The invoice case study modelling in Event B
Dominique Cansell, Dominique Méry

To cite this version:


HAL Id: inria-00000857
https://inria.hal.science/inria-00000857

Submitted on 26 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
The invoice case study modelling in Event B

Dominique Cansell  
Université de Metz & LORIA  
cansell@loria.fr

Dominique Méry  
Université Henri Poincaré Nancy 1 & LORIA  
mery@loria.fr

November 26, 2005
Contents
1 Introduction 3
2 Analysing the text of the case study 4
3 Event-based modelling 10
4 Modelling the first event B model Case 1 13
5 Model Refinement 15
6 Modelling the second event B model Case 2 by refinement of Case 1 16
7 The Natural Language Description of the event B models 21
8 Conclusion 21
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 03a,
ABR 98] 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 de-
veloping models from a textually-defined system, while preserving correctness; it im-
plements the proof-based development paradigm. Each development includes proofs
for invariance and refinement. Operations of the B classical 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. An event is not called but
observed. When refining machines in classic B, one should maintain the number of op-
erations 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, a B
event-based 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 classical chapter introduces useful notations for the
event B method like set theory, generalised 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 relation-
ship 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 obliga-
tions, 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 in-
variant properties; they also include aspects related to the termination. Such models are
thus very close to action systems introduced by Back [BAC 79] (see the chapter of Sin-
clair), 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.

**Organisation of the text**  It introduces in a very progressive way the different notations and concepts required for developing the case study. Section 2 analyses the case study and extracts informations for constructing a first skeleton of B event-based model. The B event-based modelling 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.

## 2 Analysing 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 analyse 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 means 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. But, 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* ∈ *ALL_ORDERS* → *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 to other orders* means that the quantity is related to an order and not to a product.
Question 6: What are the consequences for the modelling decisions?

Answer: A set of orders can not be a subset of \( \text{PRODUCTS} \times \mathbb{N}^* \) (\( \mathbb{N}^* \) 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 \( \text{PRODUCTS} \times \mathbb{N}^* \) 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 \text{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:

\[ \text{orders} \subseteq \text{ALL_ORDERS}. \]

Access operations are defined through the following functions:

\[ \text{reference} \in \text{orders} \rightarrow \text{PRODUCTS} \]

where \text{reference} assigns a product to each order and is a function, because an order is related to one and only one ordered product.

\[ \text{quantity} \in \text{orders} \rightarrow \mathbb{N}^* \]

where \text{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:

\[ \text{reference_quantity} \in \text{orders} \rightarrow \text{PRODUCTS} \times \mathbb{N}^* \]

\( \text{reference_quantity} \) is a function for defining the set of (product, quantity) pairs 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 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:

\[ stock \in \text{PRODUCTS} \rightarrow \mathbb{N} \]

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

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 an up-to-date state.

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

**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 ≠ ∅: the set of all possible orders is not empty.
- orders ⊆ ALL_ORDERS: the set of current existing orders is a subset of the set of all possible orders.
- invoiced_orders ⊆ orders: The set of invoiced orders is a subset of the existing orders.
- pending_orders ⊆ orders: The set of pending orders is a subset of the existing orders.
- invoiced_orders ∪ pending_orders = orders and invoiced_orders ∩ pending_orders = ∅ are two safety properties linking the three variables.
- reference ∈ orders → PRODUCTS
- quantity ∈ orders → N^*
- stock ∈ PRODUCTS → 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 Case1, is sketched in the next following lines; the model is not yet completed and the events should be defined.
MODEL
Case1
SETS
ALL_ORDERS: PRODUCTS
CONSTANTS
...
PROPERTIES
ALL_ORDERS ≠ ∅
VARIABLES
orders, stock, invoiced_orders, reference, quantity
INVARIANT
orders ⊆ ALL_ORDERS ∧
stock ∈ PRODUCTS → N ∧
invoiced ⊆ orders ∧
quantity ∈ orders → N* ∧
reference ∈ orders → PRODUCTS
ASSERTIONS
...
INITIALISATION
stock := PRODUCTS × {0} ||
invoiced_orders, orders, quantity, reference := ∅, ∅, ∅, ∅
EVENTS
invoice_order = ...
cancel_orders = ...
new_orders = ...
delivery_to_stock = ...
END

A 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 initialised 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 re-
B guidelines: The requirements should be re-structured; basic sets should be identified

3 Event-based modelling

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 generalised substitutions in order to take into account the idea of formal models. Roughly speaking, a B event-based formal model is characterised 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 to consider that the B event-based method and the B classical method are identical; they share foundational notions like generalised 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 and Prove [CLE 04, ABR 05a].

