Automation of Java Card component development using the B method.

Conference Paper · January 2006

DOI: 10.1109/ICECCS.2006.51 · Source: DBLP

3 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

Automation of Java Card component development using the B method*

David Déharbe† Bruno Gurgel Gomes
Anamaria Martins Moreira
Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN)
Campus Universitário, Lagoa Nova
59072-970 – Natal – RN – Brazil
{david,bruno,anamaria}@consiste.dimap.ufrn.br

Abstract

This paper presents a method 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 application logic does not need to take into account the specific aspects of the Java Card platform (in particular, communication between the card acceptance device and the smart card itself). A sequence of pre-established refinements is then applied to the original specification to yield an implementation-level B description of the component, which can then be used to synthesize Java Card code. This method reduces significantly the required amount of user-interaction and improves productivity. An interesting side-effect of this approach is that the specification may be reused with any other platform of implementation.

1. Introduction

Smart card support is a component of the IT infrastructure in a growing number of sectors [12]: 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 [5] is one of the leading technologies in this sector as it provides significant features: multiple applications, portability, compatibility with a popular 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 method [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 is one of the foremost formal methods with a strong industrial support. It has been succesfully applied in the development of complex and safety-critical applications such as the automatic train operating system for METEOR, a driverless metro in the city of Paris [3].

In this paper, we propose a specialization of the B method that aims at improving the productivity and reliability in the development of Java Card software. Previous work showed the possibility to automatically generate Java code [17] and Java Card 2.1 components [8] from B modules. [17] 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. [8] does not take advantage of the high-level communication facilities provided in Java Card 2.2. The goal of the research presented in this paper is to provide B design guidelines specific for the Java Card framework which make it possible to target the Java Card 2.2 computing environment. An implementation of these guidelines is under development. This implementation will provide tool support for our method, automating part of the design process.

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

*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.

†Contact author.

of the B method 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

An increasing number of IT applications require that users provide data via a portable device. Moreover, for security reasons, it is desirable that such devices include at least simple processing capabilities. The smart card technology answers these needs and gains acceptance and popularity. Unique amongst the different smart card operating platforms, Java Card provides vendor inter-operability and has now reached a *de facto* standard status in this industry [11].

The Java Card platform provides a subset of the Java programming language. 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 [5]. 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 cycle, 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 code running 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 located 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 Applications are stored in the card memory. 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 EEPROM memory. The EEPROM memory can also be written by applications to store their data. Smart cards also have a (faster) RAM memory to store temporary data. 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% [15] 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 packet 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* [5].

With this protocol, it is possible to write a host application in a different programming language than that of the applet, but it requires coding with a high-degree of details, requiring to encapsulate data into sequences of bytes. The Java Card platform, since Version 2.2, provides a remote invocation mechanism (RMI) that hides most of the complexity involved in communicating with the APDU protocol.

2.2. The Java Card Remote Method Invocation framework

The Java Card RMI assumes that the host application needs to use a service provided by an applet on the card as an application programming interface (API). This service is specified as a Java interface that extends directly the predefined Java `Remote` interface (see example in Figure 1). The methods of the corresponding implementation class (see Figure 2) are invokable from a different virtual machine (in this case, the host-side virtual machine). This implementation class needs to be developed in the Java Card dialect. This class shall inherit from the `CardRemoteObject` class provided by the Java Card RMI framework, that provides methods to enable and disable the remote access to objects. The RMI also provides the class `RMIService` that translates method invocations to APDU-level communications. Java Card imposes restrictions due to the very nature of the platform it runs on: there is a restricted set of basic data types, no threads, no guarantee that there is a garbage collector either. Also, in order to avoid loss of data consistency on the card, the Java Card framework provides transaction facilites, as well as specific mechanisms to distinguish persistent data (need to be maintained when the power is turned off) from transient data (may be erased when the card is reset).

In the Java Card environment, applications are called *applets*, and are classes inheriting from the *javacard.framework.Applet* class. An applet must provide an implementation for the methods *install* and *process*. 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. In Figure 3, we present on an example how the implementation of `Remote` object can be integrated into an applet and associated with a `RMIService` object responsible for the communication with the host-side.

Finally, a reference to the remote object needs to be created on the host-side. The OpenCard framework [9] provides functions to get such reference. Once bound to a local object, the RMI is transparent to the programmer (see example of client code in Figure 4).

