3 B: A Model-Based Method Using Generalised Substitutions

Hassan Diab and Marc Frappier

3.1 Overview of the B Notation

The B notation [1] was developed by Jean-Raymond Abrial. It supports a large segment of the development life cycle, from specification to implementation. The B notation is formal: it has an axiomatic semantics based on the weakest-precondition calculus of Dijkstra [6]. Abrial has significantly extended the initial set of Diskstra's guarded commands, proposing a complete specification and design notation scalable to large system development. Some of these extensions are inspired from the work of the programming research group at Oxford (*e.g.,* [7]). The B notation is closely related to the Z notation and the VDM notation (Abrial was a strong contributor to the development of Z).

The basic building block of B specifications is the notion of an *abstract machine*. Such a construct serves to encapsulate suitable (set-theoretic) *state variables*, the values of which must always satisfy its *invariant* (stated as a predicate). The behavioural aspects are specified in terms of an *initialisation*, and a set of *operations* that may be used to access or modify this abstract state.

One distinctive characteristic of the B method is that every such specification is validated by means of (automatically generated) *proof obligations*. At the level of an abstract machine the main proofs ensure that its initialisation *establishes* the specified invariant, and that this is then *preserved* by any calls to its associated operations (see Section 3.4).

Both the initialisation and each individual operation are defined using *generalised substitutions*. Such substitutions are similar to conventional 'assignment statements', but with a well-defined (mathematical) semantics. They identify which variables are modified, without mentioning those that are not. The generalisation proposed by Abrial allows the definition of non-deterministic specifications, guarded specifications and *miraculous* specifications. For some initial state, a miraculous specification may terminate in a state which satisfy any predicate, including **false**. Obviously, a miraculous specification is not implementable.

Large machines are constructed using smaller machines through various machine access relations. A machine may *include*, *use*, *see*, *import* or *extend* other machines. Each access relation imposes constraints on the access, from the referencing machine, to the various parts of the referenced machine. Encapsulation is supported by allowing the modification of the state variables only through the operations of a machine; on the other hand, a state variable may be read by the referencing machines in some of the access relations. Other details on the B notation will be provided as the case study specifications are presented.

There are two commercially available case tools that support the B notation: Atelier B [8] and the B tool [5]. They both provide syntax checkers, theorem provers and document management facilities. For this case study, we used Atelier B.

3.2 Analysis and Specification of Case 1

There are several ways of tackling a specification problem with the B notation. In this chapter, we start by identifying the operations required from the system according to the user requirements; operations provide the inputs, the outputs and the relationship between. Alternative approaches [1] begin with the identification of the state variables and their invariant.

3.2.1 Identifying Operations

The first question one should ask to conduct the analysis is:

Question 1: What operations are required from the system?

Answer: The only operation required is **invoice_orders**.

The next question is:

Question 2: What are the input parameters of this operation?

Answer: The user requirements provides that orders are invoiced according to the state of the stock. There are two options:

1. invoice all pending orders;
2. invoice a subset of the pending orders.

We select option 2. Therefore, a set of orders is an input parameter of the operation.

This choice raises another question:

Question 3: In what sequence should the orders of this set be processed?

Answer: The processing sequence is important, because it may affect the ability to invoice a particular order. For instance, assume that two orders reference the same product and that there is enough stock for only one of them. Depending on the sequence in which these orders are processed, one order will be invoiced and the other will remain in the state pending, since there is not enough stock left after invoicing the first order. There are three options:

1. nondeterministically select a sequence in the set of orders;

2. accept as input a sequence of orders instead of a set of orders;
3. accept as input a single order.

We select option 3, because of its simplicity. Hence, we assume that the operation takes one order in parameter instead of a set of orders; the user has to invoke the operation once for each order, thereby specifying his preferred sequence of processing for orders.

Note that it is possible in B to specify nondeterministic operations; hence, either option in the answer above can be chosen. However, specifying the invoicing of a sequence of orders is more difficult (significantly) than specifying the invoicing of a single order. This fact may seem surprising, because a natural reflex would be to specify the invoicing of a sequence of orders by a loop over the sequence elements and invoicing them one by one. But, loop statements and sequential composition (";") are not allowed at the specification level in B. They are only allowed in implementations.

Question 4: How does the user specify the order?

