Applying the B Method for the Rigorous Development of Smart Card Applications

Conference Paper · February 2010

DOI: 10.1007/978-3-642-11811-1_16 · Source: DBLP

CITATIONS
4

READS
107

4 authors, including:

David Déharbe
ClearSy System Engineering
114 PUBLICATIONS **940** CITATIONS

SEE PROFILE

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

SEE PROFILE

Applying the B Method for the Rigorous Development of Smart Card Applications*

Bruno Gomes[1], David Déharbe[1], Anamaria Moreira[1], and Katia Moraes[2]

[1] Federal University of Rio Grande do Norte (UFRN), Natal, RN, Brazil
{bruno,david,anamaria}@consiste.dimap.ufrn.br
[2] Petróleo Brasileiro S.A. (PETROBRAS), Rio de Janeiro, RJ, Brazil
katikaka@yahoo.com.br

Abstract. Smart Card applications usually require reliability and security to avoid incorrect operation or access violation in transactions and corruption or undue access to stored information. A way of reaching these requirements is improving the quality of the development process of these applications. BSmart is a method and a corresponding tool designed to support the formal development of the complete Java Card smart card application, following the B formal method.

1 Introduction

Smart card applications are present in our everyday life in a wide range of sectors such as banking and finance, communication, Internet, public transport, health care, etc. These applications are stored in a resource constrained device and usually manage confidential information, such as bank account data, the medical history of a patient or user authentication data.

Java Card [1] is a version of the Java platform with a restricted API and Virtual Machine optimized for smart cards and other memory and processor constrained devices. The Java Card developer can benefit from most of the Java features, such as portability, type-safe language, object oriented development, and the available tools.

To prevent undesirable behavior and to avoid security violations, it is helpful to improve the quality of the smart card development process with the adoption of rigorous software engineering process, methods and tools, to ensure that the final product is in conformance with the specified requirements.

The BSmart project aims to contribute with a method and its corresponding tool support which support the formal development of Java Card application services through a development process that starts from a platform independent B Specification of these services (i.e., a specification where Java Card characteristics are not specified). The application installed on card, which contains the

* This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES) www.ines.org.br, funded by CNPq grant 573964/2008-4 and by CNPq grant 553597/2008-6. The author Bruno Gomes is supported by a doctoral degree scholarship from CNPq.

M. Frappier et al. (Eds.): ABZ 2010, LNCS 5977, pp. 203–216, 2010.

implementation of the services accessed by the client, is developed following a customized B refinement/implementation process of the initial specification. For the client (host application) we provide the generation of an API to communicate with the applet that takes care of all the work related with Java Card protocol aspects and transparent Java/Java Card data coding/encoding.

Previous work [2] [3] introduced the basic structure of the *BSmart* method. Its tool support has been also presented as a short paper in [4]. Here we describe the work current stage, including the API generation for the host side and the developed library machines that may be used in the process of specification, verification and code generation of these applications. With respect to these previous works, the current paper contributes with a more complete support and formalization.

This paper is organized as follows. The Section 2 introduces the Java Card system and its applications. An overview of the process is presented in Section 3. The host API generation and the development of the card application are discussed respectively in Sections 4 and 5. The tool support and the developed libraries are subject of the Section 6. Finally, in Section 7 we present some final considerations and related work.

2 Java Card

Java Card [1] is a restricted and optimized version of the Java platform to allow memory and processor constrained devices, such as smart cards, to store and run small applications. A developer can benefit of many Java features, such as portability, type-safe language, object oriented development, and available tools. This infrastructure allows a rapid application build, test and installation cycle, reducing the time and the cost of software production. However, in face of hardware limitation, Java Card presents some restrictions on resources and API. For example, dynamic class loading, threads, *Strings*, the types *float* and *double*, and multi-dimensional arrays are not present in the current versions of Java Card. The integer type (*int*) and garbage collection are optional.

The main component of the Java Card platform (Fig. 1) is its runtime environment (JCRE), composed of a Java Card Virtual Machine (JCVM), a small API, and, usually, system and industry-specific classes [1]. The JCRE acts as a small operating system, being responsible for the control of the application lifetime, security and resource management.

