The B Bank: A Complete Case Study

Martin Büchi
Turku Centre for Computer Science, Lemminkäisenkatu 14A, FIN-20520 Turku
Martin.Buechi@abo.fi http://www.abo.fi/˜mbuechi/

Abstract

We develop a Web-based banking application with cashier and automated teller machine functionality using the B formal method. The complete development, including the frontend, is specified and refined to an executable application in Atelier B. Refinement between specifications and their implementations is proved mechanically. At 2'324 lines of code and 1'397 proof obligations the B Bank is not a toy; yet it is still understandable in its details.

This paper aims at filling the gap in the B literature between small text book examples and the bird's eye view reports on large industrial-size developments. Furthermore, we use B from top to bottom, fully prove all obligations, and make the case study available online.

1. Introduction

We develop a simple banking application with cashier and automated teller machine (ATM) functionality. The cashier can register new customers, create accounts for them, and accept deposits. At the ATM, the customer can withdraw money, query the balance, and change her secret personal identification number (PIN).

We illustrate the combination of structured and formal methods by using object-oriented modeling techniques in the analysis. The communication from B with the environment is exemplified through the development of base machines for persistent storage of objects, string handling, and for interfacing with the Web through HTML and the common gateway interface. The latter permits us to build a uniform graphical interface for both the cashier station and the ATM (Fig. 1). The full B code as well as the final application can be accessed at http://www.abo.fi/˜mbuechi/publications/BBank.html.

1.1. B method and tool support

The B method, invented by J.-R. Abrial [1], is a state-based method built on set theory and predicate logic and extended by generalized substitutions. Specifications and implementations are represented by abstract machines. A machine encapsulates operations and state, the latter being determined by a set of variables. Development proceeds in a layered fashion, where higher level specifications are implemented using lower level constructs. Refinement is the key notion between a specification and its implementation. Notation is quite standard; we give additional explanations where appropriate. For a thorough introduction to B, the reader may consult [12, 14, 19, 1].

Steria's Atelier B in version 3.2.2 [18] has been used in this case study. The final code has also been ported to B-Core's B-Toolkit 3.4.2 [2]. Both tools include analyzers, animators, proof obligation generators, proof tools, code translators, and documentation facilities. Atelier B, which until summer '98 only came with French manuals, has captured most of the French market; the B-Toolkit is predominant in other countries. Because of this strong partitioning of the user base, the differences between the two tolls are not widely known. We briefly discuss a number of interesting disparities in the language implementations and the provided libraries in Sect. 9.

The availability of two commercial tools has lead to a relatively wide acceptance of B in industry as well as in

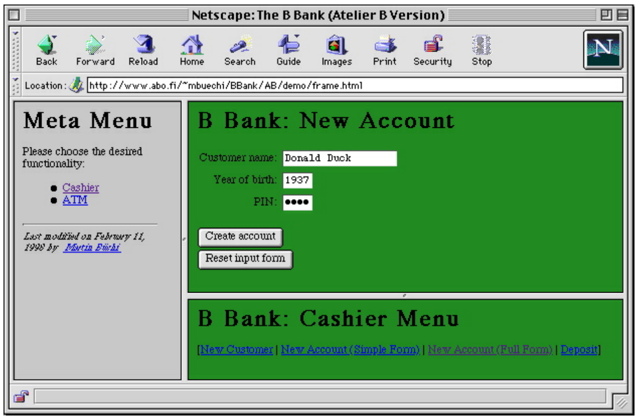

Figure 1. Screenshot of the final application

CS curricula. The distribution of the errors found in both tools —good analysis and proofs, but relatively immature backends—, however, indicates that they have mostly been used for specification purposes only.

1.2. Overview of the development

We start out by rewriting the informal requirements in structured plain English. In this exposition, we skip this and proceed directly to a semi-formal object model using the Unified Modeling Language (UML) [4] in Sect. 2. This intermediate step divides the gap between requirement specification and B machine. Before we dive into formal specifications, we give an overview of the system design in Sect. 3.

Our initial B specification *Bank* in Sect. 4 encompasses the basic functionality on an abstract level. This is the machine which we animate to find design errors and in which we prove the relevant properties of our system. On top we build a robust graphical user interface. Underneath, we build a foundation for objects and persistent storage. This combination of top-down and bottom up development, where we start with a machine describing the functionality on an abstract level is very common in B.

On top of the central machine *Bank* we construct a robust interface *RobustBank* with trivial preconditions (type information only) and error reporting in Sect. 5.

Only the algorithmic part of a program can be directly implemented in B. Communication with the environment is performed using base machines which give a B representation of a resource. Base machines are specified in B, but hand coded in C or another classical language for which a compiler exists. We illustrate the development of a base machine for interfacing with the Web in Sect. 6.

Using the robust interface and a base machine wrapping a common-gateway interface library, we build a Web-based graphical user interface for our development in Sect. 7.

The implementation of *RobustBank* in Sect. 8 shows the principle of structural refinement. A machine is refined to an implementation, which is based on a number of more basic machines. These lower level constructs are themselves implemented using more basic or base machines. Figure 2 gives an overview of the development process, including section numbers for quick reference.