Answer: There are two options:

1. submit as input a complete order with all its product references;
2. submit as input the order number.

We select option 2 since it is the most practical solution from a user perspective.

Question 5: How does the user specify the stock?

Answer: We assume that the stock is not an input parameter. Rather, the operation uses the current status of the stock, which means that the stock is accessed through state variables.

Question 6: Does the operation have output parameters?

Answer: We assume that the operation has a single output, a response code, indicating if the order was successfully invoiced.

3.2.2 Defining the State Space

Before proceeding with the definition of the operation body, we must define the state space of the specification. We mentioned previously that the main building block of a B specification is the machine construct. The next decision to take is to determine how many machines are required. This is an internal issue which does not involve the user. For the sake of reusability and maintainability, it is desirable to define highly cohesive machines. An appropriate solution is to create two machines, *Product1* and *Invoicing1*. We first provide the definition of the *Invoicing1* machine. The first clause provides the machine name.

MACHINE

Invoicing1

The next clause defines the sets which are used as types for state variables and operation parameters. We may ask the following questions to the user.

Question 7: What are the possible values for the order number, the product number, the ordered quantity, the order status and the output message?

Answer: The user requirements are not specific about the type of these elements. We choose to defer the definition of a type for an order number and a product number. We assume that the ordered quantity is a natural number. An order status is either *Order_pending* or *Order_invoiced*. An output message is either *Updated* or *Not_updated*.

In B, we may defer the actual definition of types to implementation. In the SETS clause given in the sequel, we define the possible values for the status of an order and the output messages. The specification of the possible values for order are deferred to implementation. The **INVARIANT** clause will assign a type to each state variable.

Question 8: How many products does an order reference?

Answer: The user requirements are rather ambiguous about this issue. For the following statement of the user requirements

> "On an order, we have one and only one reference to an ordered product of a certain quantity. The quantity can be different to other orders.",

we see two interpretations:

1. an order contains exactly one product reference;
2. an order may contain several product references, but each product is referenced only once per order.

We select option 2. In the sequel, the word *item* denotes the reference of a product on an order.

The next clause defines the sets of machine *Invoicing1*.

SETS

$ORDER$;
$STATUS = \{\ Order\_pending,\ Order\_invoiced\ \}$;
$RESPONSE = \{Updated,\ Not\_updated\}$

The set *ORDER* contains the set of valid order numbers. The definition of the elements of set *ORDER* is deferred to the implementation of this machine. The other sets, *STATUS* and *RESPONSE*, are defined by enumeration. The former represents the possible values for the status of an order while the latter represents possible outputs for operation **invoice_order**. The sets of a **SETS** clause are assumed to be finite and non-empty.

The next clause, **INCLUDES**, provides that machine *Invoicing1* has direct read access to the state variables of machine *Product1*, and that it may invoke *Product1* operations to modify *Product1* state variables. The definition of machine *Product1* is given in Section 3.2.4.

INCLUDES

Product1

The include relationship is transitive for variables: if some machine *M* includes machine *Invoicing1*, it can access state variables of *Product1*. The include relationship is *not* transitive for operations. Clause **PROMOTES** *op_name* may be used in machine *Invoicing1* to state that operation *op_name* from machine *Product1* is also an operation of machine *Invoicing1*.

The state of machine *Invoicing1* is defined using four variables given in the **VARIABLES** clause. Each variable is given a type in the **INVARIANT** clause. It represents a possible formalisation of the answers to Question 7 and Question 8.

VARIABLES

order, *status*, *item*, *ordered_qty*

INVARIANT

$$\begin{array}{l} order \subseteq ORDER \wedge \\ status \in order \rightarrow STATUS \wedge \\ item \in order \leftrightarrow product \wedge \\ ordered\_qty \in item \rightarrow \mathbf{NAT} \end{array}$$

Variable *order* contains the set of order numbers currently in the system. It is a subset of set *ORDER* (the set of all valid order numbers, as mentioned earlier). Variable *status* is a total function ($\rightarrow$) from *order* to *STATUS*; it provides the status of an order. Variable *item* is a relation ($\leftrightarrow$) between *order* and *product*. Variable *product* is defined in machine *Product1*, which will be described in Section 3.2.4. Variable *ordered_qty* provide the ordered quantity of an item.