In this work we are interested in the Java Card software development, presented in Section 2.2. However, before going into the software details, it is helpful to understand some aspects of the smart card system in Section 2.1.

2.1 Smart Card System

A smart card application is distributed between on-card and off-card components. The server application on the card side (called applet) provides the application services and is installed in the smart card EEPROM memory. The off-card

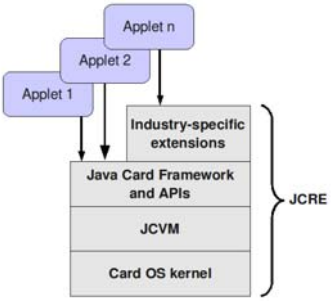

Fig. 1. Java Card platform components. Adapted from: [5].

client (called host application) resides in a computer or electronic terminal. A hardware device, named Card Acceptance Device (CAD), provides power to the card chip and the physical means that the applications uses to communicate [5]. The communication can be processed by electrical contact, when working with contact cards, or by radio frequency for contactless cards.

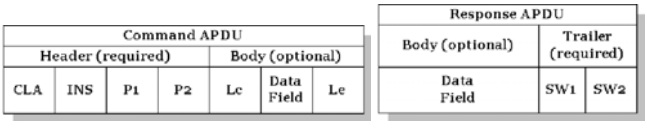

Fig. 2. Command and response APDUs. Source: [5].

The information exchange between the host and card applications is made through a half-duplexed low level communication unit, named *Application Protocol Data Unit* (APDU). The ISO 7816-4 standard specifies two kinds of APDU's, which are the command and the response APDU (Fig. 2). Both specify data packets. A command APDU is sent by the host, requiring some applet service and a response APDU is sent by the applet, responding to the host request with the result of the service processing. Examining Figure 2, one can note that the APDU packets are a low-level structure, requiring information coding (instruction code, data, etc.) when requesting a service and information decoding (data, status), after the service execution.

2.2 Developing Java Card Applications

The complete development process of a Java Card application involves essentially the (i) development of the card side applet and some auxiliary classes when needed, (ii) test and simulation of the developed application, (iii) conversion of the generated bytecode into an appropriate format to be installed on a smart card, and (iv) development of the client side host application. The activities (i) and (iv) are the focus of our work. In the following we give more information on the host application and on the Java Card applet.

Java Card host application: Java Card allows the inter-operability of the developed applet among different smart cards with compatible Java Card specifications. However, complete compatibility can only be obtained when the whole smart card environment, including cards, readers, protocols, and host applications are in accordance to common standards. To achieve this goal, some initiatives taken by a consortia of smart card companies emerged, such as PC/SC [6] and Global Platform [7]. These standards define complete APIs to initialise terminals and cards and to manage the communication through APDU encoding, taking in consideration not only on-card security, but global system security aspects. Other recent API, part of Java 6, is the Smart Card I/O. It is simpler than the others, but is compatible with PC/SC readers and is suitable to most application needs, offering the basic structure for applet communication. A portable host application must then use one of these standards in its implementation.

Java Card applets: A Java Card applet is a class that inherits the *javacard.framewok.Applet* class of the Java Card API and is implemented upon the Java Card subset of Java. During the applet conversion for card installation, a verification phase is performed to check conformance of the classes with Java Card restrictions.

The current usual Java Card specification (2.2.x) allows two kinds of applets. The older, and most commonly used, kind of applet manipulates directly the APDU packages while the newer one abstracts from the lower level protocol using Remote Method Invocation (RMI). In this paper we will call the lower level applet *APDU applet* and the higher level one, *RMI applet*. RMI introduces a layer of abstraction above the APDU protocol, and, due to this fact, it is usually less efficient than APDU applets.

3 Formal Java Card Development with the B Method

Smart card applications usually require the management of confidential information, such as monetary values and data for secure authentication. Thus, a major part of smart card development is devoted to the implementation of an API containing the services provided by the card whilst protecting these data. Being a restricted domain of usually small applications with the need for safety and correctness, smart card applications represent a suitable domain to the adoption of formal development methods.