As mentioned above, Sect. 9 discusses some aspects of the B-Toolkit port, which can also be found online. Note that in this port —unlike in the original Atelier B implementation— we have not proved all obligations.

In Sect. 10 we address the question of proofs in B. What types of properties about our system can we prove within B? Section 11 gives some metrics of the project, discusses what we have gained from the use of formal methods, points to related work and draws the conclusions.

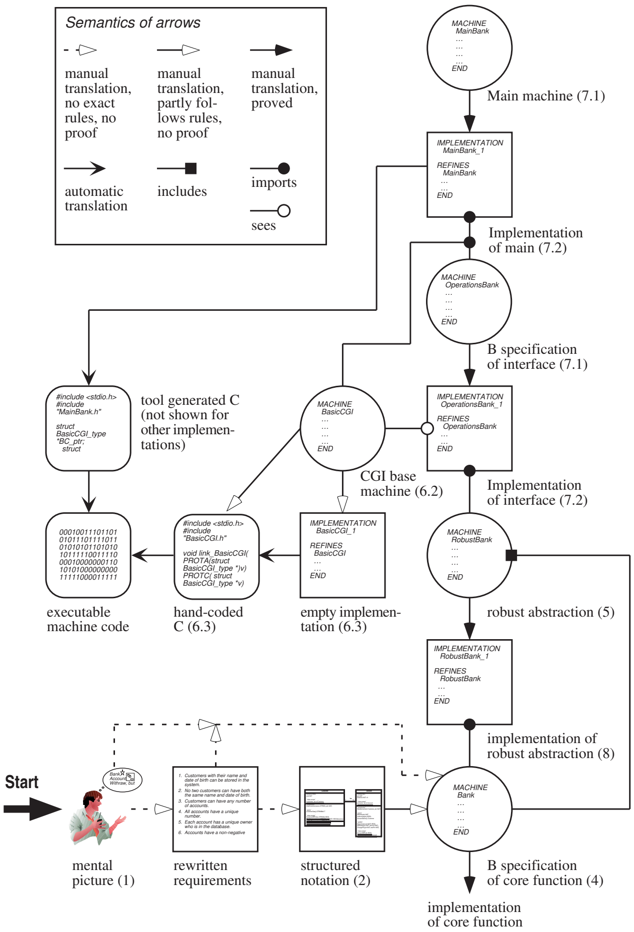

Figure 2. Overview of the development process

2. Structured Models

During analysis we produce a structured model from the problem statement. The structured notations help to produce specifications which are correct with respect to the user requirements. This step is performed manually, following some heuristics. It lacks formal rules and, therefore, also a proof. This step could be skipped, going directly to a B specification. However, this would be a rather big step and, hence, also a source of errors. The benefits of integrating formal and structured methods are becoming recognized by many researchers [7, 9]. The IEC 65A 122 standard for safety-critical software also recommends the use of both structured and formal methods for software of the highest integrity level [10].

For our case study only static structure diagrams are relevant. The large amount of information captured in static structure diagrams is widely acknowledged [11].

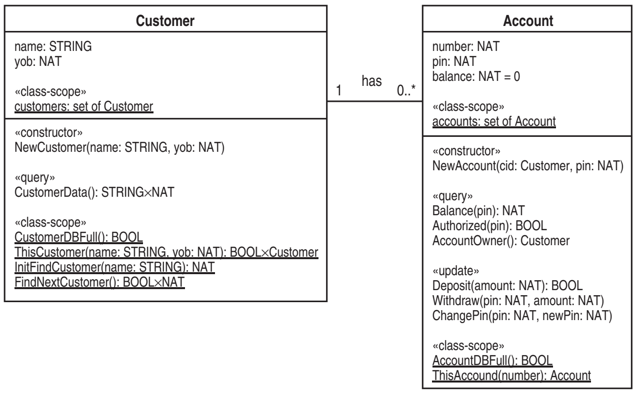

Figure 3. Object model

2.1. Class diagrams

The UML [4] class diagram shows the static data structure of the real-world system and organizes it into workable pieces. It describes real-world object classes and their relationships to each other.

In our case we identify *Customer* and *Account* as object classes (Fig. 3). Our simple data dictionary defines them as follows: A *Customer* is the holder of zero or more accounts. An *Account* is an entity in our bank against which transactions can be made.

Next, we enumerate the attributes, i.e., the properties and the operations of the individual classes. Each *Customer* has a *name* and a year of birth (*yob*). In addition to the instance-scope attributes, of which each instance has its own copy, class *Customer* has the class-scope attribute *customers*, the set of all customers in the system. Class-scope members are underlined in the diagram. The class *Customer* has a single constructor and a single query function. The product type *STRING*×*NAT* indicates that *CustomerData* returns both the name and the year of birth of a customer. The class-scope operations permit to inquire whether the database is full, to retrieve a customer, and to find all customers with a certain name.