If the reader is accustomed to the formal notation for relational database specification, he might find the following alternative state space definition more natural.

VARIABLES

$order, item$

INVARIANT

$order \subseteq ORDER \times STATUS \wedge$
$item \subseteq ORDER \times PRODUCT \times \mathbf{NAT}$

These definitions provide that *order* and *item* are relations. In B, one must use projection functions $\mathbf{prj}_1$ or $\mathbf{prj}_2$ to access a particular *coordinate* (*i.e.,* an *attribute* in relational database terminology) of a tuple of a Cartesian product. That makes specifications less explicit, thus harder to read and understand. Moreover, the integrity constraints that the order number is unique (primary key) and that a couple order number and product is also unique are already catered for in the first definition. Consequently, the first definition of the state space is preferred.

The **INITIALISATION** clause defines the initial state of the *Invoicing1* machine.

INITIALISATION

$order := \varnothing \mid\mid$
$status := \varnothing \mid\mid$
$item := \varnothing \mid\mid$
$ordered\_qty := \varnothing$

Each variable is assigned a value using an *elementary substitution*. An elementary substitution is of the form $v := t$, where v is a state variable or an operation output parameter, and t is a term. An elementary substitution behaves like an assignment statement: after the execution, the new value of v is t; the other variables of the machine are not modified; the state variables in t refer to the value before the execution. In this case, we have chosen to initialise each variable to empty. In B, a function is represented by a set of pairs (*i.e.,* a deterministic binary relation). The operation "||" denotes the simultaneous execution of all the elementary substitutions.

3.2.3 Defining the Behaviour of the Invoicing Operation

We may now define the body of operation **invoice_order**. The following questions are raised.

Question 9: What are the necessary conditions to invoice an order?
Answer: According to the user requirements, the system can invoice an order if:

1. its status is pending;

2. it contains at least one product reference;
3. there is enough stock for each product reference of the order.

Question 10: What is the result of the operation if the previous conditions are satisfied?

Answer: According to the user requirements, we have:

1. the status is set to invoiced;
2. the items of the order are removed from the stock.

In addition, we assume that the output message "*Updated*" is issued.

Question 11: What is the result of the operation if the previous conditions are not satisfied?

Answer: We assume that:

1. the system state is unchanged;
2. the output message "*Not_updated*" is issued.

We provide below the specification of the operation according to these answers.

```
OPERATIONS

$response \leftarrow \mathbf{invoice\_order}(oo) \triangleq$

    PRE
        $oo \in ORDER$
    THEN
        IF
            $status(oo) = Order\_pending \wedge$
            $oo \in \mathbf{dom}(item) \wedge$
            $\forall\, pp.(\; pp \in product \;\; \wedge \;\; (oo \mapsto pp) \in item$
                    $\Rightarrow$
                    $ordered\_qty(oo \mapsto pp) \leq quantity\_in\_stock(pp))$
        THEN
            $\mathbf{status}(oo) := Order\_invoiced \;||$
            $\mathbf{decrease\_stock}($
                $\lambda\, pp.(\; pp \in product \;\; \wedge \;\; (oo \mapsto pp) \in item$
                        $|$
                        $ordered\_qty(oo \mapsto pp)))\;||$
            $response := Updated$
        ELSE
            $response := Not\_updated$
        END
    END
```

The operation has an input parameter, *oo*, and an output parameter, *response*. Parameter *response* is set to *Updated* if the order was successfully invoiced, otherwise it is set to *Not_updated*.

To write a complex operation that modifies several variables, elementary substitutions are combined using compound substitutions. The main substitution of operation **invoice_order** is a *precondition* substitution of the form **PRE** p **THEN** S **END**, where p is a predicate and S is a substitution. This construct means that the substitution (corresponding to S) is only well-defined when p holds – which gives rise to a (static) proof obligation in the context of each separate call (as opposed to a 'run-time' test). A minimal precondition for an operation must specify at least the 'types' of its input parameters, if any, but as shown in the sequel, additional constraints may be introduced as well.

The **THEN** part of the precondition substitution is expressed as a *conditional* substitution of the form **IF** p **THEN** S_1 **ELSE** S_2 **END**, where p is a predicate, and S_1 and S_2 are substitutions. Such a construct has the same meaning as in conventional programming language. The condition of the **IF** contains three conjuncts which refer to the three conditions raised in the answer of Question 9. Two variables of machine *Product1* are referenced: *product*, which denotes the set of product numbers currently in the system; *quantity_in_stock*, which denotes the number of product units in inventory for a product number.