Generalised 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 generalised 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 generalised 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 generalised substitutions with the normal form.
Generalised Substitution | Normalisation
---|---
$x : | P(x_0, x)$ | $x : | P(x_0, x)$
$x := E(x, y)$ | $x : | (x = E(x_0, y))$
$x \in A(x, y)$ | $x : | (x \in A(x_0, y)$
$x_1 := E_1(x_1, x_2, y)$ \parallel
$x_2 := E_2(x_1, x_2, y)$ | $(x_1, x_2) : | (x_1 = E_1((x_1)_0, (x_2)_0, y) \land x_2 = E_2((x_1)_0, (x_2)_0, y) )$

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 generalised substitution. An event can take one of the forms shown in the table below. In these constructs, \( \text{evt} \) 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 one. 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.

<table>
<thead>
<tr>
<th>Event</th>
<th>Before-after Predicate ( BA(x, x') )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{BEGIN } x :</td>
<td>P(x_0, x) \text{ END}</td>
</tr>
<tr>
<td>\text{WHEN } G(x) \text{ THEN } x :</td>
<td>Q(x_0, x) \text{ END}</td>
</tr>
<tr>
<td>\text{ANY } t \text{ WHERE } G(t, x) \text{ THEN } x :</td>
<td>R(x_0, x, t) \text{ END}</td>
</tr>
</tbody>
</table>

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 initialisation rule which states that the invariant holds for each initial state:

\[
\text{Init}(x) \Rightarrow I(x)
\]
It follows immediately from the very definition of the before-after predicate, $BA(x, x')$ of each event:

$$I(x) \land BA(x, x') \Rightarrow I(x')$$

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 $grd(e)(x)$. An event should be feasible and the feasibility is related to the feasibility of the generalised 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.

<table>
<thead>
<tr>
<th>Event : $E$</th>
<th>Feasibility : $fis(E)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : \mid Init(x)$</td>
<td>$\exists x \cdot Init(x)$</td>
</tr>
<tr>
<td>$\text{BEGIN } x : \mid P(x_0, x) \text{ END}$</td>
<td>$I(x) \Rightarrow \exists x' \cdot P(x, x')$</td>
</tr>
<tr>
<td>$\text{WHEN } G(x) \text{ THEN}$</td>
<td>$I(x) \land G(x) \Rightarrow \exists x' \cdot P(x, x')$</td>
</tr>
<tr>
<td>$x : \mid P(x_0, x)$</td>
<td>$\text{END}$</td>
</tr>
<tr>
<td>$\text{ANY } l \text{ WHERE } G(l, x) \text{ THEN}$</td>
<td>$I(x) \land G(l, x) \Rightarrow \exists x' \cdot P(x, x', l)$</td>
</tr>
<tr>
<td>$x : \mid P(x_0, x, l)$</td>
<td>$\text{END}$</td>
</tr>
</tbody>
</table>

For instance, the event $\text{BEGIN } x : \mid P(x_0, x) \text{ END}$ 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 afterwards). If we consider the following event $\text{BEGIN } a, b, c : \mid a = a_0 \land b = b_0 \land a_0, b_0, c_0 \in \mathbb{N} \land c = a \text{ div } b \text{ END}$, 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 formalised as follows:

$$I(x) \Rightarrow A(x)$$

Now, we have defined the main concepts for deriving a B event-based model for the first case study.
4 Modelling 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 organised according to clauses and requirements of the case study are incrementally added into the B model. In the section 7, we have analysed 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 \( o \in \text{orders} - \text{invoiced_orders} \). If the quantity in stock of the product whose reference is reference\( (o) \) is greater than the ordered one \( \text{quantity}(o) \leq \text{stock}(\text{reference}(o)) \), then the order is invoiced \( (\text{invoiced_orders} := \text{invoiced_orders} \cup \{o\}) \) and the stock is updated:
\[
\text{stock}(\text{reference}(o)) := \text{stock}(\text{reference}(o)) - \text{quantity}(o).
\]

```plaintext
invoice_order =
    ANY
    o
    WHERE
    o \in \text{orders} - \text{invoiced_orders} \land
    \text{quantity}(o) \leq \text{stock}(\text{reference}(o))
    THEN
    \text{invoiced_orders} := \text{invoiced_orders} \cup \{o\}||
    \text{stock}(\text{reference}(o)) := \text{stock}(\text{reference}(o)) - \text{quantity}(o)
    END
```