Each *Account* has a *number*, a *pin*, and a *balance* which is initially 0. Class *Account* also has a class-scope attribute *accounts*, the set of all accounts in the system. *Account* has a single constructor. The query functions permit to query the balance, check whether a PIN is valid, and get the owner of an account. The update operations provide functionality to make a deposit or withdrawal and to change the PIN. The class-scope operations can be used to check whether the database is full and to retrieve an account by its number.

Not expressed in the UML diagram is the fact that *name* and *yob* together form a key for *Customer*, that is, there cannot exist more than one customer with a given name and year of birth. Likewise, *number* is a key for *Account*.

Finally we catalog the associations, i.e., the dependencies between objects. A customer may have any number of accounts; each account has exactly one owner.

3. System Design

From the analysis of the system we progress to system design, the high-level strategy for solving the problem and building a solution. During system design, we partition the system into subsystems, decide on what external hard- and software components we use, and establish a conceptual policy.

We start with the middle layer capturing the desired functionality (Fig. 4). On top of the basic functionality layer we build a robust abstraction which performs error checking and returns error codes, rather than relying on non-trivial preconditions. The top layer gives us the desired system in the form of a Web interface as defined by the problem statement. Its second foundation is the common gateway interface (CGI) subsystem, which consists of an off-the-shelf CGI library with a B wrapper. The CGI subsystem interfaces to the server, running on the same central server. The latter communicates via TCP/IP with the Web browsers running on the ATM and the cashier's terminal.

In order to implement the core data, we build a subsystem which supports persistent objects and strings. The former in turn is based on two more basic subsystems, one giving us objects and a second one providing access to the file system. The two bottom layers represent the available resources, namely the hardware and the operating system.

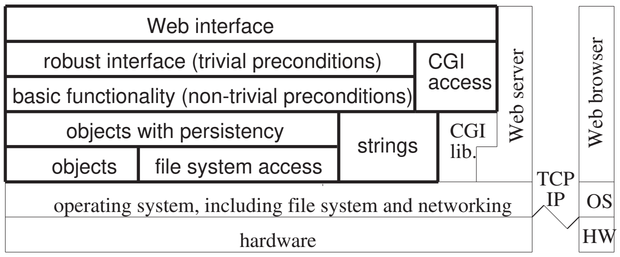

Figure 4. System design

4. B Specification

We continue by translating the structured model to a B specification, giving the middle layer of basic functionality. First we translate our object model according to fixed rules which gives the state space of the machine and the signature of the operations. Then we add the initialization and the specification of the operations with help of the rewritten requirements.

4.1. State space

The specification construct is called *MACHINE* in B. It is followed by the machine's name and its parameters, which are instantiated when the machine is included in another specification or imported in the implementation. The types of the machine parameters are given as predicates in the *CONSTRAINTS* section. The *SEES* section lists constructs imported in read-only mode. The *DEFINITIONS* section introduces textual macros. For each object class we introduce a set containing all possible instances. This gives us the sets *CUSTOMER* and *ACCOUNT*.

MACHINE
Bank(*maxCustomers*, *maxAccounts*)

CONSTRAINTS
$maxCustomers \in 1 \mathinner{..} 100000 \wedge maxAccounts \in 1 \mathinner{..} 200000$

SEES
StrTokenType

DEFINITIONS
$CUSTOMER == 0 \mathinner{..} maxCustomers\text{-}1;$
$ACCOUNT == 0 \mathinner{..} maxAccounts\text{-}1$

Variables are introduced in the *VARIABLES* section and are both typed and related to each other in the *INVARIANT* section. We introduce the two class-scope variables *customers* ($\subseteq$ *CUSTOMER)* and *accounts* ($\subseteq$ *ACCOUNT)*, denoting the sets of customers and accounts in the system.

Mandatory attributes are modeled as total functions ('$\rightarrow$') from the set of actual customers, respectively accounts, to the value of the attribute. This gives us variables *customerName*, *customerYob*, *accountNumber*, *accountPin*, and *accountBalance*. Identifiers, e.g., *accountNumber* and the relation direct product ('$\otimes$') of *customerName* and *customerYob*, are injections ('$\rightarrowtail$'), capturing the fact that no two objects with the same values for these attributes can exist.

The association *has* can be translated to the total function *accountOwner* from *accounts* to *customer*. It is a function, rather than a general relation, because the maximum multiplicity of *Customer* is 1; furthermore, it is total because the minimum multiplicity is also 1.

VARIABLES
customers, *customerName*, *customerYob*,
accounts, *accountNumber*, *accountPin*, *accountBalance*,
accountOwner, *foundCustomers*

CONCRETE_VARIABLES
fileOpen

INVARIANT
$customers \subseteq CUSTOMER \wedge$
$customerName \in customers \rightarrow STRTOKEN \wedge$
$customerYob \in customers \rightarrow \mathbf{NAT} \wedge$
$customerName \otimes customerYob \in$
$\quad customers \rightarrowtail (STRTOKEN \times \mathbf{NAT}) \wedge$
$accounts \subseteq ACCOUNT \wedge$
$accountNumber \in accounts \rightarrowtail \mathbf{NAT} \wedge$
$accountPin \in accounts \rightarrow \mathbf{NAT} \wedge$
$accountBalance \in accounts \rightarrow \mathbf{NAT} \wedge$
$accountOwner \in accounts \rightarrow customers \wedge$
$fileOpen \in \mathbf{BOOL} \wedge foundCustomers \subseteq customers$

4.2. Functionality

At startup, there are no customers or accounts in the database. Hence, the initialization assigns the empty set to the sets *customers* and *accounts* and, therefore, also to the functions representing the attributes and relations. Machines s use parallel assignment ('$||$') both in the initialization and the operations.

