Class notes

Introduction to the B-Method

Maximiliano Cristiá

Computational Science 3

Bachelor in Computer Science

Faculty of Science, Technology and Medicine

University of Luxembourg

These class notes have been written during a visit to University of Luxembourg from February until June, 2023.

Contents

We will introduce the basics of the B-Method by means of an example where a set of user requirements are formalized or modeled as a B specification. However, we start with some preliminaries about the B-Method.

1 Preliminaries about B

B specifications take the form of state machines described by means of predicate logic and set theory. A B state machine consists of:

1. *A set of states*. This set of states is given by defining *state variables*. Each state variable has a name and a type. The set of states is given by all possible values of all the state variables. For example, if we have two state variables, x whose type is $\mathbb{Z}$ and y whose type is $\{ok, no\}$, then some of the possible states of this state machine can be written in mathematics as follows (this is NOT B notation):

   $$(x = 1, y = ok), (x = 4, y = ok), (x = 4, y = no), (x = 7, y = ok), \ldots$$

   which we would draw as follows if we were using the classic graphic notation for state machines:

   Two states are equal if all the state variables have the same value. That is, every time one of the state variable changes its value, we have a new state.

2. *A set of state operations*. State operations are described by means of predicate logic and set theory. There are two classes of state operations: *state transitions* and *state queries*. State transitions make the state machine to go from one state to another, possibly taking some input values and producing some output values. State queries do not change the current state but only produce some output values, possible taking some input values.

   If x is a state variable and a state transition changes its value to another one, we will write $x := expr$ meaning that the new value of x is the one obtained by evaluating the expression $expr$.

   For example, we can say that state transition T increments the value of state variable x in 3 and leaves y unchanged. Then we can write T in mathematics as follows (this is NOT B notation):

   $$T \mathrel{\widehat{=}} [x := x + 3]$$

   Graphically, we can say that T is the transition connecting the first two states depicted above:

   In any case, T says that the state machine will move from a state to another one where the value of x is three units greater than the value of x in the first state while the value of y remains unchanged.

   When T transitions from state s_1 to state s_2 we say that s_1 is the *before state* or the *start state* and s_2 is the *next state* or *after state*.

1.1 B types

As we have said, every B variable has a type. B types are more or less like types in programming languages. For example, in Java all variables have a type. However, B types are closer to mathematics than to programming languages. If T is a B type and x is a variable, then $x \in T$ means that x is of type T.

In B we have the following types:

- *Numbers*. In B we have the type of the integer numbers, noted $\mathbb{Z}$, and the type of the natural numbers, noted $\mathbb{N}$.

  The standard arithmetic operations are available: $+$, $-$, $*$, div, mod, $>$, $<$, $\leq$ and $\geq$.

- *Enumerated types or enumerated sets*. An enumerated type or set is just an enumeration of constants. An enumerated type is introduced as follows:

  $$TYPE\_NAME = \{const_1, \ldots, const_n\}$$

  For example, if we are modeling the workflow of some dossier we may need to define the following enumerated type:

  $$STATUS = \{requested, verified, aproved\}$$

  meaning that the dossier has been *requested* but not yet *verified*, that it has been *verified* but not yet *aproved* and so on.

- *Basic types or basic sets*. A basic type is a set of values whose form or structure is unknown. A basic type is introduced by simply stating its name. For example, if we want to work with street addresses we use the name $ADDR$ to mean the set of all possible street addresses. We don't care whether street addresses are character strings or numbers or anything else.

- *Cross products*. A cross product or Cartesian product type is the product of two types. It is simply declared as follows:

  $$T \times U$$

  where T and U are two B types. An element of $T \times U$ is an ordered pair (x, y) where x is of type T and y is of type U. That is, $T \times U$ is the set of all ordered pairs (x, y) where x is an element of T and y is an element of U.

  Cross products are useful to group related characteristics of some entity. For example, we may model the personal data of people as the combination of an address and a date of birth:

  $$ADDR \times DATE$$

  Then, if we have $p \in ADDR \times DATE$ it means that p is an ordered pair whose first component is of type $ADDR$ and whose second component is of type $DATE$.

- *Powerset types*. A powerset type is the type of all subsets of a given type. It is noted as $\mathbb{P}\,T$, where T is a B type. The elements of a powerset type are sets. For example if we use:

  $$\mathbb{P}\,STATUS$$

then, possible elements of this type are: $\{\}$, $\{requested\}$, $\{verified, aproved\}$, ...

Powerset types are used all the time in B specifications. For example, the B specification of a program returning the maximum of a list of numbers is actually described with a set of numbers, and not a list of numbers. Indeed, that program will return the same result regardless of the order in which the numbers are input to it, as well as regardless of the repeated elements the list might have. For instance, the maximum of the following two lists is the same:

$[3,5,1,3]$
$[1,5,3,3]$

because both lists contain the same numbers. This is so because, essentially, these two lists can be represented by the same set: $\{5,1,3\}$. Hence, if s is the set representing the list of numbers of which we want to compute its maximum, we declare is as $s \in \mathbb{P}\mathbb{Z}$. Note that if we declare $s \in \mathbb{Z}$, it can hold only one integer number.

B provides a number of sophisticated set operators which, when used properly, can yield very concise, simple and short specifications.

Sets and set operators are at the core of the B specification language.

> Although in B sets may be infinite, in this course we will consider only finite sets.

- $X \leftrightarrow Y$ denotes the type of all *binary relations* between elements of types X and Y. It's defined as $X \leftrightarrow Y \mathrel{\widehat{=}} \mathbb{P}(X \times Y)$. Recall that a binary relation between sets X and Y is any subset of $X \times Y$. In other words, a binary relation is a set of ordered pairs.

  Hence, if we have $R \in X \leftrightarrow Y$ then we know that R is a set of ordered pairs where each first component is an element of X and each second component is an element of Y. That is, R is a set such as $\{(x_1, y_1), (x_2, y_2), \dots\}$.
- $X \nrightarrow Y$ denotes the type of all partial functions from X onto Y. A partial function is a function which might not be defined in all its domain. In B, partial functions are sets of ordered pairs. Actually we have:

  $$X \nrightarrow Y \subseteq X \leftrightarrow Y$$

  That is, every partial function is a binary relation (although not every binary relation is a partial function). More precisely:

  $$X \nrightarrow Y \mathrel{\widehat{=}} \{f \mid f \in X \leftrightarrow Y \land \forall x, y_1, y_2 . ((x, y_1) \in f \land (x, y_2) \in f \Rightarrow y_1 = y_2)\}$$

- $\text{seq}\, X$ denotes the *set* of all sequences (lists) of elements of type X. In B an element of $\text{seq}\, X$ is a partial function whose domain is the integer interval $[1, n]$ for some n (written as $1 \mathinner{..} n$ in B); n is the length of the sequence. Formally:

  $$\text{seq}(X) \mathrel{\widehat{=}} \{s \mid s \in \mathbb{N} \nrightarrow X \land \exists n . (n \in \mathbb{N} \land \text{dom}(s) = 1 \mathinner{..} n)\}$$

  Although in B sequences are sets, the language provides some *syntactic sugar* to work with them:
  - The empty list is noted $[\,]$.
  - Explicit lists can be written by enclosing their elements between square brackets: $[a_1, \dots, a_n]$.

1.2 Domain and function application

This section is to recall two simple concepts of mathematics that all you should know.

The *domain* of a binary relation is the set of the first components of the ordered pairs that belong to the relation. Formally, if $R \in X \leftrightarrow Y$ then:

$$\mathrm{dom}(R) = \{x \mid x \in X \wedge \exists y.(y \in Y \wedge (x, y) \in R)\}$$

If f is a function, then $f(x)$ denotes the *application* of f to x. Recall that if $f \in X \nrightarrow Y$, then $f(x)$ might not be defined for all $x \in X$.

2 Savings accounts – User requirements

The user requirements of the example we will use to introduce the B-Method consist of the description of the basic operations over the savings accounts of a bank. This is a simplification of a real problem as to serve as an introductory example of requirements formalization.

- Each bank customer is identified by means of his or her national identity card number (NIC).
- Each bank customer can have at most one savings account.
- Each savings account is identified with the owner's NIC.
- Each savings account will hold its current balance.
- When a savings account is just opened its balance is zero.
- Once a savings account is opened it is possible to deposit a positive amount of money; it is possible to withdraw a positive amount of money but no greater than the current balance account; and it is possible to consult the current balance.
- A savings account can be closed only if its current balance is zero.
- The balance of any account cannot be negative.

3 Savings accounts – B specification

Now, we will give the B specification corresponding to the requirements given above. We will proceed as follows:

1. We will define a type for each state variable
2. We will define the set of states of the state machine
3. We will specify the operations of the state machine

The state machine is described inside a structure called MACHINE:

MACHINE *Bank*
    . . . the state machine description goes here. . .
END

As can be seen the machine has a name: *Bank*.

3.1 Types and state variables

Finding the right types for the state variables is perhaps the most important step when writing a B specification. These types must be *abstract*; that is, they should represent a mathematical object rather than a programming object. Then, in B most often the types of state variables are sets, binary relations and partial functions. This is so because in this way the specification is simple, short and easy to understand. The types of a B specification try to capture the *essence* of the thing that is being considered. For example, if the essence of a thing is to be an integer number, then the B type is $\mathbb{Z}$ rather than the typical `int` type of a programming language. In effect, if $x \in \mathbb{Z}$ then x can be *any* integer number while if the type is `int`, x can be only a number in an integer interval such as [`minint`,`maxint`]. In other words, an `int` occupies, say, four bytes, but a $\mathbb{Z}$ occupies no bytes because is a mathematical object. If x and y are of type `int` then $x+y$ might yield an unexpected result due to overflows; if their type is $\mathbb{Z}$, $x+y$ is always what you expect. When we write a specification we don't want to worry about overflows.

As another example of the difference between a B type and an implementation type, consider the name of a person. If you want to store the name of a person in a variable of a programming language, then you probably would think of a character string of some appropriate length. Further, you'd probably analyze if 40 chars is too little and if 120 is too many. In a B specification, instead, you'd use a basic type:

$NAME$

where you can store names of any length because in a B specification memory consumption doesn't matter at all. That is, in a B specification we don't think about memory consumption because we will think about that once we understand the problem at hand. In fact, if we have $n \in NAME$ we can't even talk about the size or length of n: it has no size because it belongs to a basic type and we don't know the structure or form of the elements of basic types.