The next three events are modelling 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 either 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

\[
\begin{align*}
\left( \begin{array}{l}
orders \subseteq \text{ALL_ORDERS} \\
invoiced_orders \subseteq orders \\
quantity \in orders \rightarrow \mathbb{N}^* \\
reference \in orders \rightarrow \text{PRODUCTS}
\end{array} \right)
\end{align*}
\]

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.

```latex
\begin{align*}
cancel\text{\_orders} & = \\
& \text{BEGIN} \\
& \left( \begin{array}{l}
orders, \\
quantity, \\
reference
\end{array} \right) : | \\
& \left( \begin{array}{l}
orders \subseteq \text{ALL_ORDERS} \\
invoiced_orders \subseteq orders \\
quantity \in orders \rightarrow \mathbb{N}^* \\
reference \in orders \rightarrow \text{PRODUCTS}
\end{array} \right) \\
& \text{END}
\end{align*}
\begin{align*}
\text{new\_orders} & = \\
& \text{BEGIN} \\
& \left( \begin{array}{l}
orders, \\
quantity, \\
reference
\end{array} \right) : | \\
& \left( \begin{array}{l}
orders \subseteq \text{ALL_ORDERS} \\
invoiced_orders \subseteq orders \\
quantity \in orders \rightarrow \mathbb{N}^* \\
reference \in orders \rightarrow \text{PRODUCTS}
\end{array} \right) \\
& \text{END}
\end{align*}
```
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.

\[
\text{delivery_to_stock = }
\begin{align*}
\text{BEGIN} \\
stock & : | (stock \in \text{PRODUCTS} \rightarrow \mathbb{N}) \\
\text{END}
\end{align*}
\]

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.

Figure 1 summarises the final Case 1 model; the client may be interested by an animation and one can use an animator for testing the possible behaviours 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.

5 Model Refinement

The refinement of a formal model allows us to enrich a model in 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 for ever, and (4) relative deadlock-freeness is preserved (the relative deadlock-freeness states that the concrete model is not more blocked than the abstract one!).

15
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:

$$\text{INIT}(y) \Rightarrow \exists x. (\text{Init}(x) \land J(x, y))$$

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:

$$I(x) \land J(x, y) \land BAC(y, y') \Rightarrow \exists x'. (BAA(x, x') \land J(x', y'))$$

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 $\text{skip} (x' = x)$. This leads to the following statement to prove:

$$I(x) \land J(x, y) \land BA(y, y') \Rightarrow J(x, y')$$

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 $BA(y, y')$:

$$I(x) \land J(x, y) \land BA(y, y') \Rightarrow V(y') < V(y)$$

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

$$I(x) \land J(x, y) \land \text{grds}(AM) \Rightarrow \text{grds}(CM)$$

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

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

Answer: The behaviour of the Case 2 is more specialised than the Case 1; in the 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 the Case 2 is more deterministic than the 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.
new_orders =
  ANY
  o, q, p
WHERE
  o ∈ ALL_ORDERS − orders
  q ∈ N*
  p ∈ PRODUCTS
THEN
  orders := orders ∪ {o}||
  quantity(o) := q||
  reference(o) := p
END

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

Let o be an order which is not yet neither pending nor 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.

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 \( o ∈ orders \Rightarrow invoiced\_orders \subseteq orders − \{o\} \) and it is clearly not provable without the guard \( o \notin invoiced\_orders \).
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** (see fig. 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] analyses 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 be 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. Modelling 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 ($\text{time} \in \text{orders} \rightarrow \mathbb{N}$) 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 contains always the next ordered time ($\forall i \cdot (i \in \text{ran}(\text{time}) \Rightarrow i \leq t)$.)
The variable \( \text{time} \) records the time when the order was added and the new condition strengthens the guard of the previous event \( \text{invoice\_order} \):

\[
\forall d \cdot \begin{cases}
    d \in \text{orders} - \text{invoiced\_orders} \land \\
    \text{quantity}(d) \leq \text{stock}(\text{reference}(d)) \\
    \Rightarrow \\
    \text{time}(o) \leq \text{time}(d)
\end{cases}
\]

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

\begin{verbatim}
 invoice_order =
    ANY
    o
    WHERE
    o \in \text{orders} - \text{invoiced\_orders} \land
    \text{quantity}(o) \leq \text{stock}(\text{reference}(o)) \land
    \forall d \cdot \begin{cases}
        d \in \text{orders} - \text{invoiced\_orders} \land \\
        \text{quantity}(d) \leq \text{stock}(\text{reference}(d)) \\
        \Rightarrow \\
        \text{time}(o) \leq \text{time}(d)
    \end{cases}
    \Rightarrow
    \text{invoiced\_orders} := \text{invoiced\_orders} \cup \{o\}||
    \text{stock}(\text{reference}(o)) := \text{stock}(\text{reference}(o)) - \text{quantity}(o)
END

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

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 \( \text{orders} \). \( \text{time} \) 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 cancelled order from the domain of time.