The **THEN** part of the **IF** contains a multiple substitution of the form $S||T$. Substitutions S and T are executed simultaneously. Note that the first elementary substitution is of the form $f(xx) := t$; it is an abbreviation of the substitution $f := f \mathbin{\lhd\!\!\!+} \{(xx, t)\}$, where $\mathbin{\lhd\!\!\!+}$ is the override operation for relations (recall that a function is represented by a deterministic binary relation). Operator $\mathbin{\lhd\!\!\!+}$ is defined as follows using operators $\lhd$ (domain restriction), $\mathbin{\lhd\!\!\!-}$ (domain subtraction), and $\mapsto$ (pair construction). Let r and s be relations and A be a set; we have

$$A \lhd r \triangleq \{x, y \mid x \mapsto y \in r \wedge x \in A\}$$

$$A \mathbin{\lhd\!\!\!-} r \triangleq (\mathbf{dom}(r) - A) \lhd r$$

$$r \mathbin{\lhd\!\!\!+} s \triangleq (\mathbf{dom}(s) \mathbin{\lhd\!\!\!-} r) \cup s \ .$$

The next substitution of the **THEN** clause is a call to operation **decrease_stock** of machine *Product1*. This operation accepts one parameter, a partial function f from *product* to **NAT**, and reduces the stock of $f(pp)$ units for each product pp in the domain of f. The argument provided in the operation call is a function defined using a lambda abstraction $\lambda x.(p \mid e)$. It denotes a function f whose domain is the set of x such that p holds and the image of x is given by expression e.

3.2.4 The Product1 Machine

The *Invoicing1* machine includes the *Product1* machine. Its definition is given below.

MACHINE

Product1

SETS

PRODUCT

VARIABLES

product, *quantity_in_stock*

INVARIANT

$product \subseteq PRODUCT \wedge$
$quantity\_in\_stock \in product \rightarrow \mathbf{NAT}$

INITIALISATION

$product := \varnothing \ ||$
$quantity\_in\_stock := \varnothing$

The *Product1* machine would be better encapsulated if we had defined an operation to access the quantity in stock. However, it would be useless in this case to define such an operation, because the B notation does not allow a call to an operation in the predicate accessing the quantity in stock in operation **invoice_order**. The encapsulation mechanism of B may seem weaker than those typically found in an object-oriented programming language, where it is possible to prevent an external access to class variables. However, encapsulation is fostered at a different level of abstraction in B. A machine may be refined and implemented using machines with completely different state variables, as long as they preserve the signature of the operations and their observable behaviour. A machine M is refined by a machine N, noted $M \sqsubseteq N$, if and only if, for any sequence of operation calls where machine M terminates, machine N also terminates and delivers a result that machine M can deliver. Hence, machine N refines machine M by possibly extending the set of call sequences where M terminates and by possibly reducing the nondeterminacy of M.

Machine *Product1* has only one operation, **decrease_stock**, which is invoked from machine *Invoicing1* when an order is invoiced, or when product units are removed from the inventory.

OPERATIONS

```
**decrease_stock**$(\mathit{prod\_qty}) \triangleq$

    PRE
      $\mathit{prod\_qty} \in \mathit{product} \mathrel{+\!\!\!\rightarrow} \mathbf{NAT} \;\wedge$
       $\lambda\, xx.(\; xx \in \mathbf{dom}(\mathit{prod\_qty}) \mid \mathit{quantity\_in\_stock}(xx) - \mathit{prod\_qty}(xx))$
          $\in \mathit{product} \mathrel{+\!\!\!\rightarrow} \mathbf{NAT}$
    THEN
      $\mathit{quantity\_in\_stock} := \mathit{quantity\_in\_stock} \mathbin{\lhd\!\!\!-}$
          $\lambda\, xx.(\; xx \in \mathbf{dom}(\mathit{prod\_qty}) \mid \mathit{quantity\_in\_stock}(xx) - \mathit{prod\_qty}(xx))$
    END
```

Operation **decrease_stock** has one input parameter, *prod_qty*. The first conjunct of the precondition provides that this parameter is a partial function ($\mathrel{+\!\!\!\rightarrow}$) from *product* to the set of natural numbers. For each product *pp* in the domain of function *prod_qty*, the operation must reduce the quantity in stock by *prod_qty*(*pp*) units. The override of the quantity in stock is carried out with a function defined by a lambda abstraction.