So now we turn our attention to the types we need to specify the savings accounts. As we have said, the customers identify themselves with their NIC and NIC are used to identify their savings accounts. Hence, NIC are important in the requirements. However, are NIC numbers? Are they strings? Are they composed of more than one value? It doesn't matter. It only matters that a NIC identifies a person and a savings account. Then we declare a basic type called *NIC* inside the *Bank* machine as follows:

MACHINE *Bank*
SETS *NIC*
. . . the machine description continues here. . .
END

In this way we know nothing about the structure of the elements of *NIC*. Nevertheless, B allows us to build sets of *NIC* and to define binary relations and partial functions that take or return *NIC*. Besides, we can use the basic logic operations on the elements of *NIC*. For example, assuming $n, m \in NIC$; $A, B \in \mathbb{P}\,NIC$:

$n = m$
$n \neq m$
$n \in A$
$A \subseteq B$

are all valid B predicates.

It's very important to note that $n = m$ is a predicate, not an assignment. Then, we can write $n = m$ or $m = n$.

The user requirements say that we need to store the balance of each account and that money can be deposited and withdrawn from them. Hence, we need to talk about amounts of money in our specification. At first sight an amount of money looks like a real number. But, on the other hand, we know that the minimal subunit of Euro is 1 cent. Then, actually, an amount of money looks more like a rational number (because it always has a finite number of decimal numbers). Furthermore, we can multiply any amount by 100 thus obtaining an integer number. For instance, we may say that 1.32 euro are 132 cents of euro. Therefore, we can model any amount of money with an integer number.

Now we need to think in the sate variables of our specification. Clearly, the bank will have a number of savings accounts at any given time. For example, we may think that when the bank just opens to the public it has no savings accounts until the first customer opens his or her account. At that moment the bank has only one account whose balance is zero. So the bank transitioned from the state having no accounts to the state where there is a single account whose balance is zero. Now the owner of the account can deposit 200 euro in the account, thus making the bank to transition to a new state where there is a single account but whose balance is now 200 euro. After that, a new customer may open an account and so the bank transitions to a new state where there are two savings account one with 200 euros and the other with zero. In summary: *each state of the bank is characterized by the number of savings accounts, the identity of each owner and the balance of each account*.

The requirements say that each account will be identified with the NIC of its owner and that we should store the balance of the account. Hence, we can think of a savings account as an ordered pair $(nic, balance)$ where nic is the NIC of the owner of the account and $balance$ is the current balance of the account. Therefore, we can put all these ordered pairs in a set to get a state of the bank:

$$\{(nic_1, balance_1), \ldots, (nic_n, balance_n)\}$$

That is, we have *the number of savings accounts, the identity of each owner and the balance of each account*. So if 120 euro are deposited in the account identified with nic_1 we have a new state:

$$\{(nic_1, balance_1 + 120), \ldots, (nic_n, balance_n)\}$$

In consequence, we can specify the state of the bank with a single state variable whose values are sets of ordered pairs. We will call this variable *sa* for *s*avings *a*ccounts. Now we state that *sa* is the state variable of the *Bank* machine as follows:

MACHINE *Bank*
SETS *NIC*
VARIABLES *sa*
    . . . the machine description continues here. . .
END

Any set of ordered pairs is a binary relation. In our case the domain of the binary relation is *NIC* and the range is $\mathbb{N}$, because balances cannot be negative. So, to begin with, we have

$sa \in NIC \leftrightarrow \mathbb{N}$. Furthermore, note that all the nic_i are different since each person has a different NIC and requirements say that each person can have at most one savings account. Then, *for each nic_i we have an unique balance*. Hence, we have a *function between NIC and* $\mathbb{N}$. So, in a second approximation, we have $sa \in NIC \rightarrow \mathbb{N}$. That is, *sa* is a total function. However, not every person will be a bank customer. This means that there may exist $n \in NIC$ for which $sa(n)$ is undefined because the person whose NIC is n isn't a bank customer. So, finally, we arrive at the right type for the state variable, $sa \in NIC \nrightarrow \mathbb{N}$. That is, *sa* is a partial function because not every person is a bank customer at all times. The type of *sa* is encoded as follows in the *Bank* machine:

MACHINE *Bank*
SETS *NIC*
VARIABLES *sa*
INVARIANT $sa \in NIC \nrightarrow \mathbb{N}$
    . . . the machine description continues here. . .
END

The word INVARIANT indicates that $sa \in NIC \nrightarrow \mathbb{N}$ is or should be a *state invariant*. A state invariant is a predicate that is true of every state of the machine. Then, we're saying that we want *sa* to be a partial function from *NIC* onto $\mathbb{N}$ in every state of the bank. Clearly, if we arrive to a state of the bank where there is an ordered pair $(n, b) \in sa$ such that $b < 0$ we're in trouble; likewise, if the bank ever gets to a state such that $(n, b_1), (n, b_2) \in sa \wedge b_1 \neq b_2$, we're in trouble. How can we be sure the bank will never get to those states? We'll see that in Section 5. For now, we'll use INVARIANT just as the place where we give the type of our state variables.

As any state machine *Bank* starts from some initial state. In this case the bank has no savings accounts. We encode this as follows:

MACHINE *Bank*
SETS *NIC*
VARIABLES *sa*
INVARIANT $sa \in NIC \nrightarrow \mathbb{N}$
INITIALIZATION $sa := \{\}$
    . . . the machine description continues here. . .
END

Recall that in B partial functions are sets of ordered pairs. So it is legal to state that a partial function is equal to the empty set.

Also observe that we use :=, called *abstract assignment*, to set the initial state. Abstract assignments are used to set the values of state variables and output parameters. $x := expr$ can be read as "x becomes $expr$". In $x := expr$, x must be a state variable or an output parameter and $expr$ can be an expression.

3.1.1 Why not a list instead of a function for *sa*?

What if we specify that *sa* is a list of ordered pairs instead of a partial function? For example, in Java we can define the class `SavingsAccount` with two member variables, `nic` and `bal`, to store the NIC and balance of a savings account. Then we can define a list of `SavingsAccount`:

```
List<SavingsAccount> sa;
```

where we can store the complete state of the bank.

So, why not doing the same in B? Actually, lists are available in B as sets of ordered pairs, called sequences. Then, we could have defined *sa* as $sa : \text{seq}(NIC \times \mathbb{N})$ instead as a partial function. So, why not doing like this which, after all, is closer to a possible Java implementation? The answer is because defining *sa* as a sequence (or a list) is not exactly what *sa* is. For example, the following is a possible value for *sa* if it's declared as a sequence:

$$[(nic_1, 100), (nic_2, 230), (nic_1, 529)]$$

That is, apparently, the bank would have three accounts. However, note that the first and last pairs have the same account identifier, nic_1, but different balances, 100 and 529. Is this right? What is the correct balance of the nic_1 savings account? The answer is that this isn't right because an account should have exactly one balance. The problem is that a sequence (or a list) allows for repetitions making it possible to have the same account identifier with different balances. But, according to the requirements, this cannot happen.

On the other hand, if *sa* is defined as a partial function, the following set:

$$\{(nic_1, 100), (nic_2, 230), (nic_1, 529)\}$$

can't be a value for *sa* because this set isn't a partial function.

The point is that a partial function correctly models what *sa* should be. It doesn't matter if partial functions aren't available in programming languages or if they aren't efficient enough because we are writing a formal *specification*, we aren't writing a program[1]. We write a specification because we want to clearly understand what the program should do. For now we aren't concerned with performance issues; for now, we don't care how to provide an efficient program, we just care about providing a *correct* program. Later on, we can (and we will) care about providing a *correct and efficient* program.

In other words:

1. When we write a specification we are defining the *problem*
2. When we write a program we are defining the *solution* to that problem

3.2 Specifying the operations for the savings accounts

So far the specification is the following:

MACHINE *Bank*
SETS *NIC*
VARIABLES *sa*
INVARIANT $sa \in NIC \nrightarrow \mathbb{N}$
INITIALIZATION $sa := \{\}$
. . . the machine description continues here. . .
END

[1]Actually, partial functions are available in, for instance, Java as hash tables.

That is, we have the types we are using in the specification, the definition of the set of states of the bank and the initial state.

The next step is to specify the operations that can be performed on the bank. According to the requirements we must specify the following five operations (which are classified as either state transitions or state queries):

1. Open an account (state transition)
2. Deposit money in an account (state transition)
3. Withdraw money from an account (state transition)
4. Query the current balance of an account (state query)
5. Close an account (state transition)

3.2.1 Open an account (state transition)

The operation specifying how an account is open is called **open**. In order to open an account we need the account number of the new account. This value is an *input parameter* to **open**; we call this parameter *an*. Operations are specified in a section named Operations, as follows.

Machine *Bank*
Sets *NIC*
Variables *sa*
Invariant $sa \in NIC \pfun \mathbb{N}$
Initialization $sa := \{\}$
Operations
  open(*an*) $\widehat{=}$ *. . . specify the operation here . . .*
    . . . the machine description continues here. . .
End

As we have said, 'open an account' is a state transition that can be described as follows:

- Take *an*
- Check if $an \in NIC$. Otherwise **open** could be called with a value that is not an account identifier.
- Check if there is an account whose identifier is *n* (because we can't open the same account twice)

  All the account identifiers used in the bank are the first components of the ordered pairs stored in *sa*. This set can be easily obtained by taking the domain of *sa*, which is exactly that: the set of the first components of the ordered pairs of *sa*.

  The domain of *sa* is written dom(*sa*). Then, checking if there is an account whose identifier is *n* can be written as $an \in \mathrm{dom}(sa)$.
- If *an* identifies a new account, then change the state of the bank by adding an ordered pair of the form $(an, 0)$ to *sa*.

  We add $(an, 0)$ because, according to the requirements, the new account's balance is zero.

Adding an ordered pair of the form $(an, 0)$ to sa can be done with set union: $sa \cup \{(an, 0)\}$. That is, $sa \cup \{(an, 0)\}$ is a partial function equal to sa except that it has one more ordered pair, $(an, 0)$.

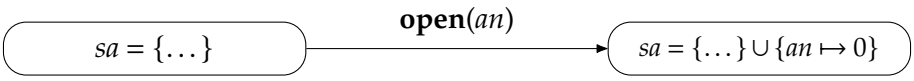
Recall that we write $sa \cup \{(an, 0)\}$ and not $sa \cup (an, 0)$ because $\cup$ expects two sets (and $(an, 0)$ is an ordered pair, not a set; $\{(an, 0)\}$ is a singleton set whose element is an ordered pair).

