The Event B Modelling Method

Designing a Small System
The Invoice System

Erasmus Exchange December 8, 2011
NUI Maynooth, Ireland

Dominique Méry

Université Henri Poincaré Nancy 1 and LORIA
BP 70239,Campus Scientifique
54506 Vandoœuvre-ls-Nancy Cedex, FRANCE
dominique.mery@loria.fr
http://www.loria.fr/∼mery
edited on December 7, 2011

Contents

Chapter 1

The Invoice System

1.1 Introduction

What is the event B method? In the sequel, we refer to the original B method as classic B [ABR 96] and its event-based evolution as event B. The event B method [ABR 98, ABR 03a] reuses the set-theoretical and logical notations of the B method [ABR 96] and provides new notations for expressing abstract systems or simply models based on events. Moreover, the refinement over models is a key feature for incrementally developing models from a textually-defined system, while preserving correctness; it implements the proof-based development paradigm. Each development. includes proofs for invariance and refinement. Operations of the classic B method do not exist in the event B method and are substituted by events. Events modify the system state (or state variables), by executing an action, but if a guard holds: nn event is not called but observed. When refining machines in classic B, one should maintain the number of operations both in the abstract machine and in the refinement; on the contrary, new events may be introduced in the refinement model and they may modify only new variables. New events bring new proof obligations for ensuring a correct refinement. Finally, an event B model is a closed system with a finite list of state variables and a finite list of events. If the system reacts to its environment, the event B model should integrate events of the environment. The B chapter introduces useful notations for the event B method like set theory, generalized substitution, predicate calculus.

Proof-based development. Proof-based development methods integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. We then gradually add details to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [BAC 79, ABR 96, CHA 88]. It is controlled by means of a number of, so-called, *proof obligations*, which guarantee the correctness of the development. Such proof obligations are proved by automatic (and interactive) proof procedures supported by a proof engine. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination properties. The invariant of an abstract model plays a central role for deriving safety properties and our methodology focuses on the incremental discovery of the invariant; the goal is to obtain a formal statement of properties through the final invariant of the last refined abstract model.

Refining formal models. Formal models contain *events* which preserve some invariant properties; they also include aspects related to the termination. Such models are thus very close to action systems introduced by Back [BAC 79] (see Chapter 8), to UNITY programs [CHA 88] and to TLA$^+$ specifications [LAM 94, LAM 02]. The refinement of formal models plays a central role in these frameworks and is a key concept for developing (sequential, distributed, communicating, ...) (computer-based) systems. When one refines a formal model, the corresponding more concrete model may have new variables and new events; it may also strengthen the *guards* of more abstract events. As already mentioned, some proof obligations are generated in order to prove that a refinement is correct. Notice that, if some proof obligations remain unproved, it means that either the formal model is not correctly refined or that an interactive proving session is required. The prover allows us to get a complete proof of the development and hence of the final system. No assumption is made about the *size* of the system, for instance, the number of nodes in a network, where the problem is to elect a leader [ABR 03e]. This contrasts with what should be done while using model-checking techniques.

Organization of the text. The text introduces in a very progressive way the different notations and concepts required for developing the case study. Section 2 analyzes the case study and extracts informations for constructing a first skeleton of B event-based model. The B event-based modeling technique is introduced in section 3 by writing an event B model. The first invoice case study model is given in section 4 and it completes the skeleton of the section 2. Section 5 defines the refinement of a event B model and it is used in the section 6 for deriving the second case study model; a refinement of this model is proposed and introduces an ordering over invoices. Sections 7 and 8 conclude our proof-based development of B event-based models for the case study. The complete B models are given in three figures.

1.2 Analyzing the text of the case study

The starting point of the incremental development of a event B model is the analysis of the requirements to extract pertinent details; the requirements are generally not very well structured and it may be helpful to structure them and then to derive logical and mathematical structures of the problem: sets, constants and properties over sets and constants. We produce the mathematical landscape through requirements elicitation.

B guidelines: *The concept of set is a central one in the B methodology; each basic object is a set; relations and functions should be considered primarily as sets.*

The lines of the case study are numbered; numbers will be used when we will analyze requirements. We will interleave questions asked by either the customer, the specifier, ... and we will answer to these questions.

1. The subject of the case study is **to invoice order**.

Question 1: What is an order? How can we model an order? What does mean to invoice?

Answer: In fact, an order is a member of a set, namely the set of orders. We define a set of orders by the name ALL_ORDERS; we do not know yet, if it is a quantity which may be modified. It is the set of all possible orders. The subject is explained later in the text.

2. To invoice is to change the state of an order (to change it from the state *pending* to *invoiced*).

   Question 2: Can you define what it means `to invoice order`?