2.3. Elements of Java Card programming

Programming for a smart card requires special care against two possible problems:

available memory Smart card usually have a very limited amount of memory; in addition, the runtime environment does not necessarily have a garbage collector.

```
package br.ufrn.smart.services;
import java.rmi.*;
import javacard.framework.*;
public interface Counter extends Remote{
  public static final short UFLOW     = (short)0x6000;
  public static final short OFLOW     = (short)0x6001;
  public static final short MAX_VALUE = (short)80;

  public short getValue()
    throws RemoteException;
  public void increment(short n)
    throws RemoteException, UserException;
  public void decrement()
    throws RemoteException, UserException;
}
```

Figure 1. Remote interface for a simple Counter service

```
package br.ufrn.smart.services;
import javacard.framework.UserException;
import javacard.framework.service.CardRemoteObject;
import java.rmi.RemoteException;
public class CounterImpl
extends CardRemoteObject implements Counter {
    private short value = 0;
    public CounterImpl() {
        super();
    }
    public void increment(short n)
    throws RemoteException, UserException {
        if ((short) value + n > MAX_VALUE)
          UserException.throwIt(OFLOW);
        ++value;
    }
    public void decrement()
    throws RemoteException, UserException {
        if (value == 0)
          UserException.throwIt(UFLOW);
        --value;
    }
    public short getValue()
    throws RemoteException {
        return value;
    }
}
```

Figure 2. Java Card **implementation class for a Counter service**

```
package br.ufrn.smart.services;
import java.rmi.*;
import javacard.framework.APDU;
import javacard.framework.ISOException;
import javacard.framework.UserException;
import javacard.framework.service.*;
public class CounterApplet
extends javacard.framework.Applet {
    private Dispatcher disp;
    private RMIService serv;
    private Remote counter;

    public CounterApplet() {
        counter = new CounterImpl();
        disp = new Dispatcher( (short) 1);
        serv = new RMIService(counter);
        disp.addService(serv,
            Dispatcher.PROCESS_COMMAND);
        register();
    }
    public static void install (
                          byte[] aid,
                          short s,
                          byte b) {
        new CounterApplet();
    }
    public void process(APDU apdu)
    throws ISOException {
        disp.process(apdu);
    }
}
```

Figure 3. An applet in the Java Card RMI framework

```
JavaCardRMIConnect jcRMI =
    new JavaCardRMIConnect (cardAccessor);
jcRMI.selectApplet (RMI_COUNTER_AID);
Counter myCounter =
    (Counter) jcRMI.getInitialReference();
myCounter.increment(1);
```

Figure 4. Binding the applet in the host application (`cardAccessor` has been previously initialized using the Open Card framework)

The programmer needs to apply specific memory allocation strategies. For instance, a method should avoid, at all cost, allocation of objects, as there is no available garbage collection mechanism. Thus, object allocation is usually restricted to the constructor. Also Java Card provides a special exception mechanism, where the cause of an exception is a short value instead of a string as in Java. This mechanism is optimized to avoid multiplicating the number of exception objects in the card memory.

data coherency The smart card may be physically removed from the card acceptance device at any time, causing a power failure and interrupting the execution of the virtual machine. To avoid that objects get into inconsistent states, Java Card provides a transaction mechanism that guarantees atomicity of execution (at a cost). In addition, objects may be specified as being persistent (maintain their value when power is turned off) or transient (are reinitialized when power is turned off).

3. Software development with B

The B method for software development [18, 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 [13]. 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 [6, 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 method

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 v can be specified as $v := v + 1$. 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, $[v := v + 1]v \geq 0 = v + 1 \geq 0$. 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 v may be specified as $v := PRE\ v < MAX\ THEN\ v := v + 1\ END$. 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 1 *A very simple example of a B machine is:*

```
MACHINE Counter
SEES Short
CONSTANTS max_val
PROPERTIES max_val : SHORT & max_val > 0
VARIABLES value
INVARIANT
    value : SHORT & value <= max_val
INITIALISATION
    value := 0
OPERATIONS
    increment (n) =
      PRE n : SHORT & n > 0 & value + n <= max_val
      THEN value := value + n
      END;
    decrement =
      PRE value  > 0
      THEN value := value - 1
      END;
    res <-- getValue =
      BEGIN
        res := value
      END
END
```

This example illustrates the most basic clauses to build a specification. The machine name is defined in the *MACHINE* clause. The state variable, named here *value*, is typed in the *INVARIANT* clause and initialized in the *INITIALISATION* clause. Operations are provided to increment, decrement and return the current value of the counter. The constant upper bound of the counter is declared in the *CONSTANTS* clause and its value is constrained in the *PROPERTIES* clause.

3.3. Proof obligations

To guarantee the correctness of a B module, proof obligations must be generated from the initialization and the operations clauses, 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: $PRE \wedge INV \Rightarrow [S]INV$.