INITIALISATION
$customers := \emptyset \mid\mid customerName := \emptyset \mid\mid customerYob := \emptyset \mid\mid$
$\ldots$

The operation *NewCustomer* creates a new customer and sets its *name* and *yob* attributes. To concentrate on the actual functionality, rather than error checking and reporting, the precondition not only gives a type to the parameters, but also states that there must not be any customer with both the same name and year of birth present in the database, and that the database must not be full. If these conditions are met, an arbitrary new customer object is selected using the *ANY*-clause. This object is added to *customers* and its *name* and *yob* attributes are set. Note that *customerName(newCustomer)* := *name* is an abbreviation for $customerName := customerName \cup \{newCustomer \mapsto name\}$.

OPERATIONS
NewCustomer(*name*, *yob*) =
PRE
$name \in STRTOKEN \wedge yob \in \mathbf{NAT} \wedge$
$(name, yob) \notin \mathbf{ran}(customerName \otimes customerYob) \wedge$
$customers \neq CUSTOMER \wedge fileOpen = \mathbf{TRUE}$
THEN
ANY *newCustomer* **WHERE**
$newCustomer \in CUSTOMER \text{ - } customers$
THEN
$customers := customers \cup \{newCustomer\} \mid\mid$
$customerName(newCustomer) := name \mid\mid$
$customerYob(newCustomer) := yob$
END
END;

Any client of *NewCustomer* must be able to verify the precondition. For this purpose we introduce the operation

ThisCustomer (not shown), which checks whether a customer with given name and year of birth already exists, and operation *CustomerDBFull* (not shown) to inquire whether the database is full. In most applications, databases are assumed to have infinite capacity and their administrators are supposed to add secondary storage as the available storage gets filled. However, the number of incidents of database and buffer overflow problems clearly shows that we should not trust this assumption in a safety-critical system.

Operation *CustomerData* is an instance-scope operation which returns the name and year of birth of a customer. Self, the identity of the object, is modeled as the normal parameter *cid*. The second conjunct of the precondition, $cid \in CUSTOMER$, although logically implied by the first conjunct, is required by the type checker. The identity of a customer object is given by *ThisCustomer* (not shown).

$name, yob \leftarrow$ **CustomerData**(cid) =
  PRE
    $cid \in customers \wedge cid \in CUSTOMER \wedge fileOpen =$ **TRUE**
  THEN
    $name := customerName(cid) \; || \; yob := customerYob(cid)$
  END;

The triple *NewAccount*, *AccountDBFull*, and *ThisAccount* represents the corresponding operations on accounts. The operation *Balance* requires the account's PIN. The PIN is only used in the precondition to verify the legitimacy of the client, but not in the body of the operation. Specifying that the entered PIN must match the stored PIN in the precondition, forces us to prove that *Balance* is always called with the correct PIN.

$bal \leftarrow$ **Balance**(aid, pin) =
  PRE
    $aid \in accounts \wedge aid \in ACCOUNT \wedge pin \in$ **NAT** $\wedge$
    $accountPin(aid) = pin \;\wedge fileOpen =$ **TRUE**
  THEN
    $bal := accountBalance(aid)$
  END;

  ...

END

We can enforce that withdrawals and balance queries can only be performed with the correct PIN. On the other hand, secrecy not being a property of behaviors, we cannot ensure it in B. Nothing can prevent an implementation to output secret pins onto a device, the state of which is not captured by the B specification.

The two final operations *Open* and *Close* (not shown) concern persistency. An image of the set of customers, accounts, and strings (see below) is stored on disk between program runs. *Open* is meant to read an arbitrary state satisfying the invariant from secondary storage. *Close* writes the current state of the machine to the three files. In B we can only reason about a single program run. That is, we cannot express that externalizing the state at the end of a run and internalizing it again in the beginning of the next run results in the machine being in the old state.

Machine *Bank*, encapsulating the basic functionality, is animated to test whether it satisfies the stated requirements and also to check whether the latter are what we actually want. The proofs for this machine ascertain that the initialization establishes the invariant and that the operations preserve it. However, the step from the rewritten requirements and the structured notation to the formal B specification cannot be formally proved.

5. Robust Abstraction

The initial machine *Bank* uses non-trivial preconditions rather than elaborate error handling to keep the specification simple. We could build a graphical user interface directly upon it. However, we opt for an intermediate layer, providing roughly the same functionality but with verification of parameters. Herewith, we effectively split up the task at hand. We avoid duplication of parameter checking for transactions which could be performed in different manners. Robustness also means that the operations could be safely called from outside B.