   Answer: `To invoice order` is an action or an event which models a modification of the status of an order. The status of an order is either `invoiced` or `pending`; the action should modify the status from pending to invoiced. The action or the event is called *invoice_order* and is triggered for each pending order. The full condition is defined later in the item 5. However, let us detail the status of an order. An order is either pending or invoiced and the action invoice allows us to modify the state from pending to invoiced. It is then clear that we should be able to express the state of orders in our model and the state may change. We can use a set $STATUS$ with two elements `invoiced` and `pending`; the variable *orders_state* can be a function from the set of orders called ALL_ORDERS to the set $STATUS$ ($orders\_state \in \mathrm{ALL\_ORDERS} \longrightarrow STATUS$); *orders_state* is a function because an order has at most only one possible status and it is a total function, because an order has at least one status. In fact, we can use a set called *invoiced_orders* containing the invoiced orders and which is a subset of the set of orders ALL_ORDERS.

   Question 3: Since the possible status of an order is either invoiced or pending, it means that it is a boolean structure and your state is in fact a predicate. Is it true?

   Answer: You point out a very interesting feature of a set-theoretical model; since the possible status of an order is either invoiced or pending, it means that we can use a state variable called *invoiced_orders* which contains the invoiced orders and the complement of *invoiced_orders* in the set of orders is the set of pending orders. At this point, we do not know if the set of orders can be modified and we leave unspecified the type of this variable.

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

Question 4: What is the structure between the features of an order?

Answer: The structure of an order is not clearly given; in fact, no new information on the orders is available. We have one and only one reference to a product of a certain quantity. This means that you can not have two different informations for the same product on the same order. If you want to order 4 products p and 5 products p, either you need to order 9 products p, or you order 4 products p and 5 products p, but you will have two different orders in the set of orders.

Question 5: But can we have several products on an order?

Answer: The answer is given in item 5: *ordered quantity* and *ordered product*. It seems that there is only one ordered product on an order. *The quantity can be different from other orders* means that the quantity is related to an order and not to a product.

Question 6: What are the consequences for the modeling decisions?

Answer: A set of orders can not be a subset of PRODUCTS $\times \mathbb{N}^*$ ($\mathbb{N}^*$ is the set of non-zero natural numbers) because two orders can have the same product and the same quantity. We can have a sequence of PRODUCTS $\times \mathbb{N}^*$, but it is not a good idea in a first abstraction. The simplest way to define the set of orders is the following one: we suppose that ALL_ORDERS is the abstract set which contains all orders (invoiced, pending and future) and orders is the set of existing orders.

We have the following safety property:

$$orders \subseteq \text{ALL\_ORDERS}.$$

Access operations are defined through the following functions:

$$reference \in orders \longrightarrow \text{PRODUCTS}$$

where $reference$ assigns a product to each order and is a function, because an order is related to one and only one ordered product.

$$quantity \in orders \longrightarrow \mathbb{N}^*$$

where $quantity$ assigns a quantity to each ordered product and we assume that if a product is ordered, the quantity is at least 1.

Another possible choice is to combine the two previous functions into a single one, as follows:

$$reference\_quantity \in orders \longrightarrow \text{PRODUCTS} \times \mathbb{N}^*$$

$reference\_quantity$ is a function for defining the set of pairs (product, quantity) of the current orders.

4. The same reference can be ordered on several different orders.

Question 7: So, there may be different orders with the same reference which is ordered?

Answer: Yes, you can order 4 bottles of wine and you (or another one) can order yet 4 bottles of wine so there are two different orders with the same reference (bottle) and the same quantity (4).

5. The state of the order will be changed into *invoiced*, if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.

   Question 8: What is the stock for? How do you use the stock feature in your model?

   Answer: When you invoice an order, you should check that there is enough quantity in stock. The text provides us the guard of the *invoice_order* event and the expression of the guard requires us to model the stock. The *stock* variable is a state variable, because the stock will evolve according to the occurrences of the *invoice_order* event and it assigns to each product the current quantity of available products in the stock:

   $$stock \in \mathrm{PRODUCTS} \longrightarrow \mathbb{N}$$

   Another possible choice is to define *stock* as a partial function but the *invoice_order* event is more complex to write, since we should first check the definability of the function.

6. You have to consider the two following cases:

   (a) **Case 1**

   All the ordered references are references in stock. The stock or the set of the orders may vary:

   - due to the entry of new orders or canceled orders;
   - due to having a new entry of quantities of products in stock at the warehouse.

   We do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.

   Question 9: How do you take this point into account?

   Answer: We state that new events are maintaining the current invariant over variables and we do not care of the way the events are modifying the variables. We keep the invariant.

   (b) **Case 2**

   You do have to take into account the entries of :

   - new orders;

- cancellations of orders;
- entries of quantities in the stock.

Question 10: Is there any relation among the two cases?

Answer: All ordered references are references in stock. Item 5 already states this fact. In fact, we want to model the first case study model, Case 1 and then derive by refinement the second case study model Case 2. We will explain this process later. Perhaps the customer says that some order can arrive with an unreferenced product. It is not really difficult to handle, since such orders can be filtered in the next refinement.

