APDU protocol and the Java Card applet, as well as realizing data conversion between Java data types and APDU buffer contents. We call these machines API_Process.mch and API_Conversions.mch.

```
res <-- op (params) =
  BEGIN
    IF params_validity
    THEN
       action
    ELSE
       exception
    END
  END
```

```
op (apdu) =
  PRE
    apdu : APDU_CMD_TYPE
  THEN
    LET params BE
      params = apdu'data
    IN
      IF params_validity
      THEN
        action and set response
      ELSE
        exception
      END
    END
  END
```

Figure 4. Translation of operation interfaces to meet Java Card standards

The `API_Process.mch` defines a unique operation, named `process`, corresponding to the `process` method that every Java Card applet needs to implement, as explained in Section 2.2.. This operation processes the APDU command in order to select which operation is to be executed. It uses data from the second machine, `API_Conversions.mch`, in order to improve readability, staying at a higher level.

```
MACHINE API_Process
SEES
  API_Conversions
INCLUDES
  API_JC
OPERATIONS
  process (apdu) =
    PRE
      apdu : APDU_CMD_TYPE
    THEN
      SELECT
        apdu'ins = op_code_of(op1) THEN
          op1(apdu)
      WHEN apdu'ins = op_code_of(op2) THEN
          op2(apdu)
      ...
      END
  END
END
```

The `API_Conversions.mch` machine specifies all encoding information of the application logic into bytes, the low level data that is communicated and manipulated in the Java Card platform. Its minimal contents are shown below, in the case where the only converted data is the operation names. Further encoding may be needed, e.g., for enumerated types, such as a type that defines a category of users. In this case, additional sets and translation functions have to be included and can be automatically generated using classic compilation techniques.

```
MACHINE API_Conversions
SETS
  OP_NAMES = // names of all operations of the API machine
CONSTANTS
  op_code_of //function relating all OP_NAMES to number codes for them
             //in the APDU
  ... other specific application data to be translated (e.g., enumerated types)
PROPERTIES
    op_code_of : OP_NAMES --> BYTE
END
```

4.2.4. The host application's side

As stated above, from the original abstract specification `API.mch` a B implementation `API_Host_imp.imp` is derived which translates the actions specified in each operation of `API.mch` into data that is used by the Java Card Runtime Environment (JCRE) to create APDU commands.

This B implementation is constructed using data from its abstract implementation `API.mch` and from a series of other machines that define the Java Card platform and the Java Card implementation of the abstract API.

The general structure of this implementation is:

```
IMPLEMENTATION API_Host_imp
REFINES
  API
SEES
  API_Conversions
IMPORTS
  JCRE
  card.API_JC
INVARIANT
  var = card.var       (for each variable in API.mch)
OPERATIONS
  res <-- op(params) =
    VAR dd, st
    IN
      sendAPDUCmd(..APDU data..);
      dd <-- getAPDUResData;
      st <-- getAPDUResStatus;
      res := dd'data
    END
```

In Figure 5 is shown the hierarchy of the resulting specifications, which all "execute" on top of the Java Card Runtime Environment (`JCRE.mch`): on the host side we have `API_Host_imp.imp`, on the card side we have `API_JC.mch` and `API_Process.mch`, and, defining the Java Card encoding of application data, the `API_Conversions.mch`.

Note that the implementation `API_Host_imp` can be automatically generated from `API.mch`. Furthermore it is interface-compatible and is amenable to automated refinement verification using existing B provers.

4.2.5. Case study

As a case study to apply the proposed methodology, we developed a specification of a module for a mass transit system, one of the main business opportunity for smart cards in the world and in Brazil particularly. The requirements in the case study are the following:

1. The cards are subdivided in three categories: full rate, student rate and gratuitous rate.
   (a) The trip price is one *token*.
   (b) Full and student cards need to be loaded with a positive amount of tokens.
   (c) The difference between the full rate and student rate only appears when the user buys tokens.
2. The information stored in the card are its category (student, full and gratuitous), the current amount of tokens, an hour and a date.

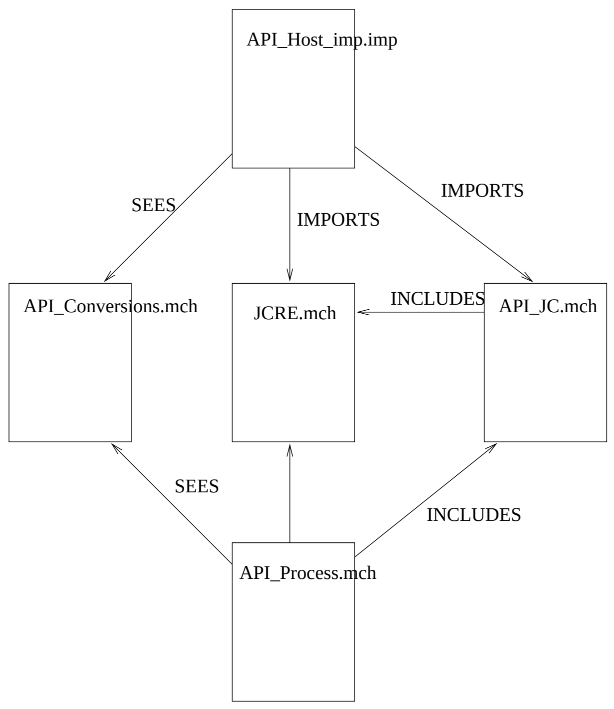


Figure 5. B specification of the Java Card platform "execution" environment

3. When boarding the vehicle, the user must pass their card through the reader.
   (a) For full and student cards, the card is debited one token, and the user is allowed to board.
   (b) For gratuitous cards, no token is debited and the user is allowed to board.
4. Reloading the card is necessary for the full and student cards as soon as the amount of tokens is equal to zero.

