Softw Syst Model (2005) / Digital Object Identifier (DOI) 10.1007/s10270-005-0083-4

State-based versus event-based specifications for information systems: a comparison of B and EB^3

Benoît Fraikin[1], **Marc Frappier**[1], **Régine Laleau**[2]

[1] GRIL, Département d'informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada J1K 2R1
e-mail: {Benoit.Fraikin,Marc.Frappier}@usherbrooke.ca
[2] Laboratoire LACL, Université Paris 12, IUT de Fontainebleau – Département Informatique, Route Hurtault, 77300 Fontainebleau, France
e-mail: laleau@univ-paris12.fr

Published online: 25 May 2005 – © Springer-Verlag 2005

Abstract. This paper compares two formal methods, B and EB^3, for specifying information systems. These two methods are chosen as examples of the state-based paradigm and the event-based paradigm, respectively. The paper considers four viewpoints: functional behavior expression, validation, verification, and evolution. Issues in expressing event ordering constraints, data integrity constraints, and modularity are thereby considered. A simple case study is used to illustrate the comparison, namely, a library management system. Two equivalent specifications are presented using each method. The paper concludes that B and EB^3 are complementary. The former is better at expressing complex ordering and static data integrity constraints, whereas the latter provides a simpler, modular, explicit representation of dynamic constraints that are closer to the user's point of view, while providing loosely coupled definitions of data attributes. The generality of these results from the state-based paradigm and the event-based paradigm perspective are discussed.

Keywords: State-based paradigm – Event-based paradigm – EB^3 – B – Process algebra – Information system – Formal specification

1 Introduction

This paper compares two formal methods, B [1] and EB^3 [20], for specifying information systems (IS). The paper covers several viewpoints. The first is the ease of describing the *functional behavior* of IS, which includes expressing event ordering, data structures, and modularity, and enforcing integrity constraints. The second viewpoint is specification validation, which includes checking the specification against user requirements by review, inspection, walk-through, animation, or scenario analysis. The next viewpoint is specification verification, which consists in checking the specification against formal properties or refinement of the specification. Finally, we consider specification evolution in order to understand the ease with which a specification can evolve to meet new user requirements. For each viewpoint, we identify the relative strengths and weaknesses of each paradigm. Ultimately, this comparison leads us to determine a specification process that includes the best of both methods.

Our comparison is domain specific and restricted to information systems. Information systems differ from other software systems by their strong dependence on the "real world" [24]. Firstly, the information they manage necessarily represents elements of this real world that are relevant to the user. This implies that an IS has to faithfully represent these elements, which leads to the definition of a great number of complex, interrelated data structures subject to strong integrity constraints. Moreover, the programs processing this information are simultaneously used by a large number of human beings in their work. Thus, they need to offer appropriate interfaces in order to be used appropriately or, at the extremum, just to be used. These programs are often large, managing complex ordering constraints among business events, to enforce a business process, but they are not necessarily of great algorithmic difficulty. Distribution and real-time constraints are usually not an issue for the formal specification of IS user requirements. Although they may be of concern at the design level, we do not address design issues in this paper. On the other hand, data integrity is a critical issue for IS, especially with the rapid deployment of the World Wide Web. Organizations are developing web access to their IS's, thereby increasing the need for high-quality systems. Data integrity becomes a critical issue when ordinary clients can use an organization's IS.

The B method has been shown to be appropriate for describing ISs [25, 27, 30] and deriving relational database implementations [26, 28]. It supports the whole life cycle, from requirements specification to implementation; it is supported by industrial-strength case tools that have been successfully used on large-scale, safety-critical industrial applications [4]. EB^3 has been defined for the purpose of specifying ISs. It is based on entities, process algebra, traces, and recursive functions defined on traces. It is chiefly event-driven, but also includes some state-oriented constructs in order to facilitate IS specification.

We have chosen B and EB^3 because they embrace several distinguishing features of the state-based paradigm and the event-based paradigm, while being both adequate for the specification of ISs. We hope to draw conclusions that are generally applicable for these two paradigms. Given their broadness, however, we do not claim to cover their entire scope.

Our comparison is illustrated by specifying part of a library management system in both B and EB^3. Section 2 provides a textual description of the user requirements. Section 3 introduces EB^3 while providing a complete specification of the library system. Section 4 describes the same behavior using the B language. Note that, in order to highlight some features of each language, we have deliberately inserted *errors* in the specifications. Both specifications contain the same errors, although they take different forms due to the different characteristics of each paradigm. The correction of these errors are addressed in Sect. 5, which compares the two specifications from the viewpoints of functional behavior description, validation, verification, and evolution. We conclude in Sect. 6 by providing a summary of the relative strengths and weaknesses of each language, from which an integrated specification process combining the strengths of each method, is derived.

2 The user requirements of a library management system

In this section, we provide a textual description of the user requirements for a simple library management system that caters for book loans and reservations to members. Even though they are quite basic, the requirements are complex enough to illustrate the difference in style between the event-based and the state-based paradigms. These requirements are numbered so that we can refer to them later in the paper.

1. A book can be acquired by the library. It can also be discarded, but only if it has not been lent or reserved.
2. An individual must join the library in order to borrow a book.
3. A book can be reserved if and only if it has been lent or already reserved by another member.
4. A member cannot borrow a book or renew a loan if the book has already been reserved.
5. If many members have reserved a book, the first one who reserved it is allowed to take it when it is returned, unless this member has decided to cancel his reservation.
6. Anyone who has reserved a book can cancel the reservation at anytime before the reservation has been used.
7. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or canceled.
8. The library system must be able to provide the current number of loans for a given member, the current borrower of a given book, and the list of books and their borrowers by book category.
9. A member cannot borrow more than the loan limit defined at the system level for all users.