Decision:
The mathematical structure is the set of all possible orders denoted ALL_ORDERS and the state variables of the system are $orders$, $stock$, $invoiced\_orders$, $reference$, $quantity$; the first case study model Case 1 explicitly states that *The stock or the set of the orders may vary* and we can now confirm the state variables, They satisfies the following properties:

- ALL_ORDERS $\neq \emptyset$: the set of all possible orders is not empty.
- $orders \subseteq$ ALL_ORDERS: the set of current existing orders is a subset of the set of all possible orders.
- $invoiced\_orders \subseteq orders$: The set of invoiced orders is a subset of the existing orders.
- $pending\_orders \subseteq orders$: The set of pending orders is a subset of the existing orders.
- $invoiced\_orders \cup pending\_orders = orders$ and $invoiced\_orders \cap pending\_orders = \emptyset$ are two safety properties linking the three variables.
- $reference \in orders \longrightarrow$ PRODUCTS.
- $quantity \in orders \longrightarrow \mathbb{N}^*$.
- $stock \in$ PRODUCTS $\longrightarrow \mathbb{N}$.

Question 11: What are the possible modifications over variables?

Answer: The text has already defined the *invoice_order* event; the item (a) defines two new events: a first event (*new_orders*) adds new orders and a second one (*cancel_orders*) cancels orders. Moreover, the stock may vary and new quantities of products may be added to the stock: the *delivery_to_stock* event.

Question 12: The pending orders disappear from your decisions?

Answer: No, in fact the set of current pending orders is defined by $orders - invoiced\_orders$ and we will understand later why we do not use the variable *pending_orders*.

The first event B model, namely Case 1, is sketched in the next following lines; the model is not yet completed and the events should be defined.

MODEL
  Case 1
SETS
  ALL_ORDERS; PRODUCTS
CONSTANTS
  ...
  PROPERTIES
  $\mathrm{ALL\_ORDERS} \neq \emptyset$
VARIABLES
  $orders, stock, invoiced\_orders, reference, quantity$
INVARIANT
  $orders \subseteq \mathrm{ALL\_ORDERS} \ \wedge$
  $stock \in \mathrm{PRODUCTS} \longrightarrow \mathbb{N} \ \wedge$
  $invoiced \subseteq orders \ \wedge$
  $quantity \in orders \longrightarrow \mathbb{N}^* \ \wedge$
  $reference \in orders \longrightarrow \mathrm{PRODUCTS}$
ASSERTIONS
  ...
INITIALIZATION
  $stock := \mathrm{PRODUCTS} \times \{0\} \ \|$
  $invoiced\_orders, orders, quantity, reference := \emptyset, \emptyset, \emptyset, \emptyset$
EVENTS
  $invoice\_order = \ldots$
  $cancel\_orders = \ldots$
  $new\_orders = \ldots$
  $delivery\_to\_stock = \ldots$
END

An event B model encapsulates variables defining the state of the system; the state should conform to the invariant and each event can be triggered when the current state satisfies the invariant. An abstract model has a name m; the clause SETS contains definitions of sets; the clause CONSTANTS allows one to introduce information related to the mathematical structure of the problem to solve and the clause PROPERTIES contains the effective definitions of constants: it is very important to list carefully properties of constants in a way that can be easily used by the tool Click'n'Prove [ABR 03c].

The second part of the model defines dynamic aspects of state variables and properties over variables using the *invariant* generally called *inductive invariant* and using *assertions* generally called *safety properties*. The invariant $I(x)$ types the variable x, which is assumed to be initialized with respect to the initial conditions, namely $Init(x)$, and which is supposed to be preserved by events (or transitions) enumerated in the EVENTS clause. Conditions of verification called *proof obligations* are generated from the text of the model using the SETS, CONSTANTS and PROPERTIES clauses for defining the mathematical theory and the INVARIANT, INITIALISATION and INVARIANT clauses to generate proof obligations for the preservation (when triggering events) of the invariant and proof obligations stating the correctness of safety properties with respect to the invariant.

B guidelines: *The requirements should be re-structured; basic sets should be identified.*

1.3 Event-based modeling

The B event-driven approach [ABR 03a] is based on the B notation [ABR 96]. It extends the methodological scope of basic concepts such as set-theoretical notations and generalized substitutions in order to take into account the idea of *formal models*. Roughly speaking, a B event-based formal model is characterized by a (finite) list x of *state variables* possibly modified by a (finite) list of *events*; an invariant $I(x)$ states some properties that must always be satisfied by the variables x and *maintained* by the activation of the events. The reader should be very careful and should not consider that the event B method and the classic B method are identical; they share foundational notions like generalized substitutions, refinement, invariance, proof obligations, but a B event-based model intends to provide a formal view of a reactive system, whereas an abstract machine provides operations which can be called and which also maintain the invariant.

In what follows, we briefly recall definitions and principles of formal models and explain how they can be managed with the help of the tool Click'n'Prove [CLE 04, ABR 05a].

Generalized substitutions provide a way to express the transformations of the values of the state variables of a formal model. In its simple form, $x := E(x)$, a generalized substitution looks like an assignment statement. In this construct, x denotes a vector build on the set of state variables of the model, and $E(x)$ a vector of expressions of the same size as the vector x. We interpret it as a *logical simultaneous substitution* of each variable of the vector x by the corresponding expression of the vector $E(x)$. There exists a more general form of generalized substitution. It is denoted by the construct $x :| P(x_0, x)$. This is to be read: "x is modified in such a way that the predicate $P(x_0, x)$ holds", where x denotes the *new value* of the vector, whereas x_0 denotes its *old value*. It is clearly non-deterministic in general. This general form could be considered as a *normal form*, since the simplest form $x := E(x)$ is equivalent to the more general form $x\ :|\ (x = E(x_0))$. In the next table, we give the correspondence of generalized substitutions with the normal form.