In B the ordered pair (x, y) can also be written as $x \mapsto y$. For instance, $sa \cup \{an \mapsto 0\}$. $x \mapsto y$ can be read as "x maps to y". We will use the $\mapsto$ notation from now on.

Now, we will write all that in the machine, as follows:

MACHINE *Bank*
SETS *NIC*
VARIABLES *sa*
INVARIANT $sa \in NIC \nrightarrow \mathbb{N}$
INITIALIZATION $sa := \{\}$
OPERATIONS
  open(an) $\widehat{=}$
    PRE $an \in NIC \wedge an \notin \mathrm{dom}(sa)$
    THEN $sa := sa \cup \{an \mapsto 0\}$
    END;
    . . . the machine description continues here. . .
END

The specification of **open** states that if the predicate written in the **PRE** section holds, then the predicate written in the **THEN** section must hold. As in the INITIALIZATION section, abstract assignments are used to set the values of the state variables in the new state. An abstract assignment is compact form of representing a predicate. We will see that in Section 5. Then, the specification of **open** can be depicted graphically as follows:



That is, if **open**(an) is executed from a state where sa is equal to some set, symbolized with $\{\ldots\}$, then the value of sa in the next state is $\{\ldots\} \cup \{(an, 0)\}$.

In general, an operation is specified by giving its *preconditions* and its *postconditions*. The **PRE** section can only hold preconditions; the **THEN** section, called *body*, can hold both, preconditions and postconditions. Then, in **open** the precondition is $an \in NIC \wedge an \notin \mathrm{dom}(sa)$, and the postcondition is $sa := sa \cup \{an \mapsto 0\}$. A precondition is a predicate that its *expected* to be true *before* the operation is executed and a postcondition is a predicate that is *guaranteed* to be true *after* the operation is executed. *If the operation is called when the precondition is not met, the operation can do anything*. In other words, the specification says that if the precondition is true before the operation is called, then the postcondition will be true after the operation is executed. It says nothing about the behavior when the precondition doesn't hold. In general, a precondition is a predicate that depends only on state variables and input parameters; and a postcondition is a predicate that can depend on any variable (state, input and output). The abstract assignment (:=) can be used only in postconditions. For example, $an \in NIC \wedge an \notin \mathrm{dom}(sa)$ is a precondition because it only depends on an (which is an input parameter) and sa (which is a state variable);

and $sa := sa \cup \{an \mapsto 0\}$ isn't a precondition, and so it's a postcondition, because it uses the abstract assignment (:=).

Therefore, when you have to specify an operation try to think first in terms of preconditions and postconditions. Then, write them down using the B language.

In the precondition we must give a type for each input parameter of the operation. We did so by stating $an \in NIC$.

> If the precondition is true before the operation is called, then the postcondition will be true after the operation is executed.

> If the operation is called when the precondition is not met, the operation can do anything.

> The caller is responsible of calling the operation when its precondition is met.

Improving the specification of open with error messages. The specification of the operation that opens an account given in **open** is fine but it doesn't say what the system should do if **open** is called when the precondition isn't true. That is, what should happen if we attempt to open an account whose identifier is already in use? So far, we have specified only called *normal behavior* or *happy path*. That is, we have specified the operation when the caller makes no mistakes. If the operation is called when the precondition does not hold, the specification says that **open** can't be executed. However, this is not very informative for the user. Imagine a user calling **open** with an account number already in use. The system will do nothing. Then, the user may think that the account has been opened because the system didn't show an error. Hence, we are going to improve **open** by adding to it *error messages* or *error codes*. More generally, we're going to specify the *abnormal behavior* or *erroneous behavior*.

Then, we need to say what are the messages the system will output when operations are executed. Further, we need to define a type for the messages. Are error messages character strings? Are they more like error codes, for instance 1 representing a successful operation and 0 an unsuccessful one? As with other types of the specification, we try to abstract these representations to capture the essence of what we want to say. Then, we will think of error messages as constants of an enumerated type. For now, we will think in just two messages:

1. *ok*, meaning that the operation has been successfully executed

2. *nicExists*, meaning that a given account identifier is already in use in the bank

The fact that these messages look like character strings doesn't mean they actually are so. *ok* and *nicExists* are just values of some type we still need to define. Given that this type will have a finite number of elements we can use an enumerated type. Then, we define the following:

MACHINE *Bank*
SETS *NIC*; *MSG* = {*ok*, *nicExists*} [new set]
VARIABLES *sa*
INVARIANT $sa \in NIC \nrightarrow \mathbb{N}$
INITIALIZATION $sa := \{\}$
OPERATIONS
open(*an*) $\widehat{=}$
PRE $an \in NIC \wedge an \notin \mathrm{dom}(sa)$
THEN $sa := sa \cup \{an \mapsto 0\}$
END;
. . . the machine description continues here. . .
END

We will add more elements to *MSG* as needed.

Showing an error message means that the system outputs the message to the environment. B provides a general mechanism for operations to produce outputs (they can be error messages or whatever other information should be sent to the environment). This mechanism is based on *output parameters*. An output parameter is declared as follows:

$msg \leftarrow$ **open**(*an*) $\widehat{=} \ldots$

In this way *msg* is an output parameter of **open**.

Now, we will redefine **open** as to output *ok* when everything goes right, and *nicExists* when an already used account number is passed in as input parameter. If $an \notin NIC$ the operation will not be executed. Since outputs are something the operation produces once it's executed we need to modify the **THEN** section as well as moving some preconditions .

MACHINE *Bank*
SETS *NIC*; *MSG* = {*ok*, *nicExists*}
VARIABLES *sa*
INVARIANT $sa \in NIC \nrightarrow \mathbb{N}$
INITIALIZATION $sa := \{\}$
OPERATIONS
$msg \leftarrow$ **open**(*an*) $\widehat{=}$
PRE $an \in NIC$
THEN **IF** $an \notin \mathrm{dom}(sa)$ **THEN** $sa, msg := sa \cup \{an \mapsto 0\}, ok$ **ELSE** $msg := nicExists$ **END**
END;
. . . the machine description continues here. . .
END

Observe the statement:

$sa, msg := sa \cup \{an \mapsto 0\}, ok$

This is called *multiple assignment*. The meaning of the multiple assignment is that several variables are simultaneously assigned each one with a value. This kind of assignment can also

be written as a *parallel assignment*:

$$sa := sa \cup \{an \mapsto 0\} \parallel msg := ok$$

Parallel assignment is commutative:

$$x_1 := expr_1 \parallel x_2 := expr_2 \equiv x_2 := expr_2 \parallel x_1 := expr_1$$

This makes it clear that the assignments aren't executed in any particular order.

Also observe that in the **ELSE** branch we don't change the state of the machine. Then, only an output is produced when $an \in \text{dom}(sa)$.

Finally, consider that if $an \in NIC$ doesn't hold the operation is not executed at all; nothing happens in that case, then.

Why not considering the possibility of calling **open**(an) with $an \notin NIC$? In other words, why we don't issue an error message when $an \notin NIC$? The idea is that the programmer will implement NIC as some type in the programming language. Then, the compiler of that programming language won't compile a program where there is a chance of an not being of the type chosen by the programmer. Hence, we don't need to consider the case where $an \notin NIC$. In this way the operation can't be called in a state where its precondition doesn't hold.

Specifications and implementations. Had this been the full specification of the system we should handle it to a programmer so (s)he would implement it in some programming language. It wouldn't be necessary to handle him or her the user requirements (Section 2). The specification contains enough information for the programmer to implement a correct program. More importantly, the specification is *precise*, *unambiguous* and *abstract*. Then, the specification doesn't forbid any particular implementation as long as it's correctly implemented.

However, before handling the specification to the programmer we can *mathematically* check that the specification is more or less correct. We'll see more on this in Section 5. Furthermore, specifications can be *mathematically* transformed into programs. In this way, the implementation is said to be *correct by construction*. The B-Method is particularly well-suited for that, although we won't get that far in this course.

3.2.2 Deposit money in an account (state transition)

Once we have an account we can deposit money in it. The operation, called **deposit**, receives two inputs: the account identifier, an, and the amount to be deposited, amt. The type of an is NIC and the type of amt is $\mathbb{N}$. The preconditions for successfully depositing an amount of money amt in the account an are the following:

1. an must be a NIC. Formally:

   $$an \in NIC$$

   This is the type of an. It's mandatory to include the type of each input parameter.

2. an must be a valid account identifier. Formally:

   $$an \in \text{dom}(sa)$$

3. *amt* must be a natural number. Formally:

   $amt \in \mathbb{N}$

   This is the type of *amt*. It's mandatory to include the type of each input parameter.
4. *amt* must be a positive number (it makes no sense to deposit zero euro). Formally:

   $amt > 0$

The postconditions for the *normal behavior* are the following:

1. The message *ok* must be shown.
2. The balance of *an* must be equal to its balance in the start state plus *amt*; all other accounts remain the same; no accounts are added nor deleted from the bank.

   The balance of *an* in the start state is $sa(an)$. Indeed, since *sa* is a function from *NIC* onto $\mathbb{N}$, then $sa(an)$ is the balance of *an*. It's like, what is the cosine of π? The answer is $\cos(\pi)$. That is, we just apply the function to the argument. Besides, *sa* denotes the set of accounts of the bank in the start state, so $sa(an)$ is the balance of *an* in the start state. Now, $sa(an) + amt$ is equal to the balance of *an* in the start state plus *amt*. Then, $sa(an) + amt$ must be the balance of *an* in the next state. We would like to write something like: $sa(an) := sa(an) + amt$. However, this is illegal in B because the left hand side of an abstract assignment must always be a variable. Anyway, let's keep an eye on this.