3 The EB^3 specification

3.1 An overview of EB^3

The EB^3 method [20] has been specially designed to specify the functional behavior of ISs. An EB^3 specification consists of the following elements:

1. A user requirements class diagram, which includes entities, associations, and their respective actions and attributes.
2. A process expression (PE), denoted by `main`, which defines valid input traces.
3. Recursive functions, defined on the traces of `main`, that assign values to entity and association attributes.
4. Input-output rules, which assign an output to each valid input trace.

The denotational semantics of an EB^3 specification is given by a relation R defined on $\mathcal{T}(\texttt{main}) \times O$, where $\mathcal{T}(\texttt{main})$ denotes the traces accepted by `main` and O is the set of output events. The operational behavior of an EB^3 specification may be explained as follows. Let `trace` denote the system trace, which contains the *valid* input events accepted so far in the execution, let $t::\sigma$ denote the right append of element σ to trace t, and let `[]` denote the empty trace. Then we have:

```
trace := [];
forever do
    receive input event $\sigma$;
    if main can accept trace::$\sigma$ then
        trace := trace::$\sigma$;
        send output event $o$ such that $(\texttt{trace}, o) \in R$;
    else
        send error message;
```

An action denotes a service of the IS that the user can invoke. The signature of an action is given by a declaration

$$\texttt{a}(p_1 : T_1, \dots, p_n : T_n) : (p_{n+1} : T_{n+1}, \dots, p_m : T_m)$$

where `a` is called the *label* of the action, $p_1, \dots, p_n$ are input parameters of types $T_1, \dots, T_n$ and $p_{n+1}, \dots, p_m$ are output parameters of types $T_{n+1}, \dots, T_m$. An action

$\mathtt{a}(t_1, ..., t_n)$ constitutes a process expression. The special symbol "_" may be used as an actual parameter of an action to denote an arbitrary value of the corresponding type. An input event is an instantiation of an action. We use Σ_e to denote the set of input events.

EB^3 notation uses a process algebra similar to LOTOS [7], CCS [32], and CSP [22]. Process expressions can be combined using the following operators. The *sequence* $E_1.E_2$ first executes E_1. When E_1 completes its execution, it transforms itself into PE $\boxdot$, which denotes successful completion; expression E_2 may then start its execution. The choice E_1 `|` E_2 can execute either E_1 or E_2, as in a regular expression. The *Kleene closure* E`^*`, also drawn from regular expressions, can execute E an arbitrary number of times (zero or many). It transforms into $\boxdot$ when it has completed its execution. PE E_1 `|[Δ]|` E_2 is the parameterized parallel composition of E_1 and E_2 with synchronization on actions that belong to set Δ. Drawn from LOTOS, it has the following meaning: if E_1 (or, dually, E_2) can execute σ, and the label of σ is not in Δ, then the composition can execute σ. When the label of σ is in Δ, then E_1 and E_2 must synchronize, that is, they must both execute σ. Operations `|||` and `||` are the *interleave* and *parallel composition* of CSP, respectively; they are syntactic abbreviations of `|[...]|`. PE E_1 `|||` E_2 denotes E_1 `|[]|` E_2 (i.e., no synchronization between E_1 and E_2). PE E_1 `||` E_2 denotes E_1 `|[Δ]|` E_2, where Δ is the intersection of the action labels of E_1 and E_2 (i.e., synchronization between E_1 and E_2 on shared actions). The *guard* p `==>` E can execute E if predicate p holds; otherwise, execution is blocked until p is satisfied. The truth value of p may evolve over time with the execution of actions, since p can refer to the system trace. The *process call* $P(t_1, \ldots, t_n)$ executes the body of the process definition of P with actual parameters $t_1, \ldots, t_n$. Quantification (indexing) is permitted for choice (`|x : T : ...`), interleaving (`|||x : T : ...`) and parameterized parallel composition (`|[...]|x : T : ...`). For instance, the quantification `|||x : 1,2,3 : a(x)` denotes the PE `a(1) ||| a(2) ||| a(3)`. The precedence of operators is, from highest to lowest: `^*`, `==>`, `.`, `|`, (`|||`, `||`, `|[...]|`, which have the same precedence), and quantified expressions.

The main differences between EB^3 and LOTOS, CSP, and CCS are i) EB^3 allows one to use a single state variable, the system trace, in predicates of guard statements; ii) EB^3 uses a single operator, concatenation (as in regular expressions), instead of prefixing and sequential composition, which makes specifications easier to read and write. Moreover, the operational semantics of EB^3 is interested only in the traces of a process expression. For instance, PE `a.(b | c)` and PE `a.b | a.c` have the same semantics, which is the set of traces $\{a, ab, ac\}$.

The EB^3 process algebra has an operational semantics; Fig. 1 shows a subset of its transition rules. They inductively define a transition relation $E_1 \xrightarrow{\sigma} E_2$, which denotes that process expression E_1 can execute action σ and transform into process expression E_2. For instance, rule EB^3-1 states that an action can execute itself and transform into $\boxdot$. Rule EB^3-5 states that a sequence $E_1 . E_2$ can execute σ if E_1 can execute σ. Moreover, if E_1' is the result of executing σ on E_1, then $E_1' . E_2$ is the result of executing σ on $E_1 . E_2$. The symbol λ denotes an internal action that a process may execute without requiring input from the environment.