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 cancelled 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/cancellation.

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

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.

---

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 cancelled 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/cancellation.

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

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.
Each proof obligation requires less than one interaction step using the Click and Prove tool. We emphasise 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.

References


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


MODEL
    Case1
SETS
    ALL_ORDERS; PRODUCTS
PROPERTIES
    ALL_ORDERS ≠ ∅
VARIABLES
    orders, stock, invoiced_orders, reference, quantity
INVARIANT
    orders ⊆ ALL_ORDERS ∧
    stock ∈ PRODUCTS → N ∧
    invoiced_orders ⊆ orders ∧
    quantity ∈ orders → N* ∧
    reference ∈ orders → PRODUCTS
INITIALISATION
    stock, invoiced_orders, orders, quantity, reference := PRODUCTS × {0}, ∅, ∅, ∅, ∅
EVENTS
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;
cancel_orders =
BEGIN
    orders, quantity, reference : | (orders ⊆ ALL_ORDERS ∧
    invoiced_orders ⊆ orders ∧
    quantity ∈ orders → N* ∧
    reference ∈ orders → PRODUCTS)
END;
new_orders =
BEGIN
    orders, quantity, reference : | (orders ⊆ ALL_ORDERS ∧
    invoiced_orders ⊆ orders ∧
    quantity ∈ orders → N* ∧
    reference ∈ orders → PRODUCTS)
END;
delivery_to_stock =
BEGIN
    stock : | (stock ∈ PRODUCTS → N)
END
END

Figure 1: Case 1
MODEL
Case2
REFINES
Case1
VARIABLES
orders, stock, invoiced_orders, reference, quantity
INITIALISATION
stock, invoiced_orders, orders, quantity, reference := PRODUCTS × {0}, ∅, ∅, ∅, ∅
EVENTS
cancel_orders =
ANY
o
WHERE
  o ∈ orders − invoiced_orders
THEN
  orders := orders − {o}||
  quantity := {o} ≪ quantity||
  reference := {o} ≪ reference
END;
new_orders =
ANY
o, q, p
WHERE
  o ∈ ALL_ORDERS − orders
  q ∈ N*
  p ∈ PRODUCTS
THEN
  orders := orders ∪ {o}||
  quantity(o) := q||
  reference(o) := p
END;
delivery_to_stock =
ANY
p, n
WHERE
  p ∈ PRODUCTS
  n ∈ N
THEN
  stock(p) := stock(p) + n
END
END

Figure 2: Case 2
MODEL
  Case3
REFINES
  Case2
VARIABLES
  orders, stock, invoiced_orders, reference, quantity, time, t
INITIALISATION
  stock, invoiced_orders, orders, quantity, reference := PRODUCTS × {0}, ∅, ∅, ∅, ∅
EVENTS
invoice_order =
  ANY
  o
  WHERE
    o ∈ orders − invoiced_orders ∧
    quantity(o) ≤ stock(reference(o)) ∧
    ∃d .
      d ∈ orders − invoiced_orders ∧
      quantity(d) ≤ stock(reference(d))
    ⇒
      time(o) ≤ time(d)
  THEN
    invoiced_orders := invoiced_orders ∪ {o}||
    stock(reference(o)) := stock(reference(o)) − quantity(o)
END;
new_orders =
  ANY
  o, q, p
  WHERE
    o ∈ ALL_ORDERS − orders
    q ∈ N*
    p ∈ PRODUCTS
  THEN
    orders := orders ∪ {o}||
    quantity(o) := q||
    reference(o) := p||
    time(o) := t||
    t := t + 1
END;
cancel_orders =
  ANY
  o
  WHERE
    o ∈ orders − invoiced_orders
  THEN
    orders := orders − {o}||
    quantity := {o} ⊲ quantity||
    reference := {o} ⊲ reference||
    time := {o} ⊲ time
END
END

Figure 3: Case 3