Inclusion, as given by the *INCLUDES* clause, can for the purpose of this paper be understood as simple textual inclusion with read-only access on variables of the included machines. Variables of included machines can only be changed by calling operations of the included machine. This restriction makes the proofs more compositional. Included machines have to be instantiated, that is, given values for their machine parameters. We rename the included machine *Bank* to '*BK.*' for better traceability of indentifiers. The *SETS* clause introduces enumerated sets.

MACHINE
  $RobustBank(maxCustomers, maxAccounts)$

CONSTRAINTS
  $maxCustomers \in 1 \,..\, 100000 \wedge maxAccounts \in 1 \,..\, 200000$

INCLUDES
  $BK.Bank(maxCustomers, maxAccounts)$

SEES
  StrTokenType

DEFINITIONS
  $CUSTOMER == 0 \,..\, maxCustomers\text{-}1;$
  $ACCOUNT == 0 \,..\, maxAccounts\text{-}1$

SETS
  $RESULT = \{success, dbFull, dbError,$
    $customerAlreadyPresent, \ldots\}$

```
OPERATIONS
 result ← RobustNewCustomer(name, yob) =
  PRE name ∈ STRTOKEN ∧ yob ∈ NAT THEN
   IF BK.fileOpen = TRUE THEN
    IF BK.customers ≠ CUSTOMER THEN
     IF (name,yob) ∉
         ran(BK.customerName ⊗ BK.customerYob)
      THEN
       result := success || BK.NewCustomer(name, yob)
     ELSE result := customerAlreadyPresent
     END
    ELSE result := dbFull
    END
   ELSE result := dbError
   END
  END;
  ...
END
```

6. Base Machines

Before we can build a graphical user interface on top of the robust abstraction, we need to build support for the desired input and output mechanisms. A program consists of two parts: computation and interaction with the environment. The algorithmic aspects of a program can be expressed in B, whereas the input and output must be coded in a traditional language.

The B development can be interfaced in two ways with its environment. In the first approach so-called base machines which encapsulate a resource are used. They are specified in B to make them available to other constructs, but implemented directly in the desired classical language, e.g., C or Ada. The alternative solution is to use B to create a service subsystem, a subroutine library, and write the main program which interfaces with the environment and calls the B subsystem in a classical programming language.

In many industrial applications, especially in those that build on existing components, B is only used to create the most safety-critical algorithmic part in the middle, building on well-tested databases for persistent storage and complex graphical user interfaces.

6.1. Strings in Atelier B

Atelier B has a type *STRING* for constant character chains. *STRING* can be used for passing a message like "Hello world" to a terminal output machine or, in our case, to pass the names of the dump files. However, there is no support for non-literal strings as needed for customers' names. Atelier B does not permit objects of variable length, such as strings, to be passed between operations. Because there is no support for constant-length strings either, we are forced to use tokens as references to the actual strings, which are stored in a base machine.

We define a type of string tokens *STRTOKEN* as an abstract set in machine *StrTokenType*. We also define a base machine *BasicString* which stores strings and issues tokens. Any I/O machine, such as *BasicCGI* discussed below, has to communicate with *BasicString* 'behind the scenes.' Upon a string input request, the I/O machine inputs the string and passes it to *BasicString*. The latter stores the string and issues a token. The I/O machine then returns this token to its client. For an output request, a client gives a string token to the output machine. The output machine gets the corresponding string from *BasicString* and outputs it. A more detailed treatment of approaches to string handling in Atelier B can be found in [6].

6.2. Machine *BasicCGI*

In order to input and output data to the Web, we need a machine to access the common gateway interface (CGI), which we call *BasicCGI* (Fig. 6.2). CGI is a standard for interfacing external applications with information servers, such as Web servers. A plain hypertext markup language (HTML) document that the Web daemon retrieves is static, which means it exists in a constant state: a text file that doesn't change. A CGI program, on the other hand, is executed in real-time, so that it can output dynamic information. The user fills out a form in the browser and sends the data to the server which executes the CGI program. The CGI program processes the input, modifies the local database, and generates an output which is sent back to the user's browser for display.

Operation *ReadNat* inputs a natural number value of a field, designated by its name, from a form as presented to a CGI program by a Web server. Since the user can enter an arbitrary number into a given field, we can only assure that *num* is a natural number. The browser, the server, and the connection between them being outside the realm of our specification, we cannot specify that the reported value is actually the one entered by the user. An implementation that would always return 0 —independently of the user's input— would, therefore, be formally correct. The result code is supposed to indicate whether the Web server has reported an error. Likewise, an implementation which always reports an error would be formally correct.

Similarly to *ReadNat* there is an operation to input strings and operations for outputting a new document in response to the user's request. The latter are simply specified as *skip*, as exemplified by *WriteString*. Since we do not model what is sent to, or even displayed by the browser, no stronger specification is possible.

MACHINE
BasicCGI

SEES
StrTokenType

OPERATIONS

$status, num \leftarrow$ **ReadNat**$(name)$ =
  PRE $name \in STRING$ **THEN**
    $status :\in$ **BOOL** $||$ $num :\in$ **NAT**
  END;

$status, str \leftarrow$ **ReadTokenString**$(name, maxLength)$ =
  PRE $name \in STRING \wedge maxLength \in$ **NAT THEN**
    $status :\in$ **BOOL** $||$ $str :\in STRTOKEN$
  END;

  . . .