Generalized Substitution	Normalization
$x\ :|\ P(x_0,x)$	$x\ :|\ P(x_0,x)$
$x := E(x,y)$	$x\ :|\ (x = E(x_0,y))$
$x :\in A(x,y)$	$x\ :|\ (x \in A(x_0,y)$
$x_1 := E_1(x_1,x_2,y)\ \Vert$ $x_2 := E_2(x_1,x_2,y)$	$(x_1,x_2)\ :|\ \begin{pmatrix} x_1 = E_1((x_1)_0,(x_2)_0,y)\ \wedge \\ x_2 = E_2((x_1)_0,(x_2)_0,y) \end{pmatrix}$

An event is essentially made of two parts: a *guard*, which is a predicate built on the state variables, and an *action*, which is a generalized substitution. An event can take one of the forms shown in the table below. In these constructs, *evt* is an identifier: this is the event name. The first event is not guarded: it is thus always enabled. The guard of the other events, which states the necessary condition for these events to occur, is represented by $G(x)$ in the second case, and by $\exists t \cdot G(t,x)$ in the third case. The latter defines a non-deterministic event where t represents a vector of distinct local variables. The so-called before-after predicate $BA(x,x')$ associated with each event shape, describes the event as a logical predicate expressing the relationship linking the values of the state variables just before (x) and just after (x') the event "execution".

Event	Before-after Predicate $BA(x,x')$
BEGIN $x\ :|\ P(x_0,x)$ END	$P(x,x')$
WHEN $G(x)$ THEN $x\ :|\ Q(x_0,x)$ END	$G(x)\ \wedge\ Q(x,x')$
ANY t WHERE $G(t,x)$ THEN $x\ :|\ R(x_0,x,t)$ END	$\exists t \cdot (\, G(t,x)\ \wedge\ R(x,x',t)\,)$

Proof obligations are produced from events in order to state that the invariant condition $I(x)$ is preserved. We next give general rules to be proved. The first one is the initialization rule which states that the invariant holds for each initial state:

$$\boxed{Init(x) \;\Rightarrow\; I(x)}$$

It follows immediately from the very definition of $BA(x,x')$, the before-after predicate, of each event:

$$\boxed{I(x) \,\wedge\, BA(x,x') \;\Rightarrow\; I(x')}$$

Notice that it follows from the two guarded forms of events that this obligation is trivially discharged when the guard of the event is false. When it is the case, the event is said to be "disabled". An event is essentially a reactive entity and reacts with respect to its guard $\mathsf{grd}(e)(x)$. An event should be *feasible* and the feasibility is related to the feasibility of the generalized substitution of the event: some next state must be reachable from a given state. Since events are reactive, related proof obligations should guarantee that the current state satisfying the invariant should be feasible. In the next table, we define, for each possible event, the feasibility condition.

Event : E	Feasibility : $fis(E)$
$x :\mid Init(x)$	$\exists x \cdot Init(x)$
BEGIN $x :\mid P(x_0,x)$ END	$I(x) \;\Rightarrow\; \exists x' \cdot P(x,x')$
WHEN $G(x)$ THEN $x :\mid P(x_0,x)$ END	$I(x) \;\wedge\; G(x) \;\Rightarrow\; \exists x' \cdot P(x,x')$
ANY l WHERE $G(l,x)$ THEN $x :\mid P(x_0,x,l)$ END	$I(x) \;\wedge\; G(l,x) \;\Rightarrow\; \exists x' \cdot P(x,x',l)$

For instance, the event BEGIN $x :\mid P(x_0,x)$ END is feasible, when the invariant ensures the existence of a next value x satisfying $P(x_0,x)$ (x_0 is the value of x, when the event is observed and x will be the value afterward). If we consider the following event BEGIN $a,b,c :\mid a = a_0 \wedge b = b_0 \wedge a_0, b_0, c_0 \in \mathbb{N} \wedge c = a \; div \; b$ END, the invariant should include a condition of the state of b ($b \neq 0$). Finally, predicates in the ASSERTIONS clause should be implied by the predicates of the INVARIANT clause; the condition is simply formalized as follows:

$$\boxed{I(x) \Rightarrow A(x)}$$

Now, we have defined the main concepts for deriving a B event-based model for the first case study.

1.4 Modeling the first event B model Case 1

The construction of an event B model is based on an analysis of data which are manipulated; each B model is organized according to clauses and requirements of the case study are incrementally added into the B model. In section 2, we have analyzed the requirements and we have derived a first sketch of a event B model. Events should now be completed and the model should be internally validated. The internal validation checks that proof obligations hold and is made with the help of the tool Click'n'Prove [CLE 04].