To preserve the invariant of machine *Product1*, which provides that the quantity in stock is a natural number, we must verify in the precondition of **decrease_stock** that there are enough units in inventory for each product in the domain of function *prod_qty*. When an operation defined using a **PRE** *p* **THEN** *S* **END** is called, it is the responsibility of the caller to ensure that the operation is invoked in a state where *p* is satisfied. Otherwise, the operation call *may* abort (it *may* terminate because the implementation of an operation is allowed to weaken the precondition defined in the abstract machine). To prove that an operation *op* preserves the invariant, it is also necessary to prove that the precondition of each operation called by *op* is satisfied.

Note that we could have specified this conjunct in an **IF** substitution within the **THEN** part of the **PRE** substitution. In that case, the substitution would terminate normally without modifying the inventory if there was not enough stock. It would then be natural to add an output parameter to the operation indicating if the inventory has been successfully modified, like we did for operation **invoice_order**.

Several specification styles may be used in B. A typical B specification is structured into 'layers' of machines. An interface layer defines the interaction with the environment using input-output operations. This layer reads inputs from the environment, validates them, calls appropriate operations of machines from an object layer to compute the responses (outputs) and to update the state of the objects, and writes the responses to the environment.

Our specification of the invoicing case study does not include an interface layer. We only specify machines of the object layer. Moreover, our specifications are incomplete, as they do not contain all the operations that would be expected for a complete system. For instance, we have omitted an operation to add a product to the set of products (variable *product*). The next chapter also presents a B specification, which is derived from an OMT object model. Its object layer is more structured than the one present in this chapter.

3.3 Analysis and Specification of Case 2

Case 2 is an extension of Case 1. We have defined new machines, *Product2* and *Invoicing2*, which have the same state space (state variables and invariant) as the machines in Case 1, but we have added to these machines operations to increase stock and to manage orders. In the sequel, we identify the operations and provide their specifications.

3.3.1 Identifying Operations

Question 12: What are the operations required?

Answer: Considering the user requirements of Case 2, we have identified the following operations in addition to the operations of Case 1:

- **increase_stock**, which is the inverse of the **decrease_stock** operation; it takes a set of items and increases the quantity in stock for these items;
- **create_order**, which creates an order;
- **add_item**, which adds an item to an order;
- **cancel_order** and **cancel_item**, which are the inverse of the previous two operations. We assume that these operations only modify pending orders; invoiced orders cannot be modified.

No other operation is needed, considering the given requirements. Note that there is no operation to create a new product or to delete a product from the stock.

3.3.2 The Product2 Machine

Operation **decrease_stock** is the same as in machine *Product1*; hence we omit its definition. Operation **increase_stock** is similar to **decrease_stock**. Its definition is given below.

```
increase_stock(prod_qty) ≜

    PRE
        prod_qty ∈ product ⇸ NAT ∧
        λ xx.( xx ∈ dom(prod_qty) | quantity_in_stock(xx) + prod_qty(xx))
            ∈ product ⇸ NAT
    THEN
        quantity_in_stock := quantity_in_stock ⩤
            λ xx.( xx ∈ dom(prod_qty) |
                quantity_in_stock(xx) + prod_qty(xx))
    END
```

3.3.3 The Invoicing2 Machine

Operation **invoice_order** is the same as in *Invoicing1*; we omit its specification. Operation **create_order** uses a nondeterministic substitution, the unbounded choice (clause **ANY-WHERE-THEN-END**), to pick a value for local variable oo that satisfies the condition $oo \in ORDER - order$. This order number is then used to create a new order whose status is pending. The definition of this operation is given below.

```
OPERATIONS

$response \leftarrow$ create_order $\triangleq$

    IF
        $order \neq ORDER$
    THEN
        ANY  $oo$ WHERE
            $oo \in ORDER - order$
        THEN
            $order := order \cup \{ oo \} \;||$
            status($oo$) $:= Order\_pending \;||$
            $response := Updated$
        END
    ELSE
        $response := Not\_updated$
    END
```