   Now we look at the other conditions: all other accounts remain the same; no accounts are added nor deleted from the bank. Let's abuse the abstract assignment again:

   $$\begin{array}{l} \mathrm{dom}(sa) := \mathrm{dom}(sa) \\ \forall i.(i \in \mathrm{dom}(sa) \Rightarrow (i \neq an \Rightarrow sa(i) := sa(i))) \end{array}$$

   The first formula says that no account is added nor deleted from *sa*; the second says that the balances of all the accounts different from *an* suffer no modifications. So we have the following three conditions that establish the value of *sa* in the next state:

   $$\begin{array}{l} sa(an) := sa(an) + amt \\ \mathrm{dom}(sa) := \mathrm{dom}(sa) \\ \forall i.(i \in \mathrm{dom}(sa) \Rightarrow (i \neq an \Rightarrow sa(i) := sa(i))) \end{array}$$

   These conditions appear very frequently in B so the language provides an operator, called *update* and noted $\lhd\!\!+$, that specifies those three conditions in a single formula. Then, the right B formula for the postcondition is the following:

   $$sa := sa \lhd\!\!+ \{an \mapsto sa(an) + amt\}$$

   where the left hand side is now a variable and the right hand side an expression. B provides $\lhd\!\!+$ because updating a function is a frequent operation in specifications. Updating a function means to change the second component of one or more of its ordered pairs. Hence, in general, we can use $\lhd\!\!+$ as follows:

   $$f \lhd\!\!+ \{(x_1, y_1), \ldots, (x_n, y_n)\}$$

   Furthermore, if an x_i doesn't belong to the domain of f then $\lhd\!\!+$ adds the ordered pair (x_i, y_i) to f. Then, $f \lhd\!\!+ \{(x_1, y_1), \ldots, (x_n, y_n)\}$ will always be defined for all x_i.

   We will present the mathematical details of $\lhd\!\!+$ below.

Now we can specify the normal behavior when a deposit is made.

MACHINE *Bank*
SETS NIC; $MSG = \{ok, nicExists\}$
VARIABLES sa
INVARIANT $sa \in NIC \pfun \mathbb{N}$
INITIALIZATION $sa := \{\}$
OPERATIONS
$msg \leftarrow$ **open**$(an) \mathrel{\widehat{=}}$
PRE $an \in NIC$
THEN **IF** $an \notin \mathrm{dom}(sa)$ **THEN** $sa, msg := sa \cup \{an \mapsto 0\}, ok$ **ELSE** $msg := nicExists$ **END**
END;
$msg \leftarrow$ **deposit**$(an, amt) \mathrel{\widehat{=}}$
PRE $an \in NIC \land amt \in \mathbb{N} \land an \in \mathrm{dom}(sa) \land 0 < amt$
THEN $sa, msg := sa \ovl{\cup} \{an \mapsto sa(an) + amt\}, ok$
END;
. . . the machine description continues here. . .
END

Even though $0 < amt$ implies $amt \in \mathbb{N}$, B requires $amt \in \mathbb{N}$ to be included in the precondition anyway. Same happens with $an \in \mathrm{dom}(sa)$ and $an \in NIC$. We can say that $amt \in \mathbb{N}$ and $an \in NIC$ are type constraints, whereas $0 < amt$ and $an \in \mathrm{dom}(sa)$ are functional preconditions. In a technical sense both are preconditions but of a different nature.

Next, we need to specify the abnormal behaviors. In order to do that we need to negate each *functional* precondition and specify an error message for each negation[2]. In this case we have two abnormal behaviors given by the negation of each functional precondition:

- $an \notin \mathrm{dom}(sa)$ then output *nicNotExists*
- $amt = 0$ then output *amountError*

This makes three behaviors for **deposit**: one normal plus two abnormal. We can specify this with a nested **IF-THEN-ELSE**, as follows:

IF $an \in \mathrm{dom}(sa) \land 0 < amt$
THEN $sa, msg := sa \ovl{\cup} \{an \mapsto sa(an) + amt\}, ok$
ELSE IF $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$ **ELSE** $msg := amountError$ **END**
END

However, B provides a more convenient statement for these cases that avoids nesting:

SELECT $an \in \mathrm{dom}(sa) \land 0 < amt$ **THEN** $sa, msg := sa \ovl{\cup} \{an \mapsto sa(an) + amt\}, ok$
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $amt = 0$ **THEN** $msg := amountError$
END

[2]Another option is to define a single behavior comprising all the abnormal behaviors. In that case we can specify that a generic *error* message is shown. However, in this course we will always write one case for each abnormal behavior.

MACHINE *Bank*
SETS *NIC*; *MSG* = {*ok*, *nicExists*, *nicNotExists*, *amountError*} [more elements in *MSG*]
VARIABLES *sa*
INVARIANT $sa \in NIC \pfun \mathbb{N}$
INITIALIZATION $sa := \{\}$
OPERATIONS
$msg \leftarrow$ **open**$(an) \mathrel{\widehat{=}}$
PRE $an \in NIC$
THEN **IF** $an \notin \mathrm{dom}(sa)$ **THEN** $sa, msg := sa \cup \{an \mapsto 0\}, ok$ **ELSE** $msg := nicExists$ **END**
END;
$msg \leftarrow$ **deposit**$(an, amt) \mathrel{\widehat{=}}$
PRE $an \in NIC \wedge amt \in \mathbb{N}$
THEN
SELECT $an \in \mathrm{dom}(sa) \wedge 0 < amt$ **THEN** $sa, msg := sa \ovl{\leftarrow} \{an \mapsto sa(an) + amt\}, ok$
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $amt = 0$ **THEN** $msg := amountError$
END
END
END

Figure 1: Specification of the **deposit** operation

If several conditions on a **SELECT** statement are true at the same time then the choice of the branch that is executed is made nondeterministically. If none of the conditions is true, then the entire statement won't be executed. In our case the last two branches can be true at the same time. This means that in that case **deposit** will show only one error message, but we don't know which one.

Finally, we can define the complete operation as shown in Figure 1.

Note that we move the functional preconditions to the **THEN** section in order to be able to select what postcondition must hold in each case. Actually, the specification of **deposit** is equivalent to the following formula:

$$\begin{aligned} & an \in NIC \wedge amt \in \mathbb{N} \\ & \wedge (an \in \mathrm{dom}(sa) \wedge 0 < amt \Rightarrow sa, msg := sa \ovl{\leftarrow} \{an \mapsto sa(an) + amt\}, ok) \\ & \wedge (an \notin \mathrm{dom}(sa) \Rightarrow msg := nicNotExists) \\ & \wedge (amt = 0 \Rightarrow msg := amountError) \end{aligned} \tag{1}$$

Keep in mind that $x := expr$ isn't a predicate but represents a predicate. So the above formula is not exactly right because we're mixing predicates with non-predicates. Nevertheless, we think that the formula helps to understand what's going on.

Mathematical details of the update operator ($\ovl{\leftarrow}$) and other important B operators. If X and Y are two B types and $R, S \in X \leftrightarrow Y$, then the formal definition of the update operator is the

following:

$$R \lhd\!\!+ S = ((\mathrm{dom}(S)) \lhd\!\!| R) \cup S$$

where, if $A \in \mathbb{P}\, X$ then:

$$A \lhd\!\!| R = \{x \mapsto y \mid x \mapsto y \in R \wedge x \notin A\}$$

Therefore, $\lhd\!\!+$ is actually defined for binary relations, and not only for partial functions. As any partial function is a binary relation then we can apply $\lhd\!\!+$ to partial functions as well. Clearly, if $R \in X \leftrightarrow Y$ and $f \in X \nrightarrow Y$, then $R \lhd\!\!+ f$ and $f \lhd\!\!+ R$ are both type-correct. Besides, as any set of the form $\{x \mapsto y\}$ is a binary relation then we can write $sa \lhd\!\!+ \{an \mapsto sa(an) + amt)\}$ as we did in **deposit**. Consider, though, that $\lhd\!\!+$ is defined for two binary relations of the *same type*.

As can be seen, $\lhd\!\!+$ is defined in terms of other B operator, $\lhd\!\!|$, called *domain anti-restriction*. This operator takes a set and a binary relation, and returns another binary relation. In effect, the returned relation is a *restriction* of the initial relation; that is $A \lhd\!\!| R \subseteq R$. $A \lhd\!\!| R$ removes from R all the ordered pairs whose first component doesn't belong to A. For example:

$$\{2,4\} \lhd\!\!| \{(5,a),(4,w),(4,q),(7,e),(2,q)\} = \{(5,a),(7,e)\}$$

Therefore, if $x \in \mathrm{dom}(R)$ then $R \lhd\!\!+ \{x \mapsto y\}$ first removes from R all the ordered pairs of the form $x \mapsto \cdot$ and then adds $x \mapsto y$ to the resulting relation. In other words:

$$\mathrm{dom}(\{x \mapsto y\}) \lhd\!\!| R = \{x\} \lhd\!\!| R \qquad \text{removes pairs of the form } x \mapsto \cdot$$
$$(\{x\} \lhd\!\!| R) \cup \{x \mapsto y\} = R \lhd\!\!+ \{x \mapsto y\} \qquad \text{adds } x \mapsto y$$

Clearly, if $x \notin \mathrm{dom}(R)$ then $R \lhd\!\!+ \{(x,y)\}$ is equal to $R \cup \{(x,y)\}$.

As an example consider the following:

$$\begin{aligned}
&\{(5,a),(4,w),(4,q),(7,e),(2,q)\} \lhd\!\!+ \{(7,x)\} \\
&= (\mathrm{dom}(\{(7,x)\}) \lhd\!\!| \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(7,x)\} \\
&= (\{7\} \lhd\!\!| \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(7,x)\} \\
&= \{(5,a),(4,w),(4,q),(2,q)\} \cup \{(7,x)\} \\
&= \{(5,a),(4,w),(4,q),(2,q),(7,x)\}
\end{aligned}$$

But also check out this example:

$$\begin{aligned}
&\{(5,a),(4,w),(4,q),(7,e),(2,q)\} \lhd\!\!+ \{(4,y)\} \\
&= (\mathrm{dom}(\{(4,y)\}) \lhd\!\!| \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(4,y)\} \\
&= (\{4\} \lhd\!\!| \{(5,a),(4,w),(4,q),(7,e),(2,q)\}) \cup \{(7,x)\} \\
&= \{(5,a),(7,e),(2,q)\} \cup \{(4,y)\} \\
&= \{(5,a),(7,e),(2,q),(4,y)\}
\end{aligned}$$

where the final relation has fewer elements than the initial relation.

B defines many other powerful set and relational operators such as $\lhd\!\!+$ and $\lhd\!\!|$. The following table lists some of them; B textbooks provide complete lists of them.