Refinements are also subject to verification. The B method generates proof obligations establishing that the result of a refinement is compatible with the original specification: pre-conditions cannot be weakened, post-conditions cannot be strengthened, and the invariant must be preserved.

4. Applying the B method to Java Card development

Smart cards store software components used by client applications, also called *host applications*, that communicate with the card via a card acceptance device. Due to the limitations of the physical support, the code embedded in smart cards needs to have a simple structure. In particular, Java Card imposes stringent restrictions on the Java language, e.g. excluding complex data types and multithreading. 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 the 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. If the target platform is Java Card 2.1, then the code generated from the B artifacts needs to implement the coding and decoding of method calls in the APDU-based communication protocol to access the card. We have shown in a previous work an approach to generate automatically such code from a B specification [8].

When the target platform is Java Card 2.2, then the code synthesis can take advantage of the Remote Method Invocation mechanism (RMI) to code at a higher level of abstraction. 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:

- The remote interface, as in Figure 1, contains the declaration of a Java Card method for each operation appearing in the B specification. This interface provides the smart card services usable from the host applications (as in Figure 4).
- The Java Card class, as in Figure 2, provides an implementation of the remote interface and is installed on the smart card. Each operation specified and refined in the B specification is implemented, respecting the limitations of the Java Card platform.
- A Java Card applet instantiates an implementation object. The generation of the applet code is very simple as it consists in instantiating a template with the name of the implementation class (see Figure 3).

4.1. Restrictions on the specification development

Smart-card aware applications require robust card-side operations, and are developped using a defensive style of programming and be prepared to handle all invalid data. The type system of Java Card provides a limited support to check the pre-condition on the input parameters of the operations and must be complemented with conditional constructs to test the parameter values and, possibly, generate run-time exceptions.

Also, due to the limitations of Java Card, we initially assume that the interface of the component under development satisfies these limitations, in particular regarding the available data types. Thus, some restrictions on the B machine and its successive refinements apply:

1. The B integer variables must be restricted to the range of some Java Card integer type: *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 machines that model the basic Java Card data types.
2. The B implementation must respect the Java Card limitations. For instance, a generated class can have at most 256 public or protected static fields [14] and, consequently, a B machine must obey this restriction.

4.2. The development process

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

1. B conditions on variables, parameters and constants do not provide a clear distinction between typing and further semantics restrictions. In this first step, we require that the user produces a refinement of `API.mch`, named `APItype.ref`, making a clear syntactical separation of these different concerns (Section 4.2.1).
2. Then, based on this separation, the user shall write a refinement of `APItype.ref`, named `APIfull.ref`, with full-function behaviors, i.e., all operations have minimum pre-conditions (only typing restrictions) and

deal internally (through exceptions) with all the potential problems of invalid data and actions. A by-product of this phase is the definition of an additional machine, named `UserException.mch`, specifying the exceptions data (Section 4.2.2).

3. The next step introduces the architecture of the remote invocation mechanism. The refinement `APIfull.ref` is used to generate a new refinement, called `APIhost.ref`, that models the RMI wrapper class on the host-side, and a new machine, called `APIcard.mch`, to model the implementation class on the card-side (Section 4.2.3).

4. As usual in the B method, other refinement steps may be needed to make the substitutions that define the operations directly translatable into Java Card code.

The relations among these different artifacts are depicted in Figure 5; the details of the method will be illustrated on the machine of Example 1.

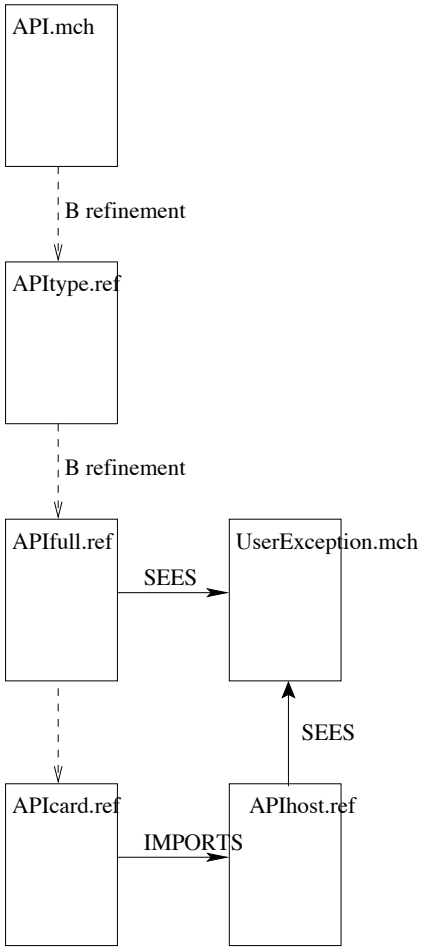

Figure 5. Full function refinement of an API machine

4.2.1 Refinement for typing

Due to the platform requirements, the implementation needs to be full-function. In B, this corresponds to so-called minimal preconditions, which only define typing of data. However, B does not distinguish typing constraints from other semantic restrictions in the different clauses specifying the possible values of machine data (e.g. variables, parameters, etc.). When a B implementation is translated to a programming language implementation, the type-related constraints are mapped to types, whereas the remaining conditions are omitted, as they were shown preserved throughout the refinement process.