The next operation adds an item to an order. It updates the state if and only if the order status is pending and if the product of the item is not already referenced on the order. Its definition is very similar to operation **create_order**.

```
$response \leftarrow$ add_item($oo$, $pp$, $qq$) $\triangleq$

    PRE
        $oo \in ORDER \wedge$
        $pp \in PRODUCT \wedge$
        $qq \in$ NAT
    THEN
        IF
            $oo \in order \wedge$
            $status(oo) = Order\_pending \wedge$
            $pp \in product \wedge$
            $(oo,pp) \notin item$
        THEN
            $item := item \cup \{oo \mapsto pp\} \;||$
            ordered_qty($oo \mapsto pp$) $:= qq \;||$
            $response := Updated$
        ELSE
            $response := Not\_updated$
        END
    END
```

The next operation, **cancel_order**, removes a pending order from the set of orders. It must update all variables related, by the invariant, to the set *order* and the relation *item*.

```
$response \leftarrow$ cancel_order$(oo) \triangleq$

    PRE
        $oo \in ORDER$
    THEN
        IF
            $oo \in order \wedge$
            $status(oo) = Order\_pending$
        THEN
            $order := order - \{oo\} \;||$
            $item := \{oo\} \lhd\!\!- \; item \;||$
            $ordered\_qty := (\{oo\} \lhd\!\!- \; item) \lhd ordered\_qty \;||$
            $status := \{oo\} \lhd\!\!- \; status \;||$
            $response := Updated$
        ELSE
            $response := Not\_updated$
        END
    END
```

Operation **cancel_item** is very similar to operation **cancel_order**.

```
$response \leftarrow$ cancel_item$(oo, pp) \triangleq$

    PRE
        $oo \in ORDER \wedge$
        $pp \in PRODUCT$
    THEN
        IF
            $oo \in order \wedge$
            $status(oo) = Order\_pending \wedge$
            $pp \in product \wedge$
            $(oo,pp) \in item$
        THEN
            $item := item - \{oo \mapsto pp\} \;||$
            $ordered\_qty := \{oo \mapsto pp\} \lhd\!\!- \; ordered\_qty \;||$
            $response := Updated$
        ELSE
            $response := Not\_updated$
        END
    END
```

3.4 Validation of the Specification

We have mentioned previously that operations must preserve the invariant. The B method defines proof obligations for each operation and for initialisation substitutions. Discharging these proof obligations provides a form of specification validation. As an example, the following predicate is part of a proof obligation (a simplified version) for operation **create_order**.

$$
\begin{array}{ll}
(1) & order \subseteq ORDER \wedge \\
(2) & order \neq ORDER \wedge oo \in ORDER \wedge oo \notin order \\
 & \Rightarrow \\
(3) & order \cup \{oo\} \subseteq ORDER
\end{array}
$$

This predicate provides that when the invariant holds (1) and when the conditions of the **PRE** and **ANY** clauses hold (2), the substitution applied to the invariant must also hold (3). In other words, after adding *oo* to *order*, *order* must still be a subset of *ORDER*.

We have used Atelier B to generate all the proof obligations and to conduct the proofs. Its theorem prover has automatically discharged all proof obligations except one – which was very easy to prove in interactive mode. Interactive proofs may represent a fair challenge. When the prover fails to find a proof, one must determine whether there is something wrong in the specification or if the prover is simply unable to find a proof. When the specification seems correct, one must build a proof in interactive mode. This task requires a good knowledge of the proof rules used by the prover and the different ways of applying them. It is sometimes necessary to rewrite specifications in a different manner to obtain proof obligations which are easier to discharge with the theorem prover. Difficult proof obligations are usually good hints that the specification needs to be rewritten in a simpler manner.

Table 3.1 provides a summary of the proof obligation statistics.

We have found one defect in our specification with the theorem prover. In the precondition of operation **increase_stock**, we had forgotten to check that, for each product, the number of product units plus the quantity in stock did not exceed **MAXINT**. We found several defects with the type checker of Atelier B. Before using the prover, we conducted several inspections and walkthroughs of the specification which allowed us to find various defects.