END

6.3. Implementing *BasicCGI*

The C code for *BasicCGI* is written by hand. Rather than implementing CGI access from scratch we build upon the public domain CGI library cgic [5]. This library provides for comfortable parsing of form input.

Since we prove that all calls to operations of *BasicCGI* satisfy the respective precondition, we do not need to check the preconditions in the C code. Correctness of the hand-coded implementation with respect to its B specification has to be asserted using normal verification techniques, e.g., testing and third party code inspection.

7. User Interface

The user interface presents an entry mask to the user, parses the input with the help of *BasicCGI*, sends the request to the robust interface *RobustBank*, and presents the results, using again *BasicCGI*. Static information, such as the input forms, is kept in normal HTML files.

The cashier is presented with a menu on the bottom of her terminal, from which she can choose a form to enter a new customer, create a new account for an existing customer, or make a deposit. In the 'new customer' form (similar to Fig. 1), the cashier enters the name and year of birth of the customer and clicks on a button to send the data to the CGI application. In response, the cashier gets a screen saying that the operation has succeeded or that an error has occurred. After reading the output message, the cashier clicks on another menu choice.

When creating a new account, the cashier has the option of entering both the customer's name and year of birth or only the name. If there is only one customer with the given name in the system, an account is created. On the other hand, if there is more than one customer with this name, the cashier is presented with a list. She then simply clicks on the desired customer to create the account. In the latter case, the links contain all the parameters, which would usually be entered into the form by the cashier. For example for customer 'Garfield', born in '1978', and PIN '2001' the URL of the link would be 'http://. . ./cgi-bin/MainBank?command=1&name=Garfield&yob=1978&pin=2001'. The CGI program doesn't have to store any temporary information.

'Deposit' leads to simple one-step interaction sequences like 'new customer'. The ATM transactions follow the same pattern. We create a single CGI program to handle all user requests and employ a hidden field 'command' in the HTML forms to select the desired operation —e.g., 'command=1' selects transaction 'new account', as shown in the above example.

7.1. Main program

To keep the size of the different constructs small, we divide the user interface into two machines. Machine *MainBank* contains the main program with the single operation *main* specified as *skip*. Since the persistent state, existing beyond a single program run, cannot be modeled, *skip* is in fact the only reasonable specification for a main program. The intention is for *MainBank* to read the 'command' field and call the corresponding operation of machine *OperationsBank*, which does the actual processing.

We do not duplicate the state on the user interface level in *OperationsBank* as we do not want to perform any reasoning on this level. Hence, the specifications of the transaction operations are simply skip.

7.2. Implementations

The implementation *MainBank_1* first opens the database. Then it reads the value of the 'command' input field, calls the selected operation, and closes the database.

The implementation *OperationsBank_1* imports the robust interface of the database. The operation *NewCustomer* first outputs the header of the result screen, which is independent of the outcome of the operation. Then it reads the value of the 'name' field, calls *RobustNewCustomer* and presents the result.

Implementation constructs start with the keyword *IMPLEMENTATION*, followed by the implementations name, which in our case is the name of the corresponding specification with suffix '_1'. The name of the implemented machine is given in the *REFINES* clause. The imports clause instantiates auxiliary constructs, comparable to *INCLUDES* on the specification level. The *SEES* clause allows a construct to be visible by several implementations. It requires that the listed constructs are imported and provided with values for their parameters by another implementation. For example in our case, *BasicCGI* is also used by *MainBank_1*. Assignment from operation return values is denoted by '←'.

```
IMPLEMENTATION
  OperationsBank_1
REFINES
  OperationsBank
IMPORTS
  RB.RobustBank(100, 200)
SEES
  BC.BasicCGI
DEFINITIONS
  CASHIER_HEADER(title) == HEADER(title, "#228B22");
  HEADER(title,color) == (
    BC.WriteLiteralContentType("text/html");
    BC.WriteLiteralString("<HTML><HEAD><TITLE>
       B Bank: ");
    BC.WriteLiteralString(title);
    BC.WriteLiteralString("</TITLE></HEAD>\n");
    . . .
  );
  FOOTER == . . .
  NAME_LEN == 256
OPERATIONS
  NewCustomer =
    VAR st, name, yob, result IN
      CASHIER_HEADER("New Customer");
      st, name ← BC.ReadTokenString("name", NAME_LEN);
      IF st = TRUE THEN
        st, yob ← BC.ReadNat("yob");
        IF st = TRUE THEN
          result ← RB.RobustNewCustomer(name, yob);
          CASE result OF
            EITHER success THEN
              BC.WriteLiteralString("<P>Customer ");
              BC.WriteLatin1TokenString(name);
              BC.WriteLiteralString(" ("); BC.WriteNat(yob);
              BC.WriteLiteralString(") has been added.</P>")
            OR customerAlreadyPresent THEN
              . . .
            END
          END
        ELSE BC.WriteLiteralString(
            "<P>Could not get year of birth.</P>")
        END
      ELSE
        BC.WriteLiteralString("<P>Could not get name.</P>")
      END;
      FOOTER
    END;
  . . .
END
```

8. Implementation of the Robust Abstraction

The missing piece is the implementation of the robust layer *RobustBank*. The robust operations first check whether the parameters and the current state satisfy the preconditions of the corresponding operations in *Bank* and then call them, or report an error if the conditions do not hold. The implementation is almost identical to the specification (Sect. 5).