In this work we present a method to develop a Java Card application following the B method [8], from specification to refinement and code generation. By using B in the whole development process we aim to guarantee the preservation of the specified functional properties from the abstract specification modules until their implementation in Java Card.

Starting from a high-level formal specification of the API, the development (Fig. 3) follows two lines, one for the card side application and the other related to the host application:

host-side: The full automatic generation of an API that encapsulates the communication between the host and card applications, hiding this communication, as well as most of the details of the standards cited on section 2.2, from the developer, who can focus on the functional aspects of the application.

card-side: The development is based on refinement of the initial B specification. It progressively adds Java Card related aspects and a more concrete representation of the application towards its implementation. At the end of this chain of refinements, we are able to generate the implementation of the application.

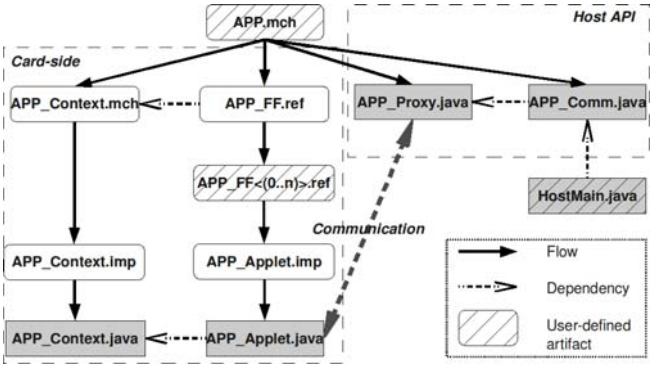

Fig. 3. A view of the BSmart method development and its artifacts

Figure 3 gives a general idea of the artifacts that appear in the development process. Most of the development can be performed using the support of tools (Section 6), but, as usual in B, the developer can add refinement levels to manage complexity or to reduce proof effort.

The Host API component encapsulates communication standards and protocols, so that the application is able to use the card services through the API method calls. In the next sections we detail the aspects related to the generation of the host side API and the development of the card-side application.

4 Generation of the Host Application API

As explained in Section 2.2, for the development of the host application there are some standards which define APIs to supply all the necessary resources to establish the communication with the card. In this work we propose to generate a set of classes (Fig. 4) to transparently communicate with the card application, using one of that APIs, freeing the user of the tedious and error-prone task of manipulating the lower-level details of the Java Card system. This encapsulation includes the management of the connection with a card application and the coding and decoding of the data sent to and received from the applet. The user

keeps the responsibility for the functional aspects of the application only, leading to an increased productivity.

The code of the API components for the host-side is generated from the original specification in a fully automatable process, since, in addition to the desired communication standard API to be used, we only need to know the expected services of the applet and the necessary data to process them. As the B method imposes that the signature of each operation does not change in the refinement process, we can obtain it directly from the operations of the high-level machine.