Table 3.1. Proof obligation statistics

Machine	Proof Obligations	Automatic Proofs	Interactive Proofs
Product1	5	5	0
Invoicing1	7	7	0
Product2	7	7	0
Invoicing2	22	21	1
Total	41	40	1

3.5 The Natural Language Description of the Specifications

3.5.1 Case 1

An order has a number, a status, and items. The status may be *Order_pending* or *Order_invoiced*. An item is reference to a product in an order. Each item has an ordered quantity given by a natural number. Among the items of a given order, there must not be two references to the same product. The stock consists of a set of products. A quantity in stock, given as a natural number, is associated to each product.

The system provides an operation, **invoice_order**, which accepts an order number as input, and produces an information message as output. This operation behaves as follows. If the order status is *Order_pending*, if it has at least one item, and if, for each item, the ordered quantity is greater or equal to the quantity in stock, then the order status is changed to *Order_invoiced*, the quantity in stock for each product referenced in an item is decreased by the ordered quantity, and the information message is set to *Updated*; otherwise, the order and the stock are left unchanged, and the information message is set to *Not_updated*.

To invoice a set of orders, the user must invoke operation **invoice_order** once for each order, in the sequence he prefers. There is no concurrency in the system: it is assumed that operations are invoked in sequence.

3.5.2 Case 2

It is an extension of Case 1. The definitions of orders and stock are the same as in Case 1. New operations are provided. Operation **create_order** creates an order with an empty set of items. The order must not *exist* in the system, that is, it has never been created, or it has been created then deleted. Operation **add_item** adds an item to an order. The product reference must not exist in the order. Operation **cancel_order** and **cancel_item** are the inverses of operations **create_order** and **add_item**, respectively. These last three operations update the system state only if the order status is *Order_pending*; invoiced orders cannot be modified.

3.6 Conclusion

The elicitation of the invoicing user requirements using B lead us to a more precise statement of the expected system functions. We had to specify the inputs, the outputs, the state space and the relation between them. The fact that we have used a formal language does not prevent us from creating incorrect description of the user requirements; it only allows us to make precise statements which can then be systematically validated to determine if they are appropriate.

Mathematics provided a common language for resolving arguments and discussions between the authors during the validation. This is a significant improvement over classical informal methods like structured analysis [9] or object-oriented analysis [3]. The precise semantics of the B notation and its powerful data abstractions like sets, functions and relations allowed us to identify exactly what information the system could convey and the exact behaviour of the operations transforming this information. Using mathematics and the B notation helped dispel ambiguities and misunderstandings in matching the user requirements with the specification. The B notation, as described in [1], does not allow to model concurrency or dynamic constraints. We refer the reader to [4] for a treatment of concurrency and [2] for the specification of dynamic constraints.

Readability is one of the weaknesses of a formal notation like B. It comprises a large array of symbols, some of them which are not common in ordinary mathematics. Moreover, the “structure” of a state space is not as easy to grasp in a B specification as it is in a graphical notation like an entity-relationship (E-R) model. For instance, consider an order and an item. In an E-R model, they would be represented as two entities with a relationship between them. The attributes would be listed on each entity. The same information in a B specification is given in a flat list of predicates. It takes more time to get a good mental representation of the information structure in a B specification than with a graphical E-R model.

Acknowledgements

The authors would like to thank Henri Habrias and Pierre Levasseur for useful suggestions on improvements to the specification.

References

1. Abrial J.-R. (1996) *The B-Book*. Cambridge University Press
2. Abrial J.R., Mussat L. (1998) Introducing Dynamic Constraints in B, in Bert D. (Ed.) *B'99: Recent Advances in the Development and Use of the B Method*. LNCS 1393, Springer-Verlag, 83–128.
3. Booch. G. (1994) *Object-Oriented Analysis and Design with Applications*. 2nd edition, Benjamin-Cummings

4. Butler M., Waldén M. (1996) Distributed System Development in B, in Habrias H. (Ed). *First Conference on the B Method*. Institut de Recherche en Informatique de Nantes, Nantes, France, 155–168.
5. B-Core Limited: Oxford, United Kingdom, `http://www.b-core.com`
6. Dijkstra E.W. (1976) *A Discipline of Programming.* Prentice Hall
7. Morgan C. (1990) *Programming from Specifications.* Prentice Hall
8. Stéria Méditerranée: Aix-en-Provence, France, `http://www.atelierb.societe.com`
9. Yourdon E. (1989) *Modern structured analysis.* Yourdon Press