A very simplified version of the specification machine has been used as an illustration for the B notation in Example 2, and a full-function refinement in Example 3. We derived all the B modules described in the previous sections and generated the corresponding proof obligations.

4.3. Java Card code generation from the B modules

The previous sections describe how, given a generic B machine specifying an API, it is possible to generate a B implementation that includes data conversion and communication aspects of the Java Card platform. The user only needs to realize a refinement to a full-function version, obeying simple conventions to handle exceptional situations. From that point, our methodology makes it possible to generate B modules that can be shown to be a refinement of the original machine and serve as a basis for a Java Card implementation of this API. The generation of Java Card code from the B implementation level has been studied previously [16] and an open-source prototype implementation is available. This tool is a Java code generator that can be configured to respect Java Card restrictions. As future work, we plan to extend this tool with the ideas presented in this paper.

5. Related work and conclusions

The main contribution of this work is to provide support for a rigorous development of Java Card components for smart card aware applications, based on the B method. Furthermore, we want to hide as much possible the idiosyncracies of Java Card and smart cards. To achieve this goal, we proposed that the specification focuses on the so-called application logic and ignores the aspect related to the implementation of the component required to implement the Java Card communication protocol. As such the specification remains generic enough to be refined and implemented towards other platforms.

Related work and tools concerning the generation of imperative (C, ADA) code from B specifications, e.g., [5, 2, 3], have been around for a while. The generation of object oriented code or models is however still a matter of current research as in, e.g., [7, 6, 15], where the translation from B specifications into UML diagrams is studied. Recently, a Java code generation tool has been developed [16]. This tool is also a product of a Smart Card development project (*Projet BOM*[1]), and it takes care of some memory use optimization issues. Our work builds upon this previous work. However, in this previous work, code generation is executed from an implementation-level B module that is already very close to the Java Card implementation, and the generated code needs to be manually modified to incorporate the communication and codification aspects particular to the Java Card platform. Our proposal is to provide automated support to generate such B IMPLEMENTATION from a generic specification, as it will be viewed by the host application on the terminal side. Thus, all Java Card and protocol specific data and methods are automatically generated from the API's specification.

The proposed refinement and code generation methodology is composed of two steps: first, a complete B specification with the Java Card aspects is produced, allowing for specific verifications to be carried out; only then, Java Card code will be generated. This second step may be partially carried out with existing tools ([16]). As future work, we plan to prototype the ideas presented in this paper, i.e. build a tool that implements all the steps that we identified as automatable, and apply them to different case studies. We also plan to investigate security and authentication aspects within the proposed process.

References

[1] J.-R. Abrial. *The B-Book — Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] B-Core Ltd. The B-Toolkit. B-Core internet site. Available at `http://www.b-core.com/btoolkit.html`. Acessed on: Aug. 13th, 2005.

[3] Didier Bert, Sylvain Boulm, Marie-Laure Potet, Antoine Requet, and Laurent Voisin. Adaptable translator of B specifications to embedded C programs. In *Proceedings of FME 2003*, volume 2805 of *LNCS*, pages 94–113, Pisa, sept. 2003. Springer-Verlag.

[4] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, Boston, 2000.

[5] Clearsy. B language reference manual: version 1.8.5. Available from: <http://www.b4free.com/public/resources.php>, 2004. Acessed on: June 15 2005.

[6] Houda Fekih, Leila Jemmi, and Stephan Merz. Transformation des spécifications B en des diagrammes UML. In *Proceedings of AFADL: Approches Formelles dans l'Assistance au Développement de Logiciels*, Besancon, june 2004.

[1] `lifc.univ-fcomte.fr/RECHERCHE/TFC/rntl_bom.html`

[7] A. Idani and Y. Ledru. Object oriented concepts identifications from formal B specifications. In *Proceedings of 9th Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, Linz, sept. 2004.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[9] Enrique C. Ortiz. An introduction to java card technology. Available from: <http://developers.sun.com/techtopics/mobility/javacard/articles/javacard1>, 2003. Acessed on: november 19 2004.

[10] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. John Wiley & Sons Ltd., Baffins Lane, 3rd. edition, 2003.

[11] Derrick Robinson. The worldwide market for smart cards and semiconductors in smart cards, 2004 edition. Technical report, IMS Research, 2005.

[12] J.M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[13] SUN MICROSYSTEMS INC. Java card platform specification. Available from: <http://java.sun.com/products/javacard/specs.html>, 2003. Acessed on: june 15 2005.

[14] SUN MICROSYSTEMS INC. Java card technology at-a-glance: The foundation for secure digital identity solutions). Available from: <http://www.sun.com/aboutsun/media/presskits/javaone2005/JavaCard_aag_fin al-3.pdf>, 2005.

[15] B. Tatibouet, A. Hammad, and J. C. Voisinet. From abstract B specification to UML class diagrams. In *Proceedings of 2nd IEEE International Symposium on Signal Processing and Information Technology (ISSPIT'2002)*, Marrakech, dec. 2002.

[16] B. Tatibouet, A. Requet, J.C. Voisinet, and A. Hammad. Java Card code generation from B specifications. In *5th International Conference on Formal Engineering Methods (ICFEM'2003)*, volume 2885 of *LNCS*, pages 306–318, Singapore, November 2003.

[17] J. B Wordsworth. *Software Engineering with B*. Addison Wesley, Boston, 1996.