$$(\text{EB}^3\text{-1}): \frac{\sigma \in \Sigma_e \cup \{\lambda\}}{\sigma \xrightarrow{\sigma} \boxdot} \qquad (\text{EB}^3\text{-2}): \frac{\Phi \wedge E \xrightarrow{\sigma} E'}{\Phi \texttt{==>} E \xrightarrow{\sigma} E'}$$

$$(\text{EB}^3\text{-3}): \frac{}{E\texttt{\^{}*} \xrightarrow{\lambda} \boxdot} \qquad (\text{EB}^3\text{-4}): \frac{E \xrightarrow{\sigma} E'}{E\texttt{\^{}*} \xrightarrow{\sigma} E' . E\texttt{\^{}*}}$$

$$(\text{EB}^3\text{-5}): \frac{E_1 \xrightarrow{\sigma} E_1'}{E_1 . E_2 \xrightarrow{\sigma} E_1' . E_2} \qquad (\text{EB}^3\text{-6}): \frac{E \xrightarrow{\sigma} E'}{\boxdot . E \xrightarrow{\sigma} E'}$$

$$(\text{EB}^3\text{-7}): \frac{E_1 \xrightarrow{\sigma} E_1'}{E_1 \,|||\, E_2 \xrightarrow{\sigma} E_1' \,|||\, E_2} \qquad (\text{EB}^3\text{-8}): \frac{E_1 \xrightarrow{\sigma} E_1'}{E_1 \,|\, E_2 \xrightarrow{\sigma} E_1'}$$

$$(\text{EB}^3\text{-9}): \frac{E_2 \xrightarrow{\sigma} E_2'}{E_1 \,|||\, E_2 \xrightarrow{\sigma} E_1 \,|||\, E_2'} \qquad (\text{EB}^3\text{-10}): \frac{E_2 \xrightarrow{\sigma} E_2'}{E_1 \,|\, E_2 \xrightarrow{\sigma} E_2'}$$

Fig. 1. A subset of the transition rules for the EB^3 process algebra

As a simple example of EB^3 specification, consider the following requirements:

"The system must accept A, followed by an arbitrary number of B, and then accept C. Output `ok` is produced for each A or B accepted. The system must output the number of B's accepted when input C is accepted." (1)

These requirements are represented by the EB^3 specification in Fig. 2. The requirements class diagram has been omitted. The signature of the actions is the following: `A:void`, `B:NAT`, and `C:void`. The special type `void` denotes an action with no input-output rule; the output of such an action is `ok` if it can be accepted by the PE `main`, otherwise its output is `error`. The process expression `main` defines the valid input traces. The recursive function `BCounter` computes the number of `B`'s accepted from the system trace. A complete description of the recursive functions and their uses can be found in [20]. The definition of a recursive function is written in a CAML[1] style. A trace is represented by a list. Operators `last` and `front`, respectively, return the last element and all but the last element of a list; they return the special value `nil` when the list is empty. Finally, the symbol "_" can match any value and is consequently used to provide default instructions. Input-output rule `R1` defines the output of input `C`.

[1] CAML is a functional language.

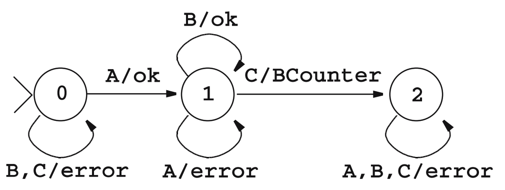
```
main = A . B^* . C

BCounter(trace : VALID_TRACE): NAT =
  match last(trace) with
        nil -> 0                        |
        B   -> BCounter(front(trace))+1 |
        _   -> BCounter(front(trace))
```

```
        Rule R1
          input C
          output BCounter(trace)
        EndRule
```

Fig. 2. EB^3 specification for the user requirement (1)

(a)



(b)

Order No.	Input	Output
1	B	error
2	A	ok
3	B	ok
4	A	error
5	C	1

Fig. 3. a The STD of the EB^3 specification; **b** A sample execution sequence

Figure 3 illustrates this EB^3 specification by providing an equivalent state-transition diagram (STD) and a sample execution sequence. As usual, a transition with input σ and output o is denoted by $q \xrightarrow{\sigma/o} q'$ on this diagram. Note that, following the operational semantics of EB^3, output `error` is sent when an input cannot be accepted by `main`; output `ok` is produced when there is no input-output rule for a given input. Table b in Fig. 3 shows the inputs submitted by the user and the output produced by the system.

Table b in Fig. 3 denotes the following input-output pairs, which are elements of relation R.

(A, ok)
(AB, ok)
$(ABC, 1)$

The first element of a pair is a valid trace (i.e., an instance of the system trace); the second element is the output produced for that trace. Note that inputs which were not

```
specification Example [ A, B, C, ok, Cout ] : noexit
behavior
    (
            A ; ok ; P [B, C, ok, Cout]
      |[B, Cout]|
            BCounter [B, Cout] (0)
    )
where
    process P [B, C, ok, Cout] : noexit :=
            C ; Cout ?nbOfB : Nat ; stop
        []
            B ; ok ; P [B, C, ok, Cout]
    endproc
    process BCounter [B, Cout] (cpt:Nat) : noexit :=
            B ; BCounter [B, Cout] (Succ(cpt))
        []
            Cout !cpt ; stop
    endproc
endspec
```

Fig. 4. The LOTOS specification of requirements (1)

accepted by process expression `main` are not included in traces of R, and that outputs are not part of the trace.

Aside from the syntactical distinctions, EB^3 also differs from CSP, CCS, and LOTOS in the treatment of outputs. In EB^3, a process expression applies only to inputs; outputs are defined by input-output rules. In other process algebras, both inputs and outputs are managed using a process expression. In fact, they do not distinguish between input and output. Figure 4 shows a LOTOS specification for the requirements (1) in order to illustrate the differences between EB^3 and other process algebras. Its main process is defined in the `specification` clause. It consists of a parameterized parallel composition whose first operand executes actions `A,B,C` (called *gates* in LOTOS) using a recursive process `P`, due to the absence of Kleene closure in LOTOS. The operator "`;`", called action prefixing, is one of the LOTOS operators denoting sequential execution. The second operand calls the process `Bcounter`, which keeps tracks of the number of `B`'s accepted. The synchronization between them occurs at gates `B` and `Cout`. Since both inputs and outputs are represented by gates, the outputs of input events `A,B,C` are represented by gates `ok` and `Cout`; the value exchanged at `Cout` is determined by the synchronisation between `P` and `BCounter`. At this gate, the process `BCounter` offers the value of the variable `cpt`, which is denoted by using the decoration `!`. The process `P` synchronizes with `BCounter` at this gate and accepts any value, which is represented by decoration `?` on the variable `nbOfB`. Note that the syntax of LOTOS requires the declaration, in the process heading, of the gates (between `[` and `]`) and of the variables (between `(` and `)`) used in the process. The traces of this LOTOS specification are equivalent to the input-output pairs of relation R in the EB^3 specification. However, note that there is no gate `error` in the LOTOS specification nor in the EB^3 specification. In EB^3, error