Name	Parameters	Definition
Range of a relation	$R \in X \leftrightarrow Y$	$\text{ran}\, R = \{y \mid y \in Y \land \exists x.(x \in X \land (x,y) \in R)\}$
Domain restriction	$R \in X \leftrightarrow Y;\ A \in \mathbb{P}\, X$	$A \lhd R = \{(x,y) \mid (x,y) \in R \land x \in A\}$
Range restriction	$R \in X \leftrightarrow Y;\ B \in \mathbb{P}\, Y$	$R \rhd B = \{(x,y) \mid (x,y) \in R \land y \in B\}$
Range anti-restriction	$R \in X \leftrightarrow Y;\ B \in \mathbb{P}\, Y$	$R \rhd\!\!> B = \{(x,y) \mid (x,y) \in R \land y \notin B\}$
Relational image	$R \in X \leftrightarrow Y;\ A \in \mathbb{P}\, X$	$R[A] = \text{ran}(A \lhd R)$

Set and relational operators must be preferred over complex logical predicates including universal and existential quantifiers. The collection of operators provided by B can express many complex specifications. Logical formulas can be avoided most of the times. Whenever you're about to write a specification including an universal quantifier, think twice: most of the times there's a B operator that can do the job.

We now come back to the specification of the other operations on savings accounts.

3.2.3 Withdraw money from an account (state transition)

Withdrawing money from an account is almost the same than depositing it. The operation, called **withdraw**, receives the same two input parameters, *an* and *amt*. The functional preconditions for the normal behavior are those of **deposit** plus the following one:

- *amt* must be less than or equal to the current balance of *an*. Formally:

$$amy \leq sa(an)$$

The postconditions for the normal behavior are similar to those of **deposit** except that *amt* must be subtracted from the current balance. Formally:

$$sa := sa \Leftarrow \{an \mapsto sa(an) - amt)\}$$

Concerning the abnormal behaviors we have to add one more due to the new precondition. The complete specification of **withdraw** is shown in Figure 2.

Evaluating partial a function outside its domain. Note that in the last branch of the SELECT statement of the **withdraw** operation we added $an \in \text{dom}(sa)$ as a precondition to the assignment $msg := insufficientFunds$. In general, the precondition of an abnormal behavior is usually only the negation of the precondition we are considering. However, in that branch we have the negation of $an \leq sa(an)$ and $an \in \text{dom}(sa)$. The reason for adding $an \in \text{dom}(sa)$ as a precondition is that we need to be sure that $sa(an)$ is defined because otherwise the predicate $sa(an) < amt$ will have an undefined value. In other words, $sa(an) < amt$ is true or false if $sa(an)$ is a number and $sa(an)$ is a number if *an* belongs to the domain of *sa*. If $an \notin \text{dom}(sa)$, what is the value of $sa(an)$? Put it in another way, if $an \notin \text{dom}(sa)$, what is the balance of *an* according to *sa*? If $an \notin \text{dom}(sa)$, it

MACHINE *Bank*
SETS *NIC*; $MSG = \{ok, nicExists, nicNotExists, amountError, insufficientFunds\}$
VARIABLES *sa*
INVARIANT $sa \in NIC \pfun \mathbb{N}$
INITIALIZATION $sa := \{\}$
OPERATIONS

$msg \leftarrow$ **open**$(an) \mathrel{\widehat{=}}$
PRE $an \in NIC$
THEN **IF** $an \notin \mathrm{dom}(sa)$ **THEN** $sa, msg := sa \cup \{an \mapsto 0\}, ok$ **ELSE** $msg := nicExists$ **END**
END;

$msg \leftarrow$ **deposit**$(an, amt) \mathrel{\widehat{=}}$
PRE $an \in NIC \land amt \in \mathbb{N}$
THEN
SELECT $an \in \mathrm{dom}(sa) \land 0 < amt$ **THEN** $sa, msg := sa \ovl{\cup} \{an \mapsto sa(an) + amt\}, ok$
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $amt = 0$ **THEN** $msg := amountError$
END
END;

$msg \leftarrow$ **withdraw**$(an, amt) \mathrel{\widehat{=}}$
PRE $an \in NIC \land amt \in \mathbb{N}$
THEN
SELECT $an \in \mathrm{dom}(sa) \land 0 < amt \land amt \leq sa(an)$
THEN $sa, msg := sa \ovl{\cup} \{an \mapsto sa(an) - amt\}, ok$
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $amt = 0$ **THEN** $msg := amountError$
WHEN $an \in \mathrm{dom}(sa) \land sa(an) < amt$ **THEN** $msg := insufficientFunds$
END
END

END

Figure 2: Specification of the **withdraw** operation

means that *an* isn't an account in *sa* so, can we say what its balance is? Therefore, to be sure that $sa(an) < amt$ is a well-defined predicate we must ensure that $sa(an)$ is defined and so we must require $an \in \mathrm{dom}(sa)$.

The same situation arises in mathematics. For example, the division function over the real numbers, that is x/y with $x, y \in \mathbb{R}$, is defined only if $y \neq 0$. Then / can be defined as a (total) function:

$$\_/\_ : \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \to \mathbb{R}$$

or as a partial function:

$$\_/\_ : \mathbb{R} \times \mathbb{R} \pfun \mathbb{R}$$

Hence, for example, the expression:

$$\frac{1}{x^2-4}$$

is a real number if $x^2 - 4 \neq 0$. If $x = 2$ the expression reduces to $1/0$ which is undefined because $(1,0) \notin \text{dom}(/)$ (note that the domain of / are ordered pairs). In mathematics this problem is solved by forcing the definition of $x/0$ to be ∞. However $\infty \notin \mathbb{R}$, that is ∞ isn't a real number (that's why we write ∞ and not a number formed with digits). Therefore, in mathematics the function is 'totalized' by extending its range:

$$\_/\_ : \mathbb{R} \times \mathbb{R} \rightarrow (\mathbb{R} \cup \{\infty\})$$

where $\mathbb{R} \cup \{\infty\}$ is some times noted as $\mathbb{R}^{\infty}$.

Actually, the same situation arises in programming, too. Every time we forget to check if the denominator of a division isn't zero our program crashes. In programming, ∞ corresponds to an abnormal termination (crashing) of the program (for example, with a message such as `division by zero` or `null pointer exception` or `segmentation fault`). Programming is plenty of partial functions. For example, arrays can be seen as partial functions. Indeed, if `a` is an array of 20 components, then trying to access `a[35]` is undefined and could make the program crash. That is, an expression such as `a[_]` waits for an integer expression but it's defined only within the bounds of the array (from 1 to 20 in our example).

B is plenty of partial functions because programming is plenty of partial functions, too. In B we solve the problem of evaluating a partial function outside its domain by defining two behaviors:

1. We conjoin $x \in \text{dom}(f)$ to any predicate where $f(x)$ is included, for every partial function f; and then
2. We specify another behavior for the case $x \notin \text{dom}(f)$. For example, we state that the system doesn't transition to another state and shows an error message.

3.2.4 Query the current balance of an account (state query)

This is the first operation of our system that is not a state transition. Nevertheless, in B state queries and state transition are almost the same. The only significant difference is that state queries don't use assignment to state variables.

The operation to check the balance of an account, called **checkBalance**, receives an account identifier, *an*, and outputs the balance in *bal*. The functional precondition for the normal behavior is that *an* must be a valid account, i.e. $an \in \text{dom}(sa)$. There are two postconditions for the normal behavior:

1. $bal := sa(an)$, that is the output is exactly the balance of *an* in the current state.
2. $msg := ok$, that is the operation executed successfully.

There's only one abnormal behavior given by the negation of the functional precondition $an \in \text{dom}(sa)$. The operation is specified in Figure 3

MACHINE *Bank*
SETS *NIC*;
MSG = {*ok*, *nicExists*, *nicNotExists*, *amountError*, *insufficientFunds*, *balanceNotZero*}
VARIABLES *sa*
INVARIANT $sa \in NIC \pfun \mathbb{N}$
INITIALIZATION $sa := \{\}$
OPERATIONS

$msg \leftarrow$ **open**$(an) \mathrel{\widehat{=}}$
PRE $an \in NIC$
THEN **IF** $an \notin \mathrm{dom}(sa)$ **THEN** $sa, msg := sa \cup \{an \mapsto 0\}, ok$ **ELSE** $msg := nicExists$ **END**
END;

$msg \leftarrow$ **deposit**$(an, amt) \mathrel{\widehat{=}}$
PRE $an \in NIC \land amt \in \mathbb{N}$
THEN
SELECT $an \in \mathrm{dom}(sa) \land 0 < amt$ **THEN** $sa, msg := sa \lhd\!\!+ \{an \mapsto sa(an) + amt\}, ok$
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $amt = 0$ **THEN** $msg := amountError$
END
END;

$msg \leftarrow$ **withdraw**$(an, amt) \mathrel{\widehat{=}}$
PRE $an \in NIC \land amt \in \mathbb{N}$
THEN
SELECT $an \in \mathrm{dom}(sa) \land 0 < amt \land amt \leq sa(an)$
THEN $sa, msg := sa \lhd\!\!+ \{an \mapsto sa(an) - amt\}, ok$
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $amt = 0$ **THEN** $msg := amountError$
WHEN $an \in \mathrm{dom}(sa) \land sa(an) < amt$ **THEN** $msg := insufficientFunds$
END
END;

$bal, msg \leftarrow$ **checkBalance**$(an) \mathrel{\widehat{=}}$
PRE $an \in NIC$
THEN **IF** $an \in \mathrm{dom}(sa)$ **THEN** $bal, msg := sa(an), ok$ **ELSE** $msg := nicNotExists$ **END**
END;

$msg \leftarrow$ **close**$(an) \mathrel{\widehat{=}}$
PRE $an \in NIC \land amt \in \mathbb{N}$
THEN
SELECT $an \in \mathrm{dom}(sa) \land sa(an) = 0$ **THEN** $sa, msg := \{an\} \mathrel{⩤} sa, ok$
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $an \in \mathrm{dom}(sa) \land sa(an) \neq 0$ **THEN** $msg := balanceNotZero$
END
END
END

Figure 3: Specification of the **checkBalance** and **close** operations

3.2.5 Close an account (state transition)

The last operation we need to specify is closing an account, called **close**. It waits for the account to be closed, *an*. The functional preconditions for the normal behavior are the following:

1. *an* must be a valid account, i.e. $an \in \text{dom}(sa)$.
2. The balance of *an* must be zero. Formally:

$$sa(an) = 0$$

In turn, the postcondition for the normal behavior is that the value of *sa* in the next-state must be equal to the value in the before-state minus the account identified with *an*. We can't do this:

$$sa := sa \setminus \{an\}$$

because it's not type-correct. Recall that *sa* is a set of ordered pairs but $\{an\}$ is not. We can do this instead:

$$sa := sa \setminus \{an \mapsto 0\}$$

which is now type-correct and we know for sure that the balance of *an* is zero because it's in our preconditions. However, B offers a more convenient operator to state what we want:

$$sa := \{an\} \lhd sa$$

Recall that $\lhd$ is called domain anti-restriction. The expression $\{an\} \lhd sa$ returns a partial function equal to *sa* except that the ordered pair whose first component is *an* has been removed. This is exactly what we wanted to say when we wrote $sa := sa \setminus \{an\}$, unless the formula based on $\lhd$ is type-correct. In other words, $\{an\} \lhd sa$ removes from *sa* the ordered pair whose first component is *an* regardless of which is its second component. Obviously, if $an \notin \text{dom}(sa)$, then $\{an\} \lhd sa = sa$.