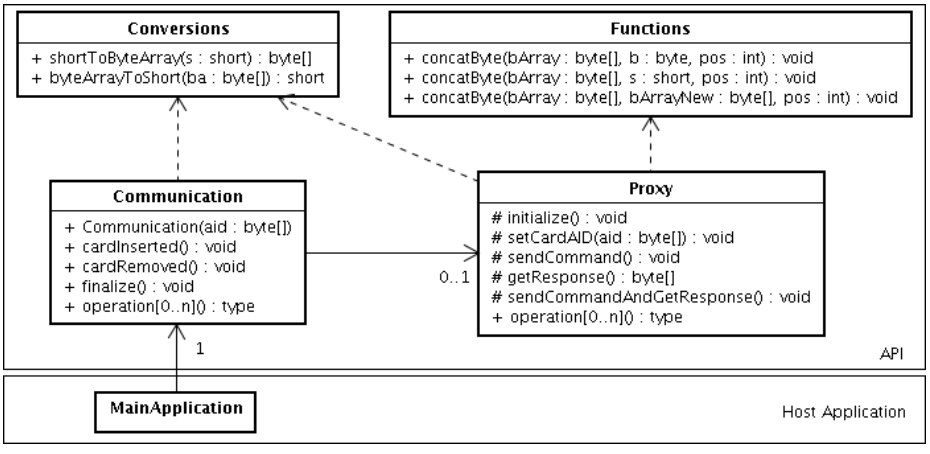

Fig. 4. API structure

As shown by the diagram of Figure 4, two main classes are generated for the host developer, named *Communication* and *Proxy*, with implementations that may vary depending on the kind of Java Card Applet (APDU or RMI) and communication standard. The *Communication* class is seen by the host application and contains high-level Java methods to call each applet service and to control the life-time of the applet. For APDU applets, the service calls are dispatched to the *Proxy* class, which is responsible for coding the received data into a packet in the command APDU format (Fig. 2) and send it to the applet. This class also decodes the returned response, sending it back to the *Communication* class. The *Proxy* class is not necessary for the generation of the RMI host application API since it allows the communication to be taken at a higher level than the APDU level.

The classes *Conversions* and *Functions* contain useful methods to help in the conversion tasks of data between the card and host applications. These are predefined library classes that can be reused in different developments.

Until now, we have implemented the generation for the APIs *OpenCard framework*, *Smart Card I/O* and RMI Client. The last is a simple API, part of the Java Card specification, for the communication with RMI applets.

4.1 Rules for Host API Code Generation

This section briefly describes the rules for the generation of the API classes, namely the rules associated with data emission and reception in the communication process. The set of translation rules is too large to be fully described here; they define a translation that follows the syntactical structure of the B specification. Most of them are straightforward, and the only complication occurs in the case of APDU applets due to the encoding and decoding of data when passed to the communication medium between the card and the host components.

In all cases, each specified operation is mapped into a method of the generated class. In the case of RMI applets, where the data encoding/decoding is taken care by the Java Card RMI library, the general rule for the translation of a B operation is:

generation_from(B_operation) =
let
    jres = java_return_type_of (B_operation)
    name = java_name_of (B_operation)
    jparams = java_param_list_of (B_operation)
    jthrows = (exception throwing declarations: depends on API)
    init_try = "try {"
    end_try = "} catch (Exception e) { e.printStackTrace(); }"
in
    "public" jres name "("jparams")" jthrows "{"
      init_try
      javarmi_app_method_call_of (B_operation)
      end_try
    "}"

The above rules rely on auxiliary functions, such as *java_return_type_of*, *java_param_list_of* and *java_name_of* that, given a B operation, respectively yield the return type, the list of parameters and the name of the corresponding Java method. The function *javarmi_app_method_call_of* uses these predefined functions to make the operation calling statement.

For the APDU version, the *Proxy* class is responsible for the encoding of data for the APDU buffer and the corresponding decoding. These rules are more complex than that for RMI as the arguments and the result of any method call need to be encoded as bytes and stored in an APDU packet. The APDU format has three fields to store data (see Fig. 2): *p1* (1 byte), *p2* (1 byte) and *data* (arbitrary size). We have thus defined and implemented a conversion algorithm that computes the number of bytes needed to encode the method arguments, and generates the Java code to convert such arguments to bytes and store them in the APDU packet. The code thus generated is optimized to reduce communications as it first fills the *p1* and *p2* fields, and uses the *data* field only when the encoding of the arguments is larger than two bytes.

5 Development of the Card-Side Application

As we introduced in Section 3, the formal development of the card applet starts from a high-level B specification of the application. This specification does not need to observe Java Card aspects. To illustrate some concepts of this branch of the development, let us introduce a simple abstract specification of a *counter* (Fig. 5) with an operation to increment the counter. This specification uses a Java compatible int type (JINT), provided in the *JInt* library machine (Fig. 10), which also defines arithmetic operations restricted to the range of the type, such as *sum_jint* to compute the sum of its two integer arguments and *gt_jint*, which returns *true* when its first argument is greater than the second one.