In the text of the description of the system, we use the following informations: *all the ordered references are references in stock* and we derive that the invoice_order event is triggered when there are enough items of a given reference in the current stock. Let o be a pending order ($o \in orders - invoiced\_orders$). If the quantity in stock of the product whose reference is $reference(o)$ is greater than the ordered one ($quantity(o) \leq stock(reference(o))$), then the order is invoiced, using:

$$invoiced\_orders := invoiced\_orders \cup \{o\}$$

and the stock is updated:

$$stock(reference(o)) := stock(reference(o)) - quantity(o) \ .$$

```
invoice_order =
  ANY
    o
  WHERE
      o ∈ orders − invoiced_orders    ∧
      quantity(o) ≤ stock(reference(o))
  THEN
      invoiced_orders := invoiced_orders ∪ {o}||
      stock(reference(o)) := stock(reference(o)) − quantity(o)
  END
```

The next three events are modeling the state changes for the variables attached to the stock and to the orders: *the stock or the set of the orders may vary*.

Question 13: How are variables modified? Can we cancel an invoiced order?

Answer: First of all, the text expresses that the stock and the set of orders may vary; either the variable *invoiced_orders* is not modified, since the invoiced orders are processed orders, or we can cancel invoiced orders, since it is possible action over orders. We can only modify the set $orders - invoiced\_orders$.

The modifications are to add a new order in the current set of orders and to set the order into the pending set, or to cancel a pending order from the set orders, or to modify the stock variable by incrementing the quantity of a product.

Question 14: Are your changes the most general ones?

Answer: I do not not understand *the most general notion.*

Question 15: Is it the most abstract model for the three events?

Answer: The specification text tells us that variables are modified and they are less precise than what we suggest. So we propose to require that the three events modify variables while the invariant is preserved, but the variable *invoiced_orders* is not modified by these events.

The event cancel_orders and the event new_orders modify the variables $orders, quantity, reference$ and the next values of these variables should satisfy:

$$\left(\begin{array}{l} orders \subseteq \mathrm{ALL\_ORDERS} \ \wedge \\ invoiced\_orders \subseteq orders \ \wedge \\ quantity \in orders \longrightarrow \mathbb{N}^* \ \wedge \\ reference \in orders \longrightarrow \mathrm{PRODUCTS} \end{array}\right).$$

We do not give details on the possible modifications and we do not care following the first case.

Question 16: Why are you defining those events which have no effect on the variables?

Answer: These events are hidden in the first case but they are explicitly mentioned. They will be refined in the second case because the second case provides more details on those events. Finally, they illustrate the *keep* concept [ABR 05c], which expresses a possible change with respect to the invariant and which simplifies the refinement.

cancel_orders $=$
BEGIN

$$\left(\begin{array}{c} orders, \\ quantity, \\ reference \end{array}\right) :| \left(\begin{array}{l} orders \subseteq \mathrm{ALL\_ORDERS} \ \wedge \\ invoiced\_orders \subseteq orders \ \wedge \\ quantity \in orders \longrightarrow \mathbb{N}^* \ \wedge \\ reference \in orders \longrightarrow \mathrm{PRODUCTS} \end{array}\right)$$

END

$$
\begin{array}{l}
\mathsf{new\_orders} = \\
\text{BEGIN} \\
\quad \begin{pmatrix} orders, \\ quantity, \\ reference \end{pmatrix} :| \begin{pmatrix} orders \subseteq \text{ALL\_ORDERS} \ \wedge \\ invoiced\_orders \subseteq orders \ \wedge \\ quantity \in orders \longrightarrow \mathbb{N}^* \ \wedge \\ reference \in orders \longrightarrow \text{PRODUCTS} \end{pmatrix} \\
\text{END}
\end{array}
$$

The event delivery_to_stock change the value of *stock* and does not change other variable. We do not know how the stock is modified and we express that a modification is possible.

$$
\begin{array}{l}
\mathsf{delivery\_to\_stock} = \\
\quad \text{BEGIN} \\
\quad\quad stock :| (stock \in \text{PRODUCTS} \longrightarrow \mathbb{N}) \\
\quad \text{END}
\end{array}
$$

Question 17: The discourse of the event method reports on checking internal consistency. Did you check the internal consistency? Is the Case 1 model internally consistent?

Answer: The checking of internal consistency is established by proving nine proof obligations, stating that the invariant is initially true and that each event is maintaining the invariant. Each proof obligation is automatically discharged by the tool Click'n'Prove.

The client may be interested by an animation and one can use an animator for testing the possible behaviors of the global model.

Question 18: How do you express that only these events can modify variables of the model?

Answer: The set of variables of the model is the frame of the model; no other variable can be modified; if a variable is not explicitly modified, it is not changed. We assume that the model is closed.

1.5 Model refinement

The refinement of a formal model allows us to enrich a model using a *step by step* approach. Refinement provides a way to construct stronger invariants and also to add details to a model. It is also used to transform an abstract model into a more concrete version by modifying the state description. This is essentially done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event into a corresponding concrete version, and by adding new events. The abstract state variables, x, and the concrete ones, y, are linked together by means of a so-called *gluing invariant* $J(x,y)$. A number of proof obligations ensure that (1) each abstract event is correctly refined by its corresponding concrete version, (2) each new event refines *skip*, (3) no new event takes control forever, and (4) relative deadlock-freeness is preserved (the relative deadlock-freeness states that the concrete model is not more blocked than the abstract one!).