The implementation of *Bank* follows already familiar patterns. A base machine for interfacing with the file system has to be constructed analogously to *BasicCGI*. Rather than implementing *Bank* directly on this base machine and a library machine for arrays, we build a generic machine for persistent objects. For brevity's sake, we do not discuss the implementation of *Bank* in any more detail. The interested reader can find the details online.

9. B Toolkit Implementation

At the present, there exist two toolkits which support the B method. Atelier B [18] from Steria in France and the B-Toolkit [2] from B-Core. Because of the strong partitioning of the user base, the differences of the two tools are not widely known. In the rest of this section, we try to convey an overview of the differences in the supported subsets of B and in the supplied libraries. A more detailed account of the variances can be found in [6]. The B method and notation has been first introduced in 1986 by Abrial and has evolved over time. Whereas the semantics of B has from the very beginning been rigorously defined, some more syntactic aspects have not been fixed well enough, resulting in some subtle differences between the two tool implementations.

Some syntactic differences can be compensated for by simple rewrites. For example, in Atelier B machine parameters are repeated in refinements and implementations whereas in the B-Toolkit they are not.

Unfortunately, there exist also a number of more serious language differences that can seriously complicate ports in either direction, as illustrated by the following examples.

- To import several copies of a machine, as for example *Object* into *Bank_1*, dot renaming is used in Atelier B. The B-Toolkit, on the other hand, does not support dot renaming in implementations. Hence, textual copies must be made which also requires identical proofs to be performed on each copy.

- In Atelier-B, the type *STRING* can only be used for string constants. It is impossible to pass other strings between constructs. In the B-Toolkit, the type *STRING* encompasses both constant and variable strings. It can be employed to pass strings between constructs.
- Loops cannot be nested in the B-Toolkit.

Unlike some programming languages, B does not define a standard library. Both Atelier B and the B-Toolkit come with a set of library constructs which provide for terminal I/O and storage in form of arrays, sequences, and sets. Although the two libraries provide roughly the same functionality, they are not compatible.

10. Proofs

Mechanical proofs are one of the key aspects of tool-supported use of formal methods. In B, we prove that all operations maintain the invariants of the machine, respectively implementation, and that implementations are refinements of their specifications. The tool generated the proof obligations and in our case proved a large part of the obligations automatically. The remaining obligations are discharged in an interactive prover.

What haven't we proved? We haven't proved that the specification corresponds to the informal requirements; especially, that we have captured all requirements as invariants. Furthermore, we haven't proved that the hand-coded base machines actually satisfy their formal specification. We are also at the mercy of the tool (Atelier B), the C compiler, and the machine with its operating system on which our application runs.

In conclusion, the many unprovable and unproved aspects even of a formal development in B are a clear sign, that good engineering practices, including animation, peer code review, and testing, are also important in a 'proved' development.

11. Discussion

11.1. Metrics

Figure 5 provides some metrics of the development. The empty implementations of the base machines, the hand-coded C sources, and the HTML pages are not included.

11.2. What have we gained from the tool-supported use of formal methods?

Using B machines, we have been able to give a concise specification —the "what"— without being distracted by implementations issues —the "how". Without tool support

Machines

Machine	length in lines	obvious proof obligations	proof obligations	auto proved
MainBank	9	3	0	100 %
OperationsBank	49	19	0	100 %
RobustBank	239	101	10	100 %
Bank	362	394	49	95 %
Object	207	125	17	100 %
BasicFile	102	26	0	100 %
BasicString	98	41	6	100 %
BasicCGI	72	15	0	100 %
StrTokenType	14	1	0	100 %
Total	1152	725	82	98 %

Implementations (without base machines)

Implementation	length in lines	obvious proof obligations	proof obligations	auto proved
MainBank_1	52	16	4	100 %
OperationsBank_1	335	1040	290	99 %
RobustBank_1	206	856	27	85 %
Bank_1	362	643	761	71 %
Object_1	207	298	231	70 %
StrTokenType_1	10	3	2	100 %
Total	1172	2856	1315	77 %

Figure 5. Statistics of the Development

and good automation of the prover, generating and proving 1'397 proof obligations would have been impossible. About 20 of the obligations that were not discharged automatically turned out to be false, indicating errors that were subsequently corrected. In this light, proofs can be viewed as a form of debugging that starts from the beginning of specification and implementation.

The layered development, and the possibility to use intermediate refinements, enabled us to carry out the refinement in small manageable steps. This is important because not everything is captured by the refinement proofs. For example, writing a wrong message to the screen is also a correct refinement of *skip*, which usually models output if the state of the screen is not part of the specification.