MACHINE *JCounter*
SEES *JInt*
VARIABLES *value*
INVARIANT *value* ∈ *JINT*
INITIALISATION
  value := 0
OPERATIONS
  increment (*vv*) =
  PRE
    vv ∈ *JINT* ∧
    gt_jint(*vv*, 0) = *true* ∧
    sum_jint (*value*, *vv*) ∈ *JINT*
  THEN
    value := *sum_jint* (*value*, *vv*)
  END
END

MACHINE *JCCounter*
SEES *JCInt*, *JInt*, (...), *InterfaceContext*
VARIABLES *jc_value*
INVARIANT *jc_value* ∈ *JCINT*
INITIALISATION
  jc_value := *jcint_of_jint* (0)
OPERATIONS
  jc_increment (*vv*) =
  PRE
    vv ∈ *JCINT* ∧
    gt_jcint(*vv*, *jcint_of_jint*(0))=*true* ∧
    sum_jcint (*jc_value*, *vv*) ∈ *JCINT*
  THEN
    jc_value := *sum_jcint* (*jc_value*, *vv*)
  END
END

Fig. 5. JCounter machine and its Java Card version JCCounter

Here we introduce a common situation that can occur in a typical development: the abstract specification uses types that are not compatible with Java Card types. The *JCounter* abstraction uses the *int* type, which is not built-in Java Card, and the B method does not allow interface and type changing in the refinement process. Thus, to follow the card-side development, we need to deal with this type incompatibility by introducing a refinement pattern.

To achieve the goal of type/interface adaptation without going out the strict rules of B refinement we use library machines that model each type and intermediate machines containing conversion functions and properties relating them. We will present here the general notion and excerpts of the corresponding modules to illustrate the solution for our example in which we model the *int* type through a pair of *shorts*. The detailed approach is described in [9]. The important point in favor of this solution is that the development rigorously obeys B refinement restrictions.

MACHINE *InterfaceContext*
SEES *JInt* , *JCInt*
CONSTANTS *jint_of_jcint*, *jcint_of_jint*
PROPERTIES
$jint\_of\_jcint \in JCINT \rightarrowtail\!\!\!\!\!\rightarrow JINT \wedge$
$jcint\_of\_jint \in JINT \rightarrowtail\!\!\!\!\!\rightarrow JCINT$
ASSERTIONS
$jint\_of\_jcint^{-1} = jcint\_of\_jint \wedge$
dom ($jint\_of\_jcint$) $= JCINT \wedge$
dom ($jcint\_of\_jint$) $= JINT$
END

Fig. 6. The InterfaceContext machine

The right side of Figure 5 presents a counter machine providing the same services as *JCounter* but with interface and typing restrictions compatible with Java Card. The *JCINT* is a definition of the Java integer type represented as a pair of *shorts*, included in the *JCInt* library machine (not detailed here). The *JCCounter* machine is also the initial model of a B development to provide an implementation of the card-side component.

The conversion function linking these two abstract (JINT) and concrete (JCINT) integer representations are put in *InterfaceContext*. This machine also contains some corollaries in the assertions clause. These additional properties are useful to simplify interactive proofs of the development. Since *JCCounter* is a machine, not a refinement, we want to be able to prove that the type adaptation succeeds as a refinement relation. We can do this by refining the abstract *JCounter*, relating the abstract and concrete types in its invariant using the functions defined in *InterfaceContext*, as we can see in Figure 7. So, in case of successful verification, we are able to continue our card development using the concrete *JCCounter*. Although the process of typing adaptation is automatable (possibly with some user assistance), the tool support for the method does not include it yet.