We suppose that an Abstract Model AM with variables x and invariant $I(x)$ is refined by a Concrete Model CM with variables y and gluing invariant $J(x,y)$. The first proof obligation states the initial concrete states implies that there is at least one initial abstract state satisfying the abstract initial condition and related to the initial concrete state by the gluing invariant:

$$\boxed{INIT(y) \;\Rightarrow\; \exists x.(Init(x) \;\wedge\; J(x,y))}$$

If $BAA(x,x')$ (standing for Before-After Abstract event) and $BAC(y,y')$ (standing for Before-After Concrete event) are respectively the abstract and concrete before-after predicates of the same event, we have to prove the following statement:

$$\boxed{I(x) \;\wedge\; J(x,y) \;\wedge\; BAC(y,y') \;\Rightarrow\; \exists x' \cdot (BAA(x,x') \;\wedge\; J(x',y'))}$$

This says that under the abstract invariant $I(x)$ and the concrete one $J(x,y)$, a concrete step $BAC(y,y')$ can be simulated ($\exists x'$) by an abstract one $BAA(x,x')$ in such a way that the gluing invariant $J(x',y')$ is preserved. A new event with before-after predicate $BA(y,y')$ must refine *skip* ($x' = x$). This leads to the following statement to prove:

$$\boxed{I(x) \;\wedge\; J(x,y) \;\wedge\; BA(y,y') \;\Rightarrow\; J(x,y')}$$

Moreover, we must prove that a variant $V(y)$ is decreased by each new event (this is to guarantee that an abstract step may occur). We have thus to prove the following for each new event with before-after predicate $BA(y,y')$:

$$\boxed{I(x) \;\wedge\; J(x,y) \;\wedge\; BA(y,y') \;\Rightarrow\; V(y') < V(y)}$$

Finally, we must prove that the concrete model does not introduce more deadlocks than the abstract one. This is formalized by means of the following proof obligation:

$$\boxed{I(x) \;\wedge\; J(x,y) \;\wedge\; \mathsf{grds}(AM) \;\Rightarrow\; \mathsf{grds}(CM)}$$

where $\mathsf{grds}(AM)$ stands for the disjunction of the guards of the events of the abstract model, and $\mathsf{grds}(CM)$ stands for the disjunction of the guards of the events of the concrete one.

1.6 Modeling the second event B model Case 2 by refinement of Case 1

According to the text of the specification, the second case study model *takes into account the entries of* :

- *new orders*;
- *cancellations of orders*;
- *entries of quantities in the stock*.

Question 19: Why do you choose that title for that section?

Answer: The behavior of Case 2 is more specialized than Case 1; in Case 1 we do not express how the variables are modified. We state that variables are modified by maintaining the invariant and it clear that Case 2 is more deterministic than case 1:

- *orders* may change by adding new orders;
- *orders* may change by removing pending orders from the current *orders*;
- *stock* changes by adding new quantities of products in the stock.

No new event is added.

Decision:
The three last events of Case 1 should be refined to handle the modifications of the variables *orders*, *quantity*, *reference* and *stock* according to the three last items:

- *new orders*: the new_orders event modifies *orders*, *quantity*, *reference*; it adds a new order called o which is not yet existing in the current set of orders called *orders*; *quantity* and *reference* are updated according to the ordered quantity q and reference p;
- *cancellations of orders*: the cancel_orders event modifies *orders*, *quantity*, *reference*; it removes a order called o which is pending in the current set of orders called *orders*; *quantity* and *reference* are updated;
- *entries of quantities in the stock*: the delivery_to_stock event adds a given quantity q of a given product p in the stock.

The text for Case 2 is very clear and it mentions specific ways to modify variables *orders*, *quantity*, *reference* and *stock*. Events will simply translate these expressions.

$$
\begin{array}{l}
\textsf{new\_orders} = \\
\quad \text{ANY} \\
\qquad o, q, p \\
\quad \text{WHERE} \\
\qquad o \in \text{ALL\_ORDERS} - orders \\
\qquad q \in \mathbb{N}^* \\
\qquad p \in \text{PRODUCTS} \\
\quad \text{THEN} \\
\qquad orders := orders \cup \{o\} \,||\, \\
\qquad quantity(o) := q \,||\, \\
\qquad reference(o) := p \\
\quad \text{END}
\end{array}
$$

Let o be an order which is not yet either pending or invoiced. It is a future order which is added to the current set of orders (*orders*) and the quantity of product is set to q; the identification of the product of the o order is set to p.

$$
\begin{array}{l}
\textsf{cancel\_orders} = \\
\quad \text{ANY} \\
\qquad o \\
\quad \text{WHERE} \\
\qquad o \in orders - invoiced\_orders \\
\quad \text{THEN} \\
\qquad orders := orders - \{o\} \,||\, \\
\qquad quantity := \{o\} \lhd\!\!\!- \, quantity \,||\, \\
\qquad reference := \{o\} \lhd\!\!\!- \, reference \\
\quad \text{END}
\end{array}
$$

Let o be an order which is pending. The event deletes the order from the set *orders* and the two functions *quantity* and *reference* are updated by removing o from the set of orders which is the domain of those functions.