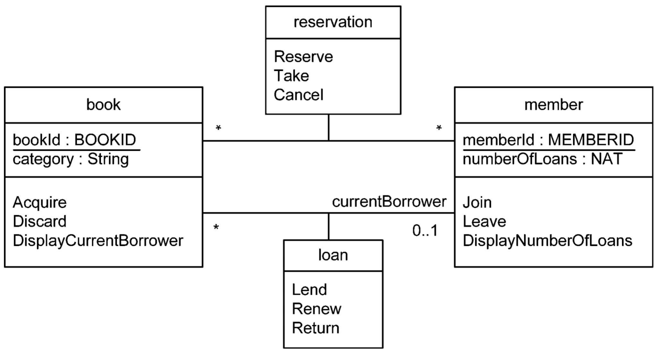


Fig. 5. EB^3 specification: User requirements class diagram of the library IS

management is defined by default in the operational semantics. This concept does not exist in the standard semantics of LOTOS.

The operational semantics of EB^3 states that an output is produced for each input received. This choice, inspired from the Cleanroom method [31], stems from the necessity in IS to inform the user of the result of his request. The specification of complex output computation using only a process algebra is very difficult.

An ongoing project at the University of Sherbrooke aims at automatically generating IS implementations from EB^3 specifications. Details on this topic can be found in [17, 18]. The process algebra interpreter developed so far can execute process expressions with reasonable performance. In many cases (e.g., the specification patterns identified in [20]), it can be used as a substitute for a human-derived implementation of the specification.

```
Acquire(bookId:BOOKID, category:STRING):void
Discard(bookId:BOOKID):void
Join(memberId:MEMBERID):void
Leave(memberId:MEMBERID):void
Lend(memberId:MEMBERID, bookId:BOOKID):void
Renew(memberId:MEMBERID, bookId:BOOKID):void
Return(memberId:MEMBERID, bookId:BOOKID):void
Reserve(memberId:MEMBERID, bookId:BOOKID):void
Cancel(memberId:MEMBERID, bookId:BOOKID):void
Take(memberId:MEMBERID, bookId:BOOKID):void
DisplayCurrentBorrower(bookId:BOOKID):
                      (memberId:MEMBERID)
DisplayNumberOfLoans(memberId:MEMBERID):
                    (numberOfLoans:NAT)
DisplayBorrowerByCategory():
                 ( (category:STRING, bookId:BOOKID,
                    memberId:MEMBERID)^* )
```

Fig. 6. EB^3 specification: Signature of actions

3.2 The library specification

Figure 5 shows the user requirements class diagram used to construct the specification. There are two entity types, `member` and `book`, and two associations between them, `reservation` and `loan`, with their corresponding actions and attributes. By convention, key entity attributes are underlined, action names have uppercase initials, process and function names start with a lowercase, and types are in uppercase. The signature of the actions are provided in the Fig. 6. `Take` and `Lend` are two distinct actions; they both allow a member to borrow a book, but in different circumstances. `Lend` triggers a 'regular' loan, i.e., the book is available and the member borrows it.

```
main =    ( ||| bId : BOOKID : book(bId)^* )
       || ( ||| mId : MEMBERID : member(mId)^* )
       || DisplayBorrowerByCategory()^*

book(bId : BOOKID) = Acquire(bId,_)
                   .   (
                           ( | mId : MEMBERID : loan(mId,bId) )^*
                       ||  ( ||| mId : MEMBERID : reservation(mId,bId)^* )
                       ||  DisplayCurrentBorrower(bId)^*
                       )
                   .   Discard(bId)

member(mId : MEMBERID) = Join(mId)
                       .   (
                               ( ||| bId : BOOKID : loan(mId,bId)^* )
                           ||  ( ||| bId : BOOKID : reservation(mId,bId)^* )
                           ||  DisplayNumberOfLoans(mId)^*
                           )
                       .   Leave(mId)

loan(mId : MEMBERID, bId : BOOKID) =
          ( Lend(mId,bId) | Take(mId,bId) ) . Renew(mId,bId)^* . Return(mId,bId)

reservation(mId : MEMBERID, bId : BOOKID) =
          Reserve(mId,bId) . ( isFirst(trace,mId,bId) ==> Take(mId,bId) |  Cancel(mId,bId) )
```

Fig. 7. EB^3 specification: Process definitions

Take is used in connection with a reservation. When a book is on loan, a member can reserve it (through action Reserve); when the book becomes available, he must use action Take to borrow it. The output of the action DisplayBorrowerByCategory is a list, denoted by the use of operator ^*, which is also a list constructor in the EB^3 type system.

Figure 7 describes the process main, which calls a process expression for entity type member and entity type book. These processes in turn call one process for each association (loan and reservation). The expression isFirst(trace,mId,bId) is a function call returning a Boolean value; it returns true when mId is first in the reservation queue of book bId; otherwise, it returns false. Figure 9 provides the definition of this recursive function; in addition, there is one function for each non-key entity attribute. Note that a recursive function returns nil when it is undefined for some argument.