To allow the smart card implementation of the API, an additional refinement is applied to make it closer to Java Card code. We achieve this making it full-function, i.e., weakening the preconditions of the operations so that they only define typing of the parameters. The remaining restrictive conditions are handled in its body through conditional substitutions, whose non-validity leads to the throwing of an exception. This is performed by modeling simple Java Card exception classes in an *Exception* library machine. A dedicated context machine contains the identifier code of each exception and any other constants or Java Card related information that the refinement needs. The full function refinement of *JCounter* and its context machine can be shown in Figure 8. The restrictive precondition stating that the value of the increment must be greater than zero was moved to its body, this way allowing the generation of this verification condition in the translated Java Card code.

```
REFINEMENT JCounter_ref
REFINES JCounter
SEES JInt, JCInt, InterfaceContext
INCLUDES JCCounter
INVARIANT value = jint_of_jcint (jc_value)
OPERATIONS
  increment ( vv ) =
  PRE
    vv ∈ JINT ∧
    sum_jint(jint_of_jcint (jc_value), vv) ∈ JINT
  THEN
    jc_increment (jcint_of_jint (vv))
  END
END
```

Fig. 7. A refinement of JCounter to verify the type adaption correctness

```
MACHINE JCCounterContext
SETS EXCEPTIONS = {non_positive_value}
END

REFINEMENT JCCounterFF_ref
REFINES JCCounter
INCLUDES
    ISOException.Exception(EXCEPTIONS)
SEES JCCounterContext, (...)
VARIABLES     jc_value
INVARIANT     jc_value ∈ JCINT
INITIALISATION
  jc_value := jcint_of_jint (0)
OPERATIONS
jc_increment ( vv ) =
PRE
    vv ∈ JCINT ∧
    sum_jcint (jc_value, vv) ∈ JCINT
THEN
    IF ¬(gt_jcint(vv, jcint_of_jint(0))=
          true)
    THEN
       ISOException.throwIt(
            non_positive_value)
    END;
    jc_value :=sum_jcint (jc_value,vv)
  END
END
```

Fig. 8. The full function version of JCCounter and its context machine

The translation of the B0 implementations to Java Card code is the last stage in the card-side development. The main development implementation, containing the services offered to the host, generates the applet class. As usual, other modules may have to be generated, such as the context machine. In Section 6.1 we can see part of the counter implementation, emphasizing the use of the APDU library machine for data sending and receiving.

6 Tool Support

The BSmart tool [4] is an Eclipse plugin connecting several software components, each responsible for implementing a different step of the BSmart method. Essential software for the B formal method is also included, such as a type checker,

and connection with external tools, such as Atelier B, for proof obligation generation and verification. Also, as explained in the next Subsection, we supply jointly with the tool a library of B modules modeling essential classes of the Java Card API, types and useful data structures.

The main components that provide support for the method are the *BSmart Modules Generator* and the *B to Java Card code translator*. The former is responsible to generate the B refinements required by the method and the latter translates all the B implementation modules into Java Card programming code and also generates the API classes for the host side client. The translator has been developed based on the Java translator of JBtools [10]. We modified this open-source B method tool to allow the translation for Java Card.

6.1 A Library of Reusable B Components

We have developed B machines to model Java/Java Card primitive types and some classes of the Java Card API. We plan to supply these verified B models to all classes of the Java Card API and to other useful tasks for Java Card applications, such as manipulation of time, date, currency, etc.

The specification of the Java Card API specification was realized using as basis the official documentation of the classes, as well as JML-based and OCL-based specifications [11,12]. In our approach, the specification serves for the purpose of: (i) providing verified B models of the API (ii) using these verified modules in the refinement of the card-side application, allowing us to verify the correctness of its use in relation to adequate data and the necessary dependencies, for example, when an operation requires the calling of a previous one and (iii) facilitating the generation of the Java Card code, since an operation of an API model is translated to its corresponding method call in Java Card.

As an example, we can see in Figure 9 some excerpts of the APDU class model in B, one of the most important of the Java Card API. Through it one can access the APDU buffer for exchanging data with the host application. The details of this process is treated internally with the Java Card Runtime Environment (JCRE) and it is not our propose to model it. As we said before, we are interested in the practical use of the operations to allow verification and code generation. On the right side of the figure, we show a practical example of the use of APDU machine in the implementation of the *counter* development. The APDU machine is imported and the operations are called as we do in a Java Card applet method to receive data and to send it after processing.

In the case of the types library we have developed machines to deal with the types *short*, *int*, *boolean* and a module to represent the type *int* for Java Card as a pair of *shorts*. As we can see for the type *int* in Figure 10, each machine has constants for type definition and useful functions to operate within the bounds of the type.