In turn, we have two abnormal behaviors given by the negation of the two functional preconditions. Hence, we specify the as is shown in Figure 3.

4 Machine parameters and constraints

Let's assume we have a new requirement for our savings account system:

- Every time a deposit of 10 000 euros or more is made in an account, the bank must report the amount and the account identifier to the central bank.

As we are writing a specification, we will abstract away the mechanism used to report such an event to the central bank. In order to do that, we will simply specify an operation that outputs the amount and account ID to the environment when the event occurs.

Secondly, we need to think how the number 10 000 will be included in the specification. We can think this number is part of the state of the bank and thus we need to define a new state variable. However, this number isn't set by the bank; it's set by an external system. Then, it makes no sense to include it as part of the state of the bank. Actually, at the implementation level, we can think this number is stored in a configuration file that's read by the system every

time it's executed; or else, the central bank may offer a public interface where banks can query what the current limit to report deposits is.

For things that we think are external to our system but which are nonetheless needed by it, B offers a construct called *machine parameters*. We use a machine parameter to configure the *Bank* machine as follows:

MACHINE *Bank(depRepLim)*
... the machine description goes here...
END

Then, *depRepLim* is a parameter for our *Bank* machine. In spite that the central bank stated that this limit is equal to 10 000 we know it can change over time but, at the same time, we know as that it's always going to be a positive number. We can specify this condition in the CONSTRAINTS section as follows:

MACHINE *Bank(depRepLim)*
CONSTRAINTS $depRepLim \in \mathbb{N} \land 0 < depRepLim$
... the machine description continues here...
END

Now we can use *depRepLim* to specify the **deposit** operation including the report to the central bank in case the deposit limit is exceeded.

$rn, ra, msg \leftarrow$ **deposit**$(an, amt) \mathrel{\widehat{=}}$
PRE $an \in NIC \land amt \in \mathbb{N}$
THEN
SELECT $an \in \mathrm{dom}(sa) \land 0 < amt$ **THEN**
IF $depRepLim \leq a$?
THEN $sa, rn, ra, msg := sa \lhd\!\!+ \{an \mapsto sa(an) + amt\}, an, amt, ok$
ELSE $sa, msg := sa \lhd\!\!+ \{an \mapsto sa(an) + amt\}, ok$
END
WHEN $an \notin \mathrm{dom}(sa)$ **THEN** $msg := nicNotExists$
WHEN $amt = 0$ **THEN** $msg := amountError$
END
END

As we have said, the report is abstracted by outputting the account number (*rn*) and the deposit amount (*ra*).

We can include all the parameters we need in the specification of a B machine.

MACHINE *MyMachine*$(P_1, \ldots, P_n, p_1, \ldots, p_m)$
... the machine description goes here...
END

Parameters that are intended to be types are written in uppercase (i.e. $P_1, \ldots, P_n$); parameters of other types are written in lowercase (i.e. $p_1, \ldots, p_m$). If present, the CONSTRAINTS section goes right after the MACHINE line. CONSTRAINTS is used to describe conditions or restrictions

on the machine parameters. Machine parameters can be used in the INVARIANT and OPERATIONS sections.

An important use for machine parameters is to leave some requirements *underspecified* or *subspecified*. In this context, underspecification means to write a specification more abstract than normally needed; to write a specification with even less details. Underspecification originates mainly for three reasons:

1. At present time you don't know exactly how the system should behave under some conditions.
2. Because you're specifying an aspect of the system that isn't interesting enough and so you don't want to spend time and effort on it.
3. You want to specify a system as general as possible.

By stating $0 < depRepLim$ we're underspecifying the system because we want to specify a system as general as possible. The system will behave the same regardless of the specific value of *depRepLim*; that is, it will report all deposits of *depRepLim* or more euros regardless of the value of *depRepLim*. In this way our specification is a good model for any value of *depRepLim*.

As an example of the second reason to leave something underspecified, consider a system where you need to compute the checksum of a list of bytes. For now, you don't want to get into the details of this algorithm, so you want to leave it underspecified. So you define a machine receiving two parameters: *BYTE*, which is the type of bytes (that's why we write it in uppercase); and *checksum*, which is intended to be the checksum function. Formally, we have the following:

MACHINE *CommSystem(BYTE,checksum)*
CONSTRAINTS $checksum \in \text{seq}(BYTE) \rightarrow BYTE$
. . . the machine description continues here. . .
END

That is, the *CommSystem* machine is going to be consistent only if *checksum* is a function taking a sequence of *BYTE* and returning a *BYTE*. And that's it. *checksum* somewhat magically computes the checksum of any list of bytes. We're not interested in the details of the checksum algorithm. We don't need to specify that algorithm. Nevertheless, we can use *checksum* in the definition of operations:

MACHINE *CommSystem(BYTE,checksum)*
CONSTRAINTS $checksum \in \text{seq}(BYTE) \rightarrow BYTE$
. . . . . .
OPERATIONS
. . . . . .
add$(msg) \mathrel{\widehat{=}}$
PRE $msg \in \text{seq}(BYTE)$
THEN $checks := checks \cup \{checksum(msg)\}$
END;
. . . . . .
END

5 Introduction to machine consistency

Is the specification correct? How can we know whether or not the specification is correct? We write a specification mainly for two reasons: to understand the problem before attempting an implementation, and to to be used by programmers as the recipe from where the implementation is produced. But, what if the specification is not correct? Is this possible? Yes, the specification can have errors. If we handle an erroneous specification to the programmers the resulting implementation will be erroneous as well. Even worse, those errors will be detected only by the client or the user during acceptation testing or when it is in production.

A specification can contain errors. Normally, the specification will contain much less errors than the implementation. At the same time, these errors are usually much easier to find than errors in the implementation. Besides, detecting errors in the specification is less costly than detecting the same errors during testing or when the system has already been delivered to the client.

There are two main ways of finding errors in the specification:

1. By running sort of tests on the specification.
2. By mathematically proving that the specification is free of certain classes of errors. Or, put it in another way, by mathematically proving that the specification enjoys some mathematical properties.

   As we have said, the specification is written from the requirements, which are an informal statement of the system's functionality. Hence, in the process of formalizing the requirements one may make mistakes or may misunderstand or misinterpret the requirements, thus obtaining a wrong specification. Actually, one of the reasons to write a specification is to understand the requirements. As a consequence, if we can prove that the specification enjoys some properties, that are obvious from the requirements, then we can increase our confidence in that the specification is a faithful formalization of the requirements. Many times these properties are easier to formalize than the operations of the state machine. So there are fewer chances to make a mistake during the formalization of the properties.

We will work on the first technique when {*log*} is introduced. Now, we are going to introduce the second technique. In general all formal specification languages are prepared to perform mathematical proofs on their specifications. The B-Method is no exception and proposes a technique called *machine consistency* to check that a specification enjoy some important mathematical properties. If the specification is machine consistent then some classes of errors will not be present in it. In this section we will provide an introduction to machine consistency—and we will see it in action when {*log*} is introduced.

The core of machine consistency is to mathematically prove that each operation of a machine preserves the *state invariant*. The state invariant is the predicate written in the INVARIANT section. Conceptually, a state invariant, or just invariant, is a property that is true of every state of the machine. In other words, a state invariant is a predicate, depending only on state variables, which is true of every state of the state machine. State invariants are important because they can be used to gain confidence on the correctness of the specification.

Recall the state invariant we have for the *Bank* machine (see Figure 3):

$$sa \in NIC \pfun \mathbb{N} \tag{2}$$

Clearly, this property must be verified by the system and is obvious from the requirements. If we manage to *prove* that this property is true in every state of the bank, we are sure the following always hold:

- Each savings account has only one balance
- The balance of any savings account is never below zero

Both properties are important for the client. The bank officials will be sure that no customer can own many to the bank and that customers won't complain about the balance of their accounts. That is, by writing a very short formula we capture important properties of a bank system.

We can make mistakes writing the invariant but it's easier to be formalized than the collection of operations of the system. At the same time, if we ask the programmer to implement a system verifying (2), (s)he doesn't have a clear description of what (s)he has to do because we aren't telling her or him what the operations of the system are. On the other hand, if we give her or him a specification of the operations which fails to verify that property, then (s)he will implement a system that will be incorrect from the user perspective. And we can't blame the programmer. This is not because the program doesn't verify the specification but because the specification isn't a faithful formalization of the requirements.

Therefore, before handling the specification to the programmer we should try to verify the specification by proving properties it should meet. The more properties we can prove, the better. Below we will show some of the properties that the B-Method requires to prove of every machine plus one more added by us. But first we recall the following concept of first-order logic.

Substitution

Let P be a predicate depending on variable v, written as $P(v)$. Let e be an expression of the same type of v. We write $P[v \mapsto e]$ to denote the substitution of v by e in P. That is, $P[v \mapsto e]$ is equal to P except that v has been substituted by e.

As an example of substitution consider:

- $sa \in NIC \nrightarrow \mathbb{N}[sa \mapsto \{\}]$ which reduces to $\{\} \in NIC \nrightarrow \mathbb{N}$
- $sa \in NIC \nrightarrow \mathbb{N}[sa \mapsto sa \lhd\!\!+ \{an \mapsto sa(an) + amt\}]$ which reduces to $sa \lhd\!\!+ \{an \mapsto sa(an) + amt\} \in NIC \nrightarrow \mathbb{N}$

Now we present the rules of machine consistency. Each rule is called *verification condition* or *proof obligation*. When we prove one of these proof obligations we say that we *discharge* the proof obligation. Assume we have the following machine.