Figure 8 provides the input-output rules for the three actions that do not have void as an output type. Rules R1 and R2 define the output for actions DisplayCurrentBorrower(bId) and DisplayNumberOfLoans(mId) by referring to the appropriate recursive function. Rule R3 defines the output for the action DisplayBorrowerByCategory(), which is easier to express with a syntax similar to SQL's SELECT statement. This statement is

```
Rule R1
  input DisplayCurrentBorrower(bId)
  output currentBorrower(trace,bId)
EndRule

Rule R2
  input DisplayNumberOfLoans(mId)
  output numberOfLoans(trace,mId)
EndRule

Rule R3
  input DisplayBorrowerByCategory()
  output SELECT category, bookId, currentBorrower
         FROM book
         WHERE currentBorrower IS NOT NULL
EndRule
```

Fig. 8. EB^3 specification: Input-output rules

```
category(trace : VALID_TRACE, bId : BOOKID): STRING =
    match last(trace) with
        nil -> nil              |
        Acquire(bId,cat) -> cat |
        Discard(bId) -> nil     |
        _ -> category(front(trace),bId)

currentBorrower(trace : VALID_TRACE, bId : BOOKID): MEMBERID =
    match last(trace) with
        nil -> nil              |
        Return(mId,bId) -> nil  |
        Lend(mId,bId) -> mId    |
        Take(mId,bId) -> mId    |
        _ -> currentBorrower(front(trace),bId)

numberOfLoans(trace : VALID_TRACE, mId : MEMBERID): NAT =
    match last(trace) with
        nil -> nil                                          |
        Join(mId) -> 0                                      |
        Lend(mId,_) -> numberOfLoans(front(trace),mId) + 1   |
        Take(mId,_) -> numberOfLoans(front(trace),mId) + 1   |
        Return(mId,_) -> numberOfLoans(front(trace),mId) - 1 |
        Leave(mId) -> nil                                   |
        _ -> numberOfLoans(front(trace),mId)

reservationQueue(trace : VALID_TRACE, bId : BOOKID): LIST of MEMBERID =
    match last(trace) with
        nil -> []                                                   |
        Reserve(mId,bId) -> reservationQueue(front(trace),bId)::mId   |
        Cancel(mId,bId) -> reservationQueue(front(trace),bId) - {mId} |
        _ -> reservationQueue(front(trace),bId)

isFirst(trace : VALID_TRACE, mId : MEMBERID , bId : BOOKID): BOOLEAN =
    match first(reservationQueue(trace,bId)) with
        mId -> true  |
        _ -> false
```

Fig. 9. EB^3 specification: Function definitions

applied to the relational database schema that can be generated from the requirements class diagram using the algorithm described in [12] (chapter 7) (see [20] for more details). In particular, this gives the table `book` with the attributes `bookId`, `category`, and `currentBorrower`.

The primary use of recursive functions is to extract information from the system trace. They are used to provide a response to some input events like the function `currentBorrower(trace,bId)`, which yields the output value of action `DisplayCurrentBorrower(bId)`. They are also quite useful in handling constraints in the specification with the use of guard as with `isFirst` in the process `reservation`. In this way, the use of functions is quite similar to a precondition. Further discussion on this subject is provided in Sect. 5.1.1.

To illustrate how the system trace and the recursive functions are related, consider the following valid trace

```
t = Acquire(b1,c1) . Acquire(b2,c2) .
    Join(m1) . Lend(m1,b1)
```

Then, we have

```
currentBorrower(t,b1) = m1
```

and

```
current\-Borrower(t,b2) = nil.
```

4 The B specification

B, a formal method developed by Abrial [1], supports a large segment of the development life cycle: specification, refinement and, implementation. It ensures, through refinement steps and proofs, that the code satisfies its specification. B has been used in industrial projects [4] and commercial case tools are available to help the specifier during the development process. These constitute the main arguments for using B rather than Z (type checking and tool assistance in proof, but limited tool support for refinement and no automated proof). The specification provided in this paper could also be expressed in a similar fashion in these state-based languages and others of the same family (such as ASM [6]).

4.1 Overview of B

In B, specifications are organized into abstract machines (similar to classes and modules). Each machine encapsulates state variables, an invariant constraining the state variables, and operations on the state variables. The invariant is a predicate in a simplified version of the ZF-set theory, enriched by many relational operators. In an abstract machine, it is possible to declare abstract sets by giving their name without further details. This allows the actual definition of types to be deferred to implementation.

Operations are specified in the Generalized Substitution Language, which is a generalization of Dijkstra's guarded command notation. Hence, operations are defined using substitutions, which are like assignment statements. A substitution provides the means for identifying which variables are modified by the operation, while avoiding mentioning those that are not. The generalization allows the definition of non-deterministic and preconditioning substitutions. The preconditioning substitution is of the form **PRE** P **THEN** S **END**, where P is a predicate and S a substitution. When P holds, the substitution is executed; otherwise, the result is undetermined and the substitution may abort.

The B notation provides various mechanisms to construct large machines from smaller ones. At an abstract level, the most useful mechanisms are **USES** and **INCLUDES**. A machine A may *use* another machine B: the variables of B can be used but not modified in A. A machine A may *include* another machine B: the variables of B can be read in A but updated only by using the operations of B. The aim of these mechanisms and restrictions is to achieve incremental specifications with separate consistency proofs.

4.2 The library specification

Each entity and association is specified in separate machines. B_Member and B_Book (see below) are built in the same way. Let us explain the first one. The variable $members$ represents the set of existing members and is included in the abstract set $MEMBER$, which represents the set of all possible members. The variable $memberId$ is defined as a function from $members$ to the type of the attribute $memberId$. It is an injective function ($\rightarrowtail$) since the attribute is defined as a key. This gives the static part of B_Member:

MACHINE B_Member

SETS $MEMBER$; $MEMBERID$

VARIABLES $members$, $memberId$

INVARIANT $members \subseteq MEMBER \ \wedge$
$memberId \in members \rightarrowtail MEMBERID$

Each machine has an INITIALISATION clause that gives an initial value for each variable.