Question 20: What happens if we forget the condition over invoiced orders in the guard of the event cancel_orders?

Answer: The refinement conditions generate a proof obligation like $o \in orders \Rightarrow invoiced\_orders \subseteq orders - \{o\}$ and it is clearly not provable without the guard $o \notin invoiced\_orders$.

delivery_to_stock = ANY p, q WHERE $q \in \mathbb{N}^*$ $p \in$ PRODUCTS THEN $stock(p) := stock(p) + q$ END	The stock can only be increased and the event increases by q units the quantity of the product p. The stock for p is increased by q.

Question 21: Is the concrete event delivery_to_stock more deterministic than the abstract one?

Answer: Yes, the concrete event only modifies the quantity of one product. The abstract event can also decrease quantities of products.

In the case study, customers mention the following statement:

But, we do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in a up-to-date state.

The last question leads to a new case study, called Case 3; it takes into account the flow of orders. The new model captures the notion of flow by a set; it means that the ordering of arrival is not expressed, for instance. We can require some fairness assumption over some events to obtain a deadlock and live-lock-free model. It is clear that we can not state any kind of fairness in B and the reason is that the B language does not provide this facility; Méry [MER 99] analyzes the extension of B scope with respect to liveness and fairness properties. However, the key question is to refine models while fairness constraints are stated and Cansell et al [CAN 00b] propose predicate diagrams to deal with these questions. In fact, it is possible that an order remains always pending and is never invoiced, because there are always other orders which are processed. Another problem is that the quantity may be not sufficient for a while and it is infinitely often sufficient for a given quantity of a given product.

If the referenced quantity changes in the stock (event delivery_to_stock), one can also invoice another order with the same referenced product. Modeling this fact in an abstract way requires strong fairness on event delivery_to_stock. A first idea is to use a sequence of orders and to invoice the first suitable order in the sequence. In this case we have no starvation if the event delivery_to_stock is fair enough. For the customer, it is not a good solution because the delay for delivery to the stock is too long and so one can invoice other orders. We decide to add a time to each order to sort orders ($time \in orders \rightarrowtail \mathbb{N}$) and to invoice the most recent possible order. The event new_orders gives to each new order its time using a variable t ($t \in \mathbb{N}$) which always contains the next ordered time ($\forall i \cdot (i \in \mathbf{ran}(time) \Rightarrow i \leq t)$).

The variable $time$ records the time when the order was added and the new condition strengthens the guard of the previous event invoice_order:

$$\forall d \cdot \left(\begin{array}{l} \quad d \in orders - invoiced\_orders \ \wedge \\ \quad quantity(d) \leq stock(reference(d)) \\ \Rightarrow \\ \quad time(o) \leq time(d) \end{array} \right)$$

The variable $time$ is an total injection from $orders$ into $\mathbb{N}$, which is defining a total ordering over $orders$.

invoice_order $=$
ANY
o
WHERE
$o \in orders - invoiced\_orders \quad \wedge$
$quantity(o) \leq stock(reference(o)) \ \wedge$
$\forall d \cdot \left(\begin{array}{l} \quad d \in orders - invoiced\_orders \ \wedge \\ \quad quantity(d) \leq stock(reference(d)) \\ \Rightarrow \\ \quad time(o) \leq time(d) \end{array} \right)$
THEN
$invoiced\_orders := invoiced\_orders \cup \{o\} ||$
$stock(reference(o)) := stock(reference(o)) - quantity(o)$
END

new_orders $=$
ANY
o, q, p
WHERE
$o \in \mathrm{ALL\_ORDERS} - orders$
$q \in \mathbb{N}^*$
$p \in \mathrm{PRODUCTS}$
THEN
$orders := orders \cup \{o\} ||$
$quantity(o) := q ||$
$reference(o) := p ||$
$time(o) := t ||$
$t := t + 1$
END

The variable t is a new shared variable which models the evolution of the timestamps; we use the same variable to be sure that we obtain a total ordering over orders. $time$ is updated according to the current value of the variable t.

```
cancel_orders =
  ANY
    o
  WHERE
    o ∈ orders − invoiced_orders
  THEN
    orders := orders − {o}||
    quantity := {o} ⩤ quantity||
    reference := {o} ⩤ reference||
    time := {o} ⩤ time
  END
```

When one cancels an order, *time* should be updated by removing the canceled order from the domain of *time*.

1.7 The natural language description of the event b models

The new description appears in our development including B models; the initial text is quite clear. The refinement-based development (starting from a very abstract model) helps us to gradually improve the understanding of the case studies. The first reading attempts to detect informations from the informal text itself and leads to the first abstract model. The new natural language description is the old one enriched by new informations derived from the question/answer game. We add the following text to the initial one:

We have one and only one reference to a product of a certain quantity per order. This means that you can not have two different informations for the same product on the same order. If you want to order 4 products p and 5 products p, either you need to order 9 products p, or you order 4 products p and 5 products p, but you will have two different orders in the set of orders. Each invoiced order can not be canceled according to the customer of the specification. The second case is simply a refinement of the first one and it gives a more precise view of the environment actions, namely stock variations or orders creation/cancelation.