The development of the library is still in progress but we expect that when concluded the developed modules can be reused by B specifiers and Java Card developers with the advantage of being fully verified using the B method. We

```
MACHINE APDU (...)
CONCRETE_VARIABLES (...)
    state, buffer
INVARIANT
    state ∈ TBYTE ∧
    state ∈ ST_INITIAL ..
            ST_FULL_OUTGOING ∧
    buffer ∈ (0 .. 132) → TBYTE (...)
OPERATIONS
    res ← setIncomingAndReceive =
    PRE
        state = ST_INITIAL
    THEN
        CHOICE
        state := ST_PARTIAL_INCOMING
          OR
        state := ST_FULL_INCOMING
        END ||
        ANY value
        WHERE
          value ∈ TSHORT ∧ value ≥ 0 ∧
          value ≤ buffer(OFFSET_LC) ∧
          buffer(OFFSET_LC) +
          BUFFER_HEADER_LENGTH
                ≤ BUFFER_LENGTH
         THEN   res :=value
        END
    END (...)
```

```
IMPLEMENTATION
    JCCounter_imp
REFINES
    JCCounterFF_ref
SEES
    JCCounterContext,
    TShort (...)
IMPORTS
    ISOException.Exception(
            EXCEPTIONS),
    apdu.APDU(...)
OPERATIONS
    jc_increment ( vv ) =
    VAR buffer, (...), value_lc, le, res
    IN
        buffer ← apdu.getBuffer; (...)
        value_lc ←
        apdu.setIncomingAndReceive;

        ( ... data processing ...)

        le ← apdu.setOutgoing;
        apdu.setOutgoingLength(1);
        buffer(0) := res;
        apdu.sendBytes(0, 1)
    END (...)
```

Fig. 9. Part of APDU machine (left) and its use in an implementation (right)

```
MACHINE JInt
SEES TBoolean
CONCRETE_CONSTANTS
    MAXJINT, MINJINT, JINT,
    sum_jint, subt_jint, ... , equal_jint, gt_jint
PROPERTIES
    MINJINT ∈ ℤ ∧ MINJINT = - 2147483648 ∧
    MAXJINT ∈ ℤ ∧ MAXJINT = 2147483647 ∧
    JINT = MINJINT .. MAXJINT ∧
    sum_jint ∈ JINT × JINT ⇸ JINT ∧
    sum_jint = λ ( a1 , a2 ) . ( a1 ∈ JINT ∧ a2 ∈ JINT ∧
        ( a1 + a2 ) ∈ JINT | a1 + a2 ) (...)
END
```

Fig. 10. Part of JInt library machine: type definition and *sum* operation

therefore contribute for the correctness of the generated application as a whole, since not only the core application, but all its necessary support classes are subject to formal development and verification.

7 Conclusions

The starting point of this work was to identify the general structure of Java Card applications, and to develop B specifications of some typical Java Card applications, e.g. ticketing, electronic wallet, etc. These case studies evolved to the BSmart method and the first version of its tool support.

Current B development tools include code generation for imperative languages such as C and ADA. The development of optimized C code for smart cards has been subject of study of the *B with Optimized Memory* (BOM) project [13] [14], proposing optimizations such as method inlining. Optimization is an open issue in our work, but we plan, for instance, to reduce the number of local variables introduced in an operation and to minimize class instantiation. Regarding the translation for Java, a recent initiative is the integration of a translator in the *Rodin* platform [15]. A first Java Card synthesis approach has been proposed in [16]. It was implemented in the *JBtools* platform [10] and provides a code generator for Java optimized for Java Card compatibility. However, there is no specific generation for the Java Card applet and no API support is provided for the host application. The code generation for Java Card is also restricted to the *short* integer type.

Our goal is to provide a complete Java Card service generation method, consisting of the card-side application development, as well as an API for host applications to transparently access the card services. Thus the user does not need to deal with type adaptation/conversion and the Java Card lower-level protocols. The method, jointly with the provided B library of Java Card classes, types and useful data structures, form the basis of an environment to effectively and efficiently develop fully-verified Java Card software.