INITIALISATION $members := \{\} \ \|$
$memberId := \{\}$

B_Member is completed by the definition of two basic operations: `B_Join`, which creates a new member, and `B_Leave`, which deletes an existing member (see below). The operator $\|$ denotes the simultaneous execution of its operands (which can be any substitution). The operator ANY allows creation of a non-deterministic substitution; it chooses an arbitrary value for the variable $memb$ verifying the WHERE predicate and executes the substitutions specified between the THEN and the END keywords. The anti-corestriction operator ($\unrhd$) of the operation `B_Leave` is defined as follows: $r \unrhd A = \{(a, b) \mid (a, b) \in$

$r \wedge b \notin A\}$. The *ran* operator, applied to a relation, gives its codomain. The notation $a \mapsto b$ denotes an element of a relation. The inverse of *rel*, where *rel* stands for a relation, is denoted rel^{-1}.

```
OPERATIONS
  B_Join($mId$) $\triangleq$
  PRE  $mId \in MEMBERID - ran(memberId) \wedge$
          $members \subset MEMBER$
  THEN
     ANY $memb$ WHERE
            $memb \in MEMBER - members$
     THEN
        $memberId := memberId \cup \{memb \mapsto mId\}$  ||
        $members := members \cup \{memb\}$
     END
  END;

  B_Leave($mId$) $\triangleq$
  PRE  $mId \in ran(memberId)$
  THEN
     $members := members - \{memberId^{-1}(mId)\}$  ||
     $memberId := memberId \rhd\!\!\!- \{mId\}$
  END

END;
```

The machine *B_Book* is defined as follows. Note that the anti-restriction operator ($\lhd\!\!\!-$) of the operation `B_Discard` is defined as: $A \lhd\!\!\!- r = \{(a, b) \mid (a, b) \in r \wedge a \notin A\}$. $E \rightarrow F$ is the set of the total functions from E to F, whereas $E \rightarrowtail F$ is the set of the total injective functions from E to F.

```
MACHINE $B\_Book$

SETS $BOOK$; $BOOKID$

VARIABLES $books$, $bookId$, $category$

INVARIANT $books \subseteq BOOK \wedge$
                    $bookId \in books \rightarrowtail BOOKID \wedge$
                    $category \in books \rightarrow STRING$

INITIALISATION $books := \{\}$  ||
                    $bookId := \{\}$  ||  $category := \{\}$

OPERATIONS
  B_Acquire($bId, cat$) $\triangleq$
  PRE  $bId \in BOOKID - ran(bookId) \wedge$
          $cat \in STRING \wedge books \subset BOOK$
  THEN
     ANY $book$ WHERE $book \in BOOK - books$
     THEN
        $bookId := bookId \cup \{book \mapsto bId\}$  ||
        $category := category \cup \{book \mapsto cat\}$  ||
        $books := books \cup \{book\}$
     END
  END;

  B_Discard($bId$) $\triangleq$
  PRE  $bId \in ran(bookId)$
  THEN
     $books := books - \{bookId^{-1}(bId)\}$  ||
     $bookId := bookId \rhd\!\!\!- \{bId\}$  ||
     $category := \{bookId^{-1}(bId)\} \lhd\!\!\!- category$
  END

END
```

Machines *B_Reservation* and *B_Loan* (see below) represent the two associations. The variable *reservations* is defined as a relation ($\leftrightarrow$) between the sets of existing books and existing members, that is, the variables *books* and *members*. As a result, these two variables must be accessible from the machine *B_Reservation*, which is achieved by the USES clause. The variable *numReserv* is used to translate requirement 5. It associates each reservation with an integer that is incremented by one when a new reservation is created (see the operation `B_Reserve`). Note that other solutions are possible, such as defining the association *reservations* as a function that associates each book with the sequence of members who have reserved the book in chronological order. For the purpose of the paper, the choice between all the solutions does not matter. The DEFINITIONS clause introduces abbreviations used in predicates, expressions, and substitutions.

```
MACHINE $B\_Reservation$

USES $B\_Member$, $B\_Book$

VARIABLES $reservations$, $numReserv$

INVARIANT
  $reservations \in books \leftrightarrow members \wedge$
  $numReserv \in reservations \rightarrow INTEGER$

DEFINITIONS
  $theMember \triangleq memberId^{-1}(mId)$;
  $theBook \triangleq bookId^{-1}(bId)$

INITIALISATION
  $reservations := \{\}$  ||
  $numReserv := \{\}$

OPERATIONS

  B_Reserve($mId, bId$) $\triangleq$
  PRE  $mId \in ran(memberId) \wedge$
     $bId \in ran(bookId) \wedge$
     $(theBook \mapsto theMember) \notin reservations$
  THEN
     $reservations := reservations \cup$
           $\{theBook \mapsto theMember\}$  ||
     $numReserv := numReserv \cup$
           $\{(theBook, theMember) \mapsto$
              $(max(ran(numReserv) \cup \{0\}) + 1)\}$
  END;
  B_Cancel$(mId, bId) \triangleq$
  PRE $mId \in ran(memberId) \;\wedge$
     $bId \in ran(bookId) \;\wedge$
     $(theBook \mapsto theMember) \in reservations$
  THEN
     $reservations := reservations -$
           $\{theBook \mapsto theMember\} \;\|$
     $numReserv :=$
           $\{theBook \mapsto theMember\} \lhd\!\!\!- numReserv$
  END

END;
```