So it is common practice that the user of the B method separates typing concerns from other semantic restrictions. In our method, we expect that this is done in the first refinement step. The user needs to rewrite the clauses constraining data so that typing conditions are represented as atoms that do not contain any other restriction.

Example 2 *The typing refinement of the `Counter` machine (Example 1) is given below.*

```
REFINEMENT Countertype
REFINES Counter
PROPERTIES max_val : SHORT & max_val > 0
VARIABLES value
INVARIANT
    value : SHORT & 0 <= value & value <= max_val
INITIALISATION
    value := 0
OPERATIONS
    increment(n) =
      PRE n : SHORT & n > 0 & value + n  <= max_val
      THEN value := value + n
      END;
    decrement =
      PRE value > 0
      THEN value := value - 1
      END;
    res <-- getValue =
      BEGIN
        res := value
      END
END
```

4.2.2 The full function API

The full-function machine `APIfull.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 generally needs to define and include such " invalid argument" exceptions. We recommend that a `UserException` machine is defined and included in the refinement. This machine contains an operation to raise an exception (`throw_it`, used in `APIfull.ref`), and the set `REASON` and is automatically generated from a list of exception names, provided by the user, and corresponding to the non-typing preconditions of the previous refinement (`APItype.ref`). At the `UserException` implementation level, these constants will be implemented by natural values and, when generating the remote interface, a `short` constant declaration is generated for each exception element of the `REASON` set (see Figure 1).

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

```
REFINEMENT Counterfull
REFINES
    Countertype
SEES
    UserException
VARIABLES
    value
INITIALISATION
    value := 0
OPERATIONS
    increment (n) =
      PRE n : SHORT
        THEN
          IF not(n > 0) THEN throw_it(iarg)
          ELSIF not((value + n) <= max_val)
              THEN throw_it(oflow)
          ELSE value := value + n
        END
      END;
    decrement =
      BEGIN
        IF not(value > 0) THEN throw_it(uflow)
        ELSE value := value - 1
        END
      END;
    res <-- getValue =
      BEGIN
        res := value
      END
END
```

The corresponding `UserException` machine is as follows.

```
MACHINE UserException
    SETS REASON = {iarg, oflow, uflow}
OPERATIONS
    throw_it (r) =
      PRE r : REASON
      THEN
        skip
      END
END
```

This whole step is automatizable from the definition of the original API machine and from mappings, given by the designer, between groups of atomic conditions in the preconditions of the machine operations and corresponding exception names for each group.

Moreover, the reasoning engine employed in the B method can be used to perform verifications on the chaining of exceptional condition tests. The traditional application B method already guarantees the correctness of the refinement, and we can additionally verify if the exceptional conditions are all live (i.e. no condition is subsumed by the disjunction of the previous conditions).

4.2.3 Introduction of the RMI architecture

The next step introduces the architectural RMI aspect into the design. The refinement `APIhost.ref` defines the host-side component. It wraps the instantiations of the card-side operations, now defined in a new machine, named `APIcard.mch`. The `APIcard.mch` machine is derived from `APIfull.ref` by renaming the operations of the API. Since this transformation only adapts the operations names, it is fully automatable. Note that here we do not strictly follow the B development process as the result of the translation is a new B abstract machine. Nevertheless, since the body of the operations in the card-side machine is unchanged, and since the operations in the host side refinement only instantiate them, the proof of correctness is trivial and automatically discharged by the current B tools.