The implementation of a fairness assumption was obtained by a time-stamp over orders.

1.8 Conclusion

The case study provides us a framework for introducing the main concepts of the event B method; the statement of the development should include a table with the required proof obligations:

Model	Unproved PO	PO	Interactive PO
Case 1	0	9	0
Case 2	0	14	3
Case 3	0	18	5

Each proof obligation requires less than one interaction step using the Click'n'Prove tool. We emphasize the central role of the model refinement in the construction of formal models; it simplifies proofs by providing a progressive and detailed view of a system through different models.

Bibliography

[ABR 96] Jean-Raymond Abrial. *The B book*. Cambrige University Press, 1996.

[ABR 98] Jean-Raymond Abrial. *On B* In Bert [BER 98], pages 1–8.

[ABR 02] Jean-Raymond Abrial, Dominique Cansell, and Guy Laffitte. *"higher-order" mathematics in B* In Bert et al. [BER 02], pages 370–393.

[ABR 03a] Jean-Raymond Abrial. *$B^{\#}$: Toward a synthesis between Z and B* In Bert et al. [BER 03], pages 168–177.

[ABR 03b] Jean-Raymond Abrial. *Event based sequential program development: Application to constructing a pointer program* In Araki et al. [ARA 03], pages 51–74.

[ABR 03c] Jean-Raymond Abrial and Dominique Cansell. *Click'n'Prove: Interactive proofs within set theory* In Basin and Wolff [BAS 03], pages 1–24.

[ABR 03d] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. *Formal derivation of spanning trees algorithms* In Bert et al. [BER 03], pages 457–476.

[ABR 03e] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. *A mechanically proved and incremental development of IEEE 1394 tree* identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.

[ABR 05a] Jean-Raymond Abrial and Dominique Cansell. *Click'n'Prove* 2004,2005. http://www.loria.fr/cansell.

[ABR 05b] Jean-Raymond Abrial and Dominique Cansell. *Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity) Journal of Universal Computer Science*, 11(5):744–770, May 2005.

[ABR 05c] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. *Refinement and reachability in Event B* In Treharne et al. [TRE 05], pages 222–241.

[ABR 98] Jean-Raymond Abrial and Louis Mussat. *Introducing dynamic constraints in B* In Bert [BER 98], pages 83–128.

[ARA 03] Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, eds. *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*. Springer, 2003.

[BAC 79] Ralph Back. *On correct refinement of programs Journal of Computer and System Sciences*, 23(1):49–68, 1979.

[BAS 03] David A. Basin and Burkhart Wolff, eds. *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rom, Italy, September 8-12, 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*. Springer, 2003.

[BER 98] Didier Bert, editor. *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, volume 1393 of *Lecture Notes in Computer Science*. Springer, 1998.

[BER 02] Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, eds. *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings*, volume 2272 of *Lecture Notes in Computer Science*. Springer, 2002.

[BER 03] Didier Bert, Jonathan P. Bowen, Steve King, and Marina A. Waldén, eds. *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*. Springer, 2003.

[CAN 00a] Dominique Cansell and Dominique Méry. *Abstraction and refinement of features* In Ryan Stephen, Gilmore and Mark, eds, *Language Constructs for Designing Features*. Springer Verlag, 2000.

[CAN 00b] Dominique Cansell, Dominique Méry, and Stephan Merz. *Diagram Refinements for the Design of Reactive Systems Journal of Universal Computer Science*, 7(2):159–174, 2001.

[CAN 02] Dominique Cansell, Ganesh Gopalakrishnan, Michael D. Jones, Dominique Méry, and Airy Weinzoepflen. *Incremental proof of the producer/consumer property for the PCI* protocol. In Bert et al. [BER 02], pages 22–41.

[CHA 88] K. Mani Chandy and Jay Misra. *Parallel Program Design A Foundation* Addison-Wesley, 1988. ISBN 0-201-05866-9.

[CLE 04] ClearSy, Aix-en-Provence (F). *B4FREE* 2004. http://www.b4free.com.

[DIJ 76] Edgster W. Dijkstra. *A Discipline of Programming* Prentice-Hall, 1976.

[EHR 85] Hart Ehrig and Bernt Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics* EATCS Monographs on Theoretical Computer Science. Springer Verlag, W. Brauer and R. Rozenberg and A. Salomaa eds, 1985.

[LAM 94] Leslie Lamport. *A temporal logic of actions ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LAM 02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[MER 99] Dominique Méry. *Requirements for a temporal B: assigning temporal meaning to abstract machines ... and to abstract systems*. In A. Galloway and K. Taguchi, eds, *IFM'99 Integrated Formal Methods 1999*, YORK, June 1999.

[MOR 05] C. Morgan, T.S. Hoang, and Jean-Raymond Abrial. *The challenge of probabilistic event B - extended abstract* In Treharne et al. [TRE 05], pages 162–171.

[TRE 05] Helen Treharne, Steve King, Martin C. Henson, and Steve Schneider, eds. *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*. Springer, 2005.