The machine *B_Loan* is built in the same way as *B_Reservation*. The variable *loans* is a partial function ($\nrightarrow$) that associates a book with its borrower. The *dom* operator, applied to a relation, gives its domain.

```
MACHINE $B\_Loan$

USES $B\_Member$,$B\_Book$

VARIABLES $loans$

INVARIANT $loans \in books \nrightarrow members$

DEFINITIONS
  $theMember \triangleq memberId^{-1}(mId)$;
  $theBook \triangleq bookId^{-1}(bId)$

INITIALISATION $loans := \{\}$
OPERATIONS
  B_Lend$(mId, bId) \triangleq$
  PRE $mId \in ran(memberId) \;\wedge$
     $bId \in ran(bookId) \;\wedge$
     $theBook \notin dom(loans)$
  THEN
     $loans(theBook) := theMember$
  END;

  B_Return$(mId, bId) \triangleq$
  PRE $mId \in ran(memberId) \;\wedge$
     $bId \in ran(bookId) \;\wedge$
     $(theBook, theMember) \in loans$
  THEN
     $loans := \{theBook\} \lhd\!\!\!- loans$
  END;

  $mId \longleftarrow$ B_CurrentBorrower $(bId) \triangleq$
  PRE $bId \in ran(bookId) \;\wedge$
     $theBook \in dom(loans)$
  THEN
     $mId := memberId(loans(theBook))$
  END;

  $nol \longleftarrow$ B_NumberOfLoans $(mId) \triangleq$
  PRE $mId \in ran(memberId) \;\wedge$
     $theMember \in ran(loans)$
  THEN
     $nol := card(loans^{-1}[\{theMember\}])$
  END;

  $report \longleftarrow$ B_BorrowerByCategory $() \triangleq$
  BEGIN
     $report := \{cat, bId, mId \,|bId \in ran(bookId) \wedge$
         $bookId^{-1}(bId) \in dom(loans) \wedge$
         $cat = category(bookId^{-1}(bId)) \wedge$
         $mId = loans(bookId^{-1}(bId))\}$
  END

END
```

Once all the machines describing the state of the IS have been defined, a new machine, called *B_Library* (see below), describing all the services of the IS, is created. It includes all the previous machines in order to call the operations that modify the state variables. An operation available from this machine corresponds to one service and is either an operation of this machine or a promoted operation (using the PROMOTES clause) from included machines. In the operation `L_Take`, the restriction operator ($\lhd$) is defined as follows: $A \lhd r = \{(a,b) \mid (a,b) \in r \wedge a \in A\}$. The operation `L_Renew` may look quite useless, since it only has a *skip* instruction, which means "do nothing", in its body. A real IS would probably update the expected return date, which, for the sake of simplicity, we do not take into account.

```
MACHINE $B\_Library$

INCLUDES $B\_Member$, $B\_Book$,
  $B\_Loan$, $B\_Reservation$

PROMOTES $B\_Join$, $B\_Acquire$, $B\_Cancel$,
  $B\_Reserve$, $B\_Lend$, $B\_Return$,
  $B\_CurrentBorrower$, $B\_NumberOfLoans$,
  $B\_BorrowerByCategory$

DEFINITIONS
  $theMember \triangleq memberId^{-1}(mId)$;
  $theBook \triangleq bookId^{-1}(bId)$

OPERATIONS

  L_Leave$(mId) \triangleq$
  PRE $mId \in ran(memberId) \;\wedge$
     $theMember \notin ran(reservations) \;\wedge$
     $theMember \notin ran(loans)$
  THEN
     $B\_Leave(mId)$
  END;

  L_Discard$(bId) \triangleq$
  PRE $bId \in ran(bookId) \;\wedge$
```

```
    $theBook \notin dom(reservations)\ \wedge$
    $theBook \notin dom(loans)$
  THEN
    $B\_Discard(bId)$
  END;

  L_Take$(mId, bId) \triangleq$
  PRE $mId \in ran(memberId)\ \wedge$
    $bId \in ran(bookId)\ \wedge$
    $theBook \notin dom(loans)\ \wedge$
    $(theBook, theMember) \in reservations\ \wedge$
    $numReserv(theBook, theMember) =$
          $min(numReserv$
                  $[\{theBook\} \lhd dom(numReserv)])$
    /* This ensures that the first member
       to reserve the book checks it out */
  THEN
    $B\_Lend(mId, bId)$
  END;

  L_Renew$(mId, bId) \triangleq$
  PRE $mId \in ran(memberId)\ \wedge$
    $bId \in ran(bookId)\ \wedge$
    $(theBook, theMember) \in loans$
  THEN
    $skip$
  END

END
```

Finally, a top-level machine, *B_Library_Interface*, defines the user interface, which means that only the operations of this machine are available to users. It takes into account error management, that is, each operation systematically returns a message that reports the result of the execution of the corresponding service. For each operation `OpLib` of the machine *B_Library*, there is one operation `OpLibInterface` in the top-level machine, to ensure that `OpLib` is called only if its precondition is verified. As a result, the precondition of `OpLibInterface` is just a parameter-typing precondition. Each conjunct of the precondition of `OpLib` is checked by an `IF` substitution; a suitable error message is associated with it and returned through an output parameter.

When an error is detected (the output parameter *result* is different from "OK") in the operation `Display-CurrentBorrower`, the output parameter mId is set to any element of $MEMBERID$, through the substitution $:\in$.

```
MACHINE $B\_Library\_Interface$

INCLUDES $B\_Library$

DEFINITIONS
  $theMember \triangleq memberId^{-1}(mId)$;
  $theBook \triangleq bookId^{-1}(bId)$

OPERATIONS

  $result \longleftarrow$ Leave$(mId) \triangleq$
  PRE $mId \in MEMBERID$
  THEN
    IF $mId \notin ran(memberId)$
    THEN $result :=$ "$mId$ is not an identifier of an
          existing member"
    ELSE
      IF $theMember \in ran(reservations) \vee$
        $theMember \in ran(loans)$
      THEN $result :=$ "the member identified by
        $mId$ has an existing reservation or loan"
      ELSE $L\_Leave(mId)\ \|$
        $result :=$ "$OK$"
      END
    END
  END;

  $result, mId \longleftarrow$ DisplayCurrentBorrower$(bId) \triangleq$
  PRE $bId \in BOOKID$
  THEN
    IF $bId \notin ran(bookId)$
    THEN $result :=$ "$bId$ is not an identifier of an
      existing book" $\|$
      $mId :\in MEMBERID$
    ELSE
      IF $thebook \notin dom(loans)$
      THEN $result :=$ "the book identified by $bId$
        is not lent at this time" $\|$
        $mId :\in MEMBERID$
      ELSE $result :=$ "$OK$" $\|$
        $mId := B\_CurrentBorrower(bId)$
      END
    END
  END;

...

END
```

For space reasons, the definition of the machine *B_Library_Interface* is incomplete.