MACHINE M
VARIABLES v
INVARIANT Inv
INITIALIZATION $v := Init$
OPERATIONS
$y \leftarrow \mathbf{op}(x) \mathrel{\widehat{=}}$ **PRE** Pre **THEN** $v := Post \parallel y := Out$ **END**;
. . . . . .
END

1. The initial state satisfies the invariant.

   $$Inv[v \mapsto Init]$$

2. Each state transition is satisfiable and can change the state.

   $$\exists v, x, y.(Pre \wedge v \neq Post \wedge y = Out)$$

3. Each state query is satisfiable:

   $$\exists v, x, y.(Pre \wedge y = Out)$$

4. Each state transition preservers the invariant (called *invariance lemma*).

   $$\forall x.(Inv \wedge Pre \Rightarrow Inv[v \mapsto Post])$$

   > An invariance lemma states that if the invariant is true of some state and the precondition of an operation is true of the same state, then the invariant is true of the next state as defined by the operation.

   This is why we say that the operation *preserves* the invariant.
   $Inv[v \mapsto Post]$ can be read as "the invariant evaluated in the next state".
   State queries trivially preserve the invariant.

The most important verification conditions are the invariance lemmas. However, if the operation or the invariant are unsatisfiable the invariance lemma will trivially hold. Hence, we also require the other two satisfiability verification conditions.

Here are some examples of the verification conditions with respect to the specification of Figure 3.

- The initial state satisfies the invariant.

  $$\begin{aligned} & sa \in NIC \nrightarrow \mathbb{N}[sa \mapsto \{\}] && \text{[by def. substitution]} \\ \Leftrightarrow\ & \{\} \in NIC \nrightarrow \mathbb{N} && \text{[by empty set is partial function]} \\ \Rightarrow\ & true \end{aligned}$$

  Therefore, we have discharged the proof obligation.

- **deposit** is satisfiable and can change the state. By applying (1) we get:

  $$\begin{aligned} &\exists sa, an, amt, msg. \\ &\quad (an \in NIC \wedge amt \in \mathbb{N} \\ &\quad \wedge (an \in \mathrm{dom}(sa) \wedge 0 < amt \Rightarrow sa \neq sa \lhd\!\!+ \{an \mapsto sa(an) + amt\} \wedge msg = ok) \\ &\quad \wedge (an \notin \mathrm{dom}(sa) \Rightarrow msg = nicNotExists) \\ &\quad \wedge (amt = 0 \Rightarrow msg = amountError)) \end{aligned}$$

  This is satisfied with $sa = \{an \mapsto 0\} \wedge amt = 1 \wedge msg = ok$.

- **checkBalance** is satisfiable.

$$\begin{array}{l}\exists sa, an, bal, msg.\\ \quad (an \in NIC\\ \quad \land\ (an \in \mathrm{dom}(sa) \Rightarrow bal = sa(an) \land msg = ok)\\ \quad \land\ (an \notin \mathrm{dom}(sa) \Rightarrow msg = nicNotExists))\end{array}$$

  This is satisfied with $sa = \{an \mapsto 0\} \land bal = 0 \land msg = ok$.

- **withdraw** preserves the invariant. By considering only the case when sa is changed we get:

$$\begin{array}{ll}\forall an, amt.\\ \quad (sa \in NIC \pfun \mathbb{N} & \text{[invariant]}\\ \quad \land\ an \in NIC \land amt \in \mathbb{N} \land n \in \mathrm{dom}(sa) \land 0 < amt \land amt \leq sa(an) & \text{[precondition]}\\ \quad \Rightarrow sa \ovl \{an \mapsto sa(an) - amt\} \in sa \in NIC \pfun \mathbb{N}) & \text{[invariant in the next state]}\end{array}$$

  Usually we won't write the leading universal quantification leaving it implicit. So the invariance lemma would be:

$$\begin{array}{l}sa \in NIC \pfun \mathbb{N}\\ \land\ an \in NIC \land amt \in \mathbb{N} \land n \in \mathrm{dom}(sa) \land 0 < amt \land amt \leq sa(an)\\ \Rightarrow sa \ovl \{an \mapsto sa(an) - amt\} \in sa \in NIC \pfun \mathbb{N}\end{array}$$

  We leave the proof of this lemma as Exercise 43.

Recall that all these verification conditions must be proved for *all* operations.

As we have said, the abstract assignment isn't a predicate but represents a predicate. The invariance lemma shows how this hidden predicate can be calculated. More precisely, if P is a predicate depending on variable v and we want to know if P is true after $v := expr$ has been executed, then all we have to do is to compute $P[v \mapsto expr]$. This is used in general to determine if a predicate is true of the next state. And this predicate is usually an invariant.

Recall that whenever possible, quantifiers should be avoided in specifications. One of the reasons to do that is, precisely, because proving properties of quantified formulas is, in general, more complex than proving properties of quantifier-free formulas. So if we write our specifications without quantifiers the invariance lemmas for those specifications will tend to be simpler to prove than those of specifications where we use quantifiers. You may already know that propositional logic (i.e. logic without quantifiers) is much simpler than predicate logic (i.e. logic with quantifiers). Furthermore, it can be said that universal quantifiers are 'worse' than existential quantifiers.

On the other hand, it isn't unusual to find out that it's easier and more obvious to write a quantified formula than a quantifier-free one. When writing software specifications the first and foremost option should always be to write down the most obvious formula; the formula that is most obviously correct. Writing formal specifications is always about making something obviously correct, because programs are (almost) never such a thing. Then, we can start with an obviously correct quantified formula ant then we can try to rewrite it without quantifiers.

Along that path we can prove the equivalence of both formulas to be sure that the quantified formula can be substituted by the quantifier-free one.

The whole discussion on avoiding quantified formulas in formal specifications is really important when some form of *computer assisted verification* is going to be applied. That is, avoiding quantified formulas in specifications is of most importance when the verification conditions of these specifications are going to be proved using software tools that are capable of helping or assisting during the proof process. Due to theoretical reasons, these tools are more effective and efficient when the properties they have to prove have the less quantified formulas as possible. Later on this course we'll use the {*log*} tool to automatically discharge proof obligations.

In any case, the mere fact of writing a specification, like the one we wrote in this notes, it's enough to uncover errors, ambiguities and incompleteness in the requirements. Proving properties true of the specification is an added value that would yield marginal gains compared with the gain got by writing the specification itself.

6 Types and sets

In B types and sets aren't clearly distinguished. In this course, however, we will define what sets are types and what aren't. The following sets are types.

- Any set declared in the SETS section is a type.
- Any set declared as a machine parameter is a type.
- The set $\mathbb{Z}$ is a type.
- If X and Y are types then the set $X \times Y$ is a type.
  The elements of this type are ordered pairs.
- If X is type then the set $\mathbb{P}\,X$ is a type.
  The elements of this type are sets.

Sets that can't be built as described above aren't types. By following those rules we can conclude the following:

- $\mathbb{N}$ is not a type.
- If X and Y are types, then $X \leftrightarrow Y$ is a type (because $X \leftrightarrow Y = \mathbb{P}(X \times Y)$).
- If X and Y are types, then $X \nrightarrow Y$ is not a type.
- If X and Y are types, then $X \rightarrow Y$ is not a type.
- If X is a type, then $\mathrm{seq}(X)$ is not a type .

The sets defined as types are so because they are *closed* under certain operations. For example, $\mathbb{Z}$ is a type because if $n, m \in \mathbb{Z}$ then we have that $x + y, x * y, x - y, x \,\mathsf{div}\, y, x \,\mathsf{mod}\, y \in \mathbb{Z}$. Hence, $\mathbb{Z}$ is closed under $+, *, -, \mathsf{div}, \mathsf{mod}$. Instead, $\mathbb{N}$ isn't a type because if $n, m \in \mathbb{N}$ then $x - y \in \mathbb{N}$ isn't necessarily true. Hence, $\mathbb{N}$ isn't closed under subtraction. So it can't be a type (unless we remove subtraction from its operations).

A variable declaration such as:

$$n \in \mathbb{N}$$

is said to be *non-normalized* because $\mathbb{N}$ is not a type but a set. The declaration can be *normalized* by declaring $n \in \mathbb{Z}$ and conjoining $n \in \mathbb{N}$ or $0 \leq x$ in the Invariant section. For example, if we have:

Machine M
. . . . . .
Variables v
Invariant $v \in \mathbb{N} \land \ldots$
. . . . . .
End

we can normalize the machine by writing it as:

Machine M
. . . . . .
Variables v
Invariant $v \in \mathbb{Z} \land 0 \leq v \land \ldots$
. . . . . .
End

The above normalization is preferred over the following:

Machine M
. . . . . .
Variables v
Invariant $v \in \mathbb{Z} \land v \in \mathbb{N} \land \ldots$
. . . . . .
End

because we got rid of an infinite set ($\mathbb{N}$) that's not a type.

Therefore, a variable declaration $x : A$ isn't normalized if A isn't a type but a set. The normalized declaration of x is obtained by substituting A by its *container type* and by conjoining $x \in A$ in the Invariant section. Below we give the container type of some sets frequently used in specifications. Whenever possible avoid the introduction of infinite sets that aren't types.

Set	Container Type
$\mathbb{N}, \mathbb{N}_1, m \mathinner{..} n$	$\mathbb{Z}$
$X \pfun Y, X \fun Y$	$X \rel Y$
$\mathrm{seq}\, X$	$\mathbb{Z} \rel X$

Then, the declaration of *sa* in our specification of the savings accounts isn't normalized:

MACHINE *Bank*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \pfun \mathbb{N}$
.....
END

We can normalize it as follows:

MACHINE *Bank1*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \rel \mathbb{Z} \land sa \in NIC \pfun \mathbb{N}$
.....
END

but we still have $\mathbb{N}$ in the invariant. We can eliminate it as follows:

MACHINE *Bank2*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \rel \mathbb{Z} \land sa \in NIC \pfun \mathbb{Z} \land \forall x, y.(x \mapsto y \in sa \Rightarrow 0 \leq y)$
.....
END

where we change an infinite set by a universally quantified formula. However, we have 'overlapping' constraints: $sa \in NIC \rel \mathbb{Z} \land sa \in NIC \pfun \mathbb{Z}$. In effect, both constraints state that the domain of *sa* is *NIC* and the range is $\mathbb{Z}$. Let's assume there's a predicate, namely *pfun*, which is true only if its argument is a partial function. Hence, we can further rewrite the invariant as follows:

MACHINE *Bank3*
SETS ...
VARIABLES *sa*
INVARIANT $sa \in NIC \rel \mathbb{Z} \land pfun(sa) \land \forall x, y.(x \mapsto y \in sa \Rightarrow 0 \leq y)$
.....
END

In this way, we have decomposed the original invariant into a type invariant ($sa \in NIC \rel \mathbb{Z}$) and two restrictions ($pfun(sa)$ and $\forall x, y.(x \mapsto y \in sa \Rightarrow 0 \leq y)$) that can't be enforced by a typechecker.