As future work, we want to verify the result of the translation to Java and Java Card. An approach is the inclusion of JML annotations in the generated code to allow runtime checking, as in the work of [17]. We also plan to apply advanced language transformation techniques, such as TXL [18] or ASF+SDF [19] to generate part of the B refinements and the Java Card application final code to replace our *ad hoc* low level implementation of the transformation rules.

Finally, to better validate the proposal and its tool support, we have to develop a more complex case study. A good candidate is the Mondex electronic purse, a case study that is part of the Verified Software Initiative. In the Mondex system some amount of monetary value is transferred from a source to a target smart card purse in a non-atomic protocol. Each purse must be implemented in isolation, without sharing properties through a global control. The Mondex system has been formally specified in Event-B in Butler and Yadav [20] work and in other work using several formalisms. We started to adapt this system specification to a programming specification, extracting the card and host specifications, modeling them according to the BSmart method.

References

1. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley, Reading (2000)
2. Gomes, B., Moreira, A.M., Déharbe, D.: Developing Java Card applications with B. In: Brazilian Symposium on Formal Methods (SBMF), pp. 63–77 (2005)
3. Deharbe, D., Gomes, B.G., Moreira, A.M.: Automation of Java Card component development using the B method. In: ICECCS, pp. 259–268. IEEE Comp. Soc., Los Alamitos (2006)
4. Déharbe, D., Gomes, B.G., Moreira, A.M.: Bsmart: A Tool for the Development of Java Card Applications with the B Method. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 351–352. Springer, Heidelberg (2008)
5. Ortiz, E.C.: An Introduction to Java Card Technology, `http://java.sun.com/javacard/reference/techart/javacard1` (2003)
6. PC/SC Workgroup: PC/SC Workgroup Web site (2009), `http://www.pcscworkgroup.com`
7. Global Platform: Global Platform Web site (2009), `http://www.globalplatform.org`
8. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge U. Press, Cambridge (1996)
9. Déharbe, D., Gomes, B.G., Moreira, A.M.: Refining Interfaces: The Case of the B Method. Technical report, Fed. Univ. of Rio Grande do Norte (2009) (to appear)
10. Voisinet, J.C.: JBtools: an experimental platform for the formal B method. In: Principles and Practice of Programming, Maynooth, NUI, pp. 137–139 (2002)
11. Meijer, H., Poll, E.: Towards a Full Formal Specification of the Java Card API. In: Attali, S., Jensen, T. (eds.) E-smart 2001. LNCS, vol. 2140, pp. 165–178. Springer, Heidelberg (2001)
12. Larsson, D.: OCL Specifications for the Java Card API. Master's thesis, School of Computer Science and Engineering, Göteborg University (2003)
13. Requet, A., Bossu, G.: Embedded formally proved code in a smart card: Converting B to C. In: ICFEM 2000, York, UK, p. 15. IEEE Computer Society, Los Alamitos (2000)
14. Bert, D., et al.: Adaptable translator of B specifications to embedded C programs. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 94–113. Springer, Heidelberg (2003)
15. Edmunds, A., Butler, M.: Code Generation for Event-B with Intermediate Specification. In: Rodin User and Developers Workshop (2009), `http://wiki.event-b.org/index.php/Rodin_Workshop_2009`
16. Tatibouet, B., Requet, A., Voisinet, J., Hammad, A.: Java Card Code Generation from B Specifications. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 306–318. Springer, Heidelberg (2003)
17. Costa, U., Moreira, A., Musicante, M., Neto, P.: Specification and Runtime Verification of Java Card Programs. In: Brazilian Symp. on Formal Methods (2008)
18. Cordy, J.: The TXL Programming Language (2009), `http://www.meta-environment.org`
19. Meta-Environment.org: The ASF+SDF Meta-Environment (2009), `http://www.txl.ca/index.html`
20. Butler, M., Yadav, D.: An Incremental Development of the Mondex System in Event-B. Formal Aspects of Computing 20(1), 61–77 (2007)