Example 4 *From the specification of Example 3, the two specifications below are generated:*

```
MACHINE Countercard
SEES UserException
CONSTANTS max_val
PROPERTIES max_val : SHORT & max_val > 0
VARIABLES
    value
INVARIANT
    value : SHORT & 0 <= value & value <= max_val
INITIALISATION
    value := 0
OPERATIONS
    increment_cs (n) =
      /* same as increment from Counterfull */
      END;
    decrement_cs =
      /* same as decrement from Counterfull */
    res <-- getValue_cs =
      /* same as getValue from Counterfull */
END

IMPLEMENTATION Counterhost
REFINES Counterfull
IMPORTS Countercard
```

```
OPERATIONS
    increment (n) =
      BEGIN
        increment_cs(n)
      END;
    decrement =
      BEGIN
        decrement_cs
      END;
    res <-- getValue =
      BEGIN
        res <-- getValue_cs
      END
END
```

The card-side machine may be further refined, following the rules of B, eventually resulting in the B implementation corresponding to the code that is to be embedded in the card.

4.2.4 A note on the data types in the interface

For sake of usability and portability, it is desirable that the data type restrictions stated in Section 4.1 are lifted (or at least relaxed). Indeed, the original B specification may include operations with interfaces (parameters and results) that do not use the pre-defined models of Java Card data types. Since B requires that refinement preserve observable behavior, including the data type of parameters, it is not always possible to directly map a B machine to a Java Card implementation.

In case the machine interface does not use models of Java Card data types, once the full-function refinement has been produced, the designer needs to specify further refinements on the host side of the two-tier architecture proposed in Section 4.2.3. This first tier is responsible for the conversion between arbitrary Java data types and the Java Card compliant data types in addition to the remote invocation of the card-located operations, while the second tier keeps its original role to define the operations that are to be executed on the card. The refinement realized by the designer will generate an operation that adheres to the following template, where `convert` defines conversion between Java and Java Card. Note that in this situation, the proof of correctness is more complex than in the case reported in Section 4.2.3, as the card-side operations may no longer have the same body as the refined operations.

```
    results <-- operation (parameters) =
      VAR jc_parameters, jc_results IN
      BEGIN
        convert(parameters, jc_parameters);
        jc_results := operation_cs(jc_parameters);
        convert(results, jc_results)
      END;
```

This first tier is then implemented in Java software and executed on the host side, while the second tier is synthesized in Java Card code, running on the smart card.

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 generation of Java Card code from the B implementation level has been studied previously [17] and an open-source prototype implementation is available. This tool is a Java code generator that can be configured to respect some Java Card restrictions. However the code synthesis algorithm implemented in this tool needs further work to take into account the memory allocation and data consistency issues discussed in Section 2.3, using the following approaches:

memory allocation Different coding strategies of the same algorithm generate different quantities of bytecode: an if statement might be advantageously replaced by a switch statement, factoring different occurences of equivalent code may be factored into a private function, several accesses to the same array position can be substituted to a single access stored into a variable, etc. The tool shall be able to try different codification strategies and choose the best one. Also, to reduce memory usage at run-time, it is important to detect the memory requirements of the different operations. If two or more operations require creating the same kind of object, instead of multiple allocations of the object, a single instance of the object can be added to the state of the applet and referenced from each of the corresponding method.

data consistency Java Card applets are, by default, maintained as a persistent objects in the persistent memory of the card. The body of a method that requires modification of the state of the applet needs to be encapsulated as a transaction. Such situations are identified, at the level of the B source code, whenever a state variable is the target of a substitution.

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., [6, 2, 4], 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., [10, 7, 16], where the translation from B specifications into UML diagrams is studied. Recently, a Java code generation tool has been developed [17]. 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. Currently, the method has the capability to produce Java Card code from a restricted subset of B specifications with minimal user intervention (explicit typing and functional conditions). In this context, the B abstract machine notation can be seen as a high-level language for smart card component development.

The proposed refinement and code generation method 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 ([17]). We are currently prototyping the ideas presented in this paper, i.e. building a tool that implements all the steps that we identified as automatable, and apply them to different case studies. Also, as work in progress, we are now developing a repository of B models of Java data types as well as their mapping to Java Card data types, to relieve the designer from the burden of specifying such conversions. 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] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Mtor: A successful application of b in a large project. In *FM'99 - Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*.

[4] D. Bert, S. Boulm, M.-L. Potet, A. Requet, and L. 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.

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

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

[7] H. Fekih, L. Jemmi, and S. 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.

[8] B. Gomes, A. M. Moreira, and D. Déharbe. Developing java card applications with b. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF'2005)*, pages 63–77, 2005.

[9] IBM. *OpenCard Framework 1.2 Programmer's Guide OpenCard Framework 1.2*, 1999.

[10] 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.

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

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

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

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

[15] SUN MICROSYSTEMS INC. Java card technology at-a-glance: The foundation for secure digital identity solutions). http://www.sun.com/aboutsun/media/presskits/javaone2005, 2005.

[16] 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.

[17] B. Tatibouet, A. Requet, J. 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, Nov. 2003.

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

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