Normalization and verification conditions. Normalizing declarations is important when discharging proof obligations. In what concerns to the specification, non-normalized declarations tend to be easier to read and understand. But when the verification conditions of that specification have to be discharged, non-normalized declarations tend to make the process harder and

less automatic. For instance, in $\{log\}$ $x \in \mathbb{N}$ can't be written but $0 \leq x$ will in general be equivalent and will make $\{log\}$ to automatically discharge more proof obligations. Same happens with *Bank*1, *Bank*2 and *Bank*3 above. *Bank*1 can't be written in $\{log\}$ but the other two can be written and $\{log\}$ will tend to automatically discharge more proof obligations.

Furthermore, normalized declarations can be statically typechecked meaning that part of the proof obligations are simply discharged by a typechecker. For instance, $sa \in NIC \leftrightarrow \mathbb{Z}$ can be typechecked in *Bank*2 and *Bank*3. Then, we have easily removed a verification condition.

We'll see this technique in action when $\{log\}$ is introduced.

7 Exercises

1. Consider the state machine with state variables x and y given in Section 1. Write the transitions that connect the 2nd and 3rd states, the 3rd and 4th and the 4th and 1st.
2. Explain why a set of the form $\{1 \mapsto 3\}$ is a binary relation. What is its type? Is it a partial function? If it is, what is its type?
3. Write all the elements of the type $\mathbb{P}\, STATUS$ where $STATUS$ is defined in Section 1.
4. Write a binary relation that is not a partial function.
5. Write three elements of each of the following types or sets, that don't belong to other types or sets in the list.
   (a) $\mathbb{Z}$
   (b) $\mathbb{Z} \times \mathbb{Z}$
   (c) $\mathbb{P}\mathbb{Z}$
   (d) $\mathbb{Z} \leftrightarrow \mathbb{Z}$
   (e) $\mathbb{Z} \nrightarrow \mathbb{Z}$
   (f) $\text{seq}(\mathbb{Z})$
6. Assume T is a type and $x \in T \land a \in \mathbb{P}\, T$. Determine what of the following are type correct. Justify.
   (a) $x = \emptyset$
   (b) $a = \emptyset$
   (c) $x \in a$
   (d) $x \notin a$
   (e) $x \subseteq a$
   (f) $\{x\} \subseteq a$
   (g) $x = a$
   (h) $\{x, x\} = a$
   (i) $\{x\} = a \cup \{x\}$
   (j) $\{x\} = \{a\}$
   (k) $\{\{x\}\} = \{a\}$
   (l) If U is another type and $b \in \mathbb{P}\, U$, $a = b = \emptyset$
7. Prove that if T is a type then $A \in \mathbb{P}\, T \Leftrightarrow A \subseteq T$.
8. Prove that the union of two binary relations is a binary relation.
9. Prove that the union of two partial functions is not necessarily a partial function.
10. Prove that the intersection of two partial functions is a partial function.
11. What is the set corresponding to the following sequence according to the definition of $\text{seq}(X)$ given in Section 1? What type is X in this case?

    $$[3, 54, 12, 3]$$

12. If possible, calculate the following; if not, justify.

(a) $\mathrm{dom}\,\emptyset$
(b) $\mathrm{dom}\{1 \mapsto a\}$
(c) $\mathrm{dom}\{1 \mapsto a, 1 \mapsto b\}$
(d) $\{1 \mapsto a, 2 \mapsto b\}(2)$
(e) $\{1 \mapsto a, 2 \mapsto b\}(3)$
(f) $\{1 \mapsto a, 2 \mapsto b\}(a)$

13. Why the following set isn't a partial function?

$$\{nic_1 \mapsto 100, nic_2 \mapsto 230, nic_1 \mapsto 529\}$$

14. Why if the bank arrives to a state where $n \mapsto b \in sa \land b < 0$ we're in trouble? Same with: $n \mapsto b_1, n \mapsto b_2 \in sa \land b_1 \neq b_2$.
15. Prove that if $sa \in NIC \pfun \mathbb{N} \land n \in NIC$ and $n \notin \mathrm{dom}\, sa$, then $sa \cup \{n? \mapsto 0\} \in NIC \pfun \mathbb{N}$.
16. Complete the definition of the *MSG* type by adding all the messages not initially covered.
17. Prove that if $x \notin \mathrm{dom}\, R$ then $R \domsub \{x \mapsto y\}$ is equal to $R \cup \{x \mapsto y\}$. After that, prove that if $\mathrm{dom}\, S \cap \mathrm{dom}\, R = \emptyset$ then $R \domsub S = R \cup S$.
18. Prove that if $f : X \pfun Y$; $x : X$; $y : Y$ then $f \domsub \{x \mapsto y\} : X \pfun Y$.
19. Determine the conditions under which $R \domsub S$ is a function.
20. Prove that if $R : X \rel Y$ and $x \notin \mathrm{dom}\, R$, then $\{x\} \domsub R = R$.
21. Prove that if $R : X \rel Y$ and $A : \mathbb{P}\, X$ then $(A \domres R) \cap (A \domsub R) = \emptyset$.
22. Prove that if $R : X \rel Y$ and $A : \mathbb{P}\, X$ then $R = (A \domres R) \cup (A \domsub R)$.
23. Replace the following **IF-THEN-ELSE** with a **SELECT** statement.

$$\textbf{IF } an \notin \mathrm{dom}(sa) \textbf{ THEN } sa, msg := sa \cup \{an \mapsto 0\}, ok \textbf{ ELSE } msg := nicExists \textbf{ END}$$

24. Explain why we have included $an \in \mathrm{dom}(sa)$ as a precondition in the third branch of **close**.
25. The postcondition of **checkBalance** in the **ELSE** branch doesn't set a value for *bal*. Is it necessary? If so, what could be that value?
26. Let's say the bank requires that when an account is opened the customer must deposit at least 200 euros. Specify this requirement.
27. Specify an operation that outputs a listing of the balances of a group of savings accounts.
28. Specify an operation that closes more than one account.
29. In the specification of **deposit** developed in Section 4, there are branches where no value for *ra* and *rn* is defined. What does it mean? What would be the correct behavior in those branches?
30. In relation to the previous exercise, modify the specification of **deposit** as to output a set of pairs of the form $rn \mapsto ra$. What is the value for the new output parameters in the branches analyzed in the previous problem?

31. Study the CONSTANTS and PROPERTIES sections of a B machine. Define a constant of type *NIC* whose informal meaning is "no account number". Rewrite the specification of **deposit** as to output this constant when no report should be sent to the central bank. What value for *ra* would you use in this case?

32. In relation to the previous exercise, what if **deposit** outputs -1 in *ra* when no report should be sent to the central bank? Is it necessary to define a value for *rn* in those cases?

33. Continuing with the same problem, do the following:
    (a) Underspecify a function that converts *NIC* into strings.
    (b) Underspecify a function that converts $\mathbb{Z}$ into strings.
    (c) Underspecify the empty string
    (d) Redefine **deposit** as to output two strings instead of a *NIC* and a $\mathbb{N}$.
    (e) Redefine **deposit** as to output the empty string in every other case.
    (f) Complete the specification of **deposit** with the abnormal behavior that is missing.

34. Redo the above problem as follows:
    (a) Declare the set-type *REPORT*.
    (b) Declare *empty* as a constant of type *REPORT*.
    (c) Underspecify a function that converts $NIC \times \mathbb{N}$ into *REPORT*.
    (d) Redefine **deposit** as to output a *REPORT*.

35. In the above two exercises we asked you underspecify some functions. Is it enough with functions? Don't you need more? What if one of these functions is defined as $\{n_1 \mapsto \text{'n1'}, n_2 \mapsto \text{'n1'}\}$ (i.e. the function mapping *NIC* onto strings)?

36. Analyze what is the best solution for outputting a report to the central bank. What should be the criterion to decide what is the best solution?

37. Assume the central bank forbids withdrawals of more than 200 000 euros. Specify this requirement.

38. Specify an operation transferring an amount of money from one account in the bank to another (different) account in the same bank.

39. Extend the specification of the savings account system where a history of the transactions (deposits and withdrawals) made on every account must be recorded. Keep the state variable *sa* defined in the specification developed in these notes. The specification must include the following operations: open an account, deposit, withdraw and check balance.

40. Concerning problem 39, can you write an operation showing the transactions made on an specific account between two dates? If you can, do it; if not, rewrite the model having this requirement in mind.

41. Extend the specification of the savings account system where a customer may have more than one savings account. The specification must include the following operations: open and account, deposit, withdraw, check balance, close an account and close all accounts of a given customer.

42. Check that $sa = \{an \mapsto 0\} \land amt = 1 \land msg = ok$ satisfies the following verification condition.

$$\begin{array}{l}\exists sa, an, amt, msg. \\ \quad (an \in NIC \land amt \in \mathbb{N} \\ \quad \land (an \in \mathrm{dom}(sa) \land 0 < amt \Rightarrow sa \neq sa \lhd\!\!+ \{an \mapsto sa(an) + amt\} \land msg = ok) \\ \quad \land (an \notin \mathrm{dom}(sa) \Rightarrow msg = nicNotExists) \\ \quad \land (amt = 0 \Rightarrow msg = amountError))\end{array}$$

43. Discharge the following invariance lemma.

$$\begin{array}{l}sa \in NIC \pfun \mathbb{N} \\ \land\, an \in NIC \land amt \in \mathbb{N} \land n \in \mathrm{dom}(sa) \land 0 < amt \land amt \leq sa(an) \\ \Rightarrow sa \lhd\!\!+ \{an \mapsto sa(an) - amt\} \in sa \in NIC \pfun \mathbb{N}\end{array}$$

44. Explain why $X \pfun Y$ can't be a type.
45. Prove the following:

$$(\forall x, y.(x \mapsto y \in sa \bullet 0 \leq y)) \Leftrightarrow (\exists k.(k \in \mathbb{Z} \land \mathrm{ran}(sa) \subseteq 0 \mapsto k))$$

46. Prove that **withdraw** preserves the invariant.
47. Write and prove all the remaining proof obligations concerning the specification of the savings account system.
48. Write a state invariant for the specification developed in 39 relating the current balance and the history of transactions of every account.
49. Write and prove all the proof obligations for the specification developed in 39.
50. Write a Java program implementing **open**.
51. Concerning exercise 50, what are the differences between the program and the specification? How did you implement *NIC*? Can you *guarantee* that your implementation verifies the specification? How would you do that?