The tools also allowed us to animate the specifications, thus reducing the impatience which often lures designers into going to 'real' code too early. Mechanically verified refinement forced us to keep specifications and implementations consistent, which has proven useful when explaining the case study to people. We would also expect it to be an invaluable aid in maintenance. When changes are made, the tool regenerates the proof obligations and requires all changed or new obligations to be mechanically verified. This process conveys more confidence than the typical 'all-the-old-things-should-still-work' handwaving.

11.3. Related Work

The B Book [1] contains a much smaller example of a database application. The database example as well as an ATM case study are included in the Atelier B distribution. The documentation of the ATM, which is in French only, provides an exemplary requirement specifications, a tractability matrix, and a set of test scenarios. On the other hand, it lacks a description of the construction process as well as a detailed explanation of the produced code. The ATM relies on a Tcl/Tk graphical interface as main program and delegates more work to unverified base machines.

The combination of B and OMT [15] object models, the predecessors of UML class models, has been pioneered by Lano [13, 12]. Different translations of object diagrams into B have been proposed [8, 17]. A simple translation of statecharts to B is given by Lano [13]. A more thorough treatment can be found in Sekerinski [16]. A comprehensive bibliography of publications on B is maintained by the B users group [3].

Z, VDM, their object-oriented extensions Object-Z and VDM++, and RAISE would have been possible alternatives to B. Z can be viewed as the predecessor of B. It is based on a similar formulation of set theory. Z is mostly limited to specification only. It lacks the advanced modularization and structuring mechanisms of B. Z uses a semi-graphical schema notation. VDM is based on the logic of partial functions. State changes are expressed as post-conditions which implicitly include the invariant. Like B, VDM can be used from specification down to implementation. For both Z and VDM their exist object-oriented extensions. RAISE is another extension of VDM with concurrency and axiomatic and imperative styles. A full bibliography and links to tools for the aforementioned methods can be found from the virtual library of formal methods at http://www.comlab.ox.ac.uk/archive/formal-methods/.

Our main reasons for choosing B was the full tool support, especially in automated proving, and the best support for modularization and layered developments.

11.4. Summary & Conclusions

We have built a simple banking application with a Web-based front-end using a combination of structured and formal methods. Stepwise refinement allowed us to go from a specification to an implementation, which was automatically translated to C code.

By proving refinement between the specification and its implementation, we ensured that the implementation conforms to the specification. The use of tools allowed us to automatically generate proof obligations, 83 % of which were proved automatically. The remaining obligations were discharged interactively.

Our goal is to illustrate a medium-sized case study in enough detail, so the reader can comprehend it in detail. By making the full B code and the final application available online we try to share our whole experience.

Acknowledgments We would like to thank Emil Sekerinski, Marina Waldén, Kaisa Sere, Ralph Back, and Wolfgang Weck for a number of discussions on the topic of this paper. Corinne Givois and Denis Sabatier of Steria and Ib Søerensen of B-Core provided valuable help during our project. The referees' comments are gratefully acknowledged.

References

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

[2] B-Core. *B-Toolkit*. England, 1995.

[3] B User Group. Online bibliography of the B method, 1998. http://estas1.inrets.fr:8001/ESTAS/BUG/WWW/BUGhome/BUGhome.html.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1998. http://www.rational.com/uml/.

[5] T. Boutell. cgic: an ANSI C library for CGI programming. http://www.boutell.com/cgic/.

[6] M. Büchi. The B Bank. In *Program Development by Refinement: Case Studies Using the B Method*, chapter 4, pages 115–180. Springer Verlag, 1998. http://www.abo.fi/~mbuechi/publications/BBook.html.

[7] D. Coleman, F. Hayes, and S. Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1), 1992.

[8] P. Facon et al. Mapping object diagrams into B specifications. In *Methods Integration Workshop*, pages 23–41, 1996.

[9] J. V. Hill. *Microprocessor Based Protection Systems*. Elsevier, 1991.

[10] IEC. Software for computers in the application of industrial safety-related systems, 1992. IEC 65A 122.

[11] C. B. Jones. A rigorous approach to formal methods. *IEEE Computer*, pages 20–21, April 1996.

[12] K. Lano. *The B Language and Method: A guide to Practical Formal Development*. Springer Verlag London, 1996.

[13] K. Lano. Integrating formal and structured methods in object-oriented system development. In S. Goldsack and S. Kent, editors, *Formal Methods and Object Technology*. Springer Verlag, 1996.

[14] K. Lano and H. Haughton. *Specification in B: An Introduction Using the B Toolkit*. Imperial College Press, London, 1996.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, and F. Eddy. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[16] E. Sekerinski. Statecharts in B. In *Proceedings of the second B conference*. LNCS, Springer Verlag, April 1998.

[17] R. Shore. Object-oriented modelling in B. In *Proceedingsof 1st Conference on the B method*, pages 133–154, 1996.

[18] Stéria Méditerranée. *Atelier-B*. France, 1996. http://www.atelierb.societe.com.

[19] J. B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.