A new variant of the B method, called Event B [2, 3, 8], has been introduced over the past few years. Its refinement relation allows the introduction of new events (operations). Events do not take any parameter and they are defined with guards rather than with preconditions. A precondition in B is non-blocking, which means that an operation can always be invoked: the operation terminates if its precondition is satisfied; otherwise, it may abort or produce an arbitrary result. Guards in Event B are blocking, which means that an event can only occur when its guard is satisfied. An Event B specification of the library system would be fairly similar to the one expressed in classic B, except for the handling of event parameters.

5 A comparison of the two specifications

5.1 Expression of functional behavior

In this section, we analyze how the elements of the user requirements have been translated into each specification. We concentrate on elements that were either particularly easy or difficult to specify.

The ease of expressing functional behavior is a most important issue in software specification. Specifications are first meant to be written, read, and understood by human beings. They must precisely describe the user requirements and constitute the main input of the design phase. It is commonly agreed that specification errors are the most expensive to fix. Therefore, it is critical for software quality and productivity that a specification language be easy to use by human beings in order to properly express the desired behavior of the software.

5.1.1 The EB^3 specification

In [20], Frappier and St-Denis described a strategy to correctly and efficiently design an EB^3 specification from the requirements class diagram. They identified several patterns of class diagrams and proposed a corresponding EB^3 pattern for each of them. The library specification makes use of four patterns: each entity satisfies the producer-modifier-consumer pattern and the multiple-associations pattern; association loan satisfies the one-to-many pattern and association reservation satisfies the many-to-many pattern.

Following these patterns, one can take each entity type (e.g., book and member) and express its ordering constraints on input events by using a process expression. The interactions between entities (e.g., when a member borrows a book, or reserves a book) are naturally expressed by composing entities in parallel using operator `||` in the process `main` (Fig. 7). The behavior of an association is also described by a process expression that is called by each entity. The multiplicity of an association (e.g., "* , 0..1" on the loan association, which means that a book may have from 0 to 1 borrower, and that a member may have from 0 to many loans) is expressed by selecting an appropriate quantification operator to encapsulate the call to the association process expression (e.g., `|` x when an entity is related to at most one entity; `|||` x when an entity is related to a number of entities). Several patterns have been defined to translate requirements of a class diagram into process expressions (see [20]).

Following the structure of the class diagram allows for the elements 1, 2, 6, and 7 of the user requirements to be taken into account as well as implicit requirements. For instance, the fact that two members cannot borrow the same book at the same time is not stated in the requirements, but it is described in the process expression `main` (Fig. 7) by the synchronization between books and members over the input events of the loan association.

The constraints that are not easy to express in a pure process algebraic style (i.e., without using recursive functions defined on the system trace) arise from conditions involving input events from the history of inputs and from a number of entities. For instance, to address user requirement element 5, the queue of active reservations of a book must be dealt with. The reservation process is a logical place to enforce this constraint, but its definition in a pure process algebraic style is not as obvious as the basic scenarios are. It requires defining a controller process that is synchronized with the reservation process (see Fig. 10). This controller process, called `reservationController`, takes a queue as a parameter and updates it by a recursive call to the process. It must be composed in parallel with the call to process `reservation` in the process `book`. Although this solution is perfectly acceptable, we have chosen a different solution by using a guard invoking the recursive function `isFirst`, as illustrated in Fig. 9. In general, this style is more appropriate to deal with complex inter-entity constraints; we shall see additional examples in Sect. 5.2.

Other specification styles have been studied for process algebra expressions in the context of distributed systems: the monolithic style, the state-oriented-style, the resource-oriented style, and the constraint-oriented style. The reader may consult [36] for more details.

Finally, each data attribute is defined by a recursive function on the system trace. Since the system trace contains the history of input events, any data attribute can be defined, usually quite easily. In [20], patterns are defined for attributes.

5.1.2 The B specification

Our B specification is structured according to the style presented in [25, 27], in which a translation between UML diagrams and B specifications is defined. This style en-

```
reservationController(bId : BOOKID, q : QUEUE of BOOKID) =

    ( | mId : MEMBERID : Reserve(mId,bId) . reservationController(bId,enQueue(q,mId)) )
  |
    ( | mId : MEMBERID : Cancel(mId,bId) . reservationController(bId,remove(q,mId)) )
  |
    ( | mId : MEMBERID :    first(q) = mId ==> Take(mId,bId)
                       .  reservationController(bId,deQueue(q,mId)) )
```

Fig. 10. EB^3 specification alternative: process `reservationController`

forces modularity by proposing to create one basic machine for each entity type and each association; an intermediate machine, built on the basic machines, that defines one operation for each input event; and a top-level machine managing errors for the user interface. It also simplifies the discharging of proof obligations required by the B method. Closely related styles have also been proposed [30, 34].

The key in writing a simple state-based specification of an IS is to define a proper state space. The structure of this state space depends on the input event ordering constraints and on the data inquiry operations. A class diagram, with the semantics of the entity-relationship model, is a good starting point.[2] Each class is represented by a set of instances and each class attribute is represented by a function from this set to the type of the attribute. Each association is represented by a relation between entity sets. Each operation has a precondition that determines when it can be invoked. Ordering constraints are therefore described in the precondition. The substitution of an operation must properly update the state variables in order to enable the precondition of the subsequent actions and to provide data for inquiries.

Complex ordering constraints can rapidly be expressed by defining appropriate state variables, using them in preconditions, and updating them in substitutions.

5.1.3 Comparison

The contrast between the two specifications is quite strong; they are quite orthogonal in structure. The EB^3 specification is closer to a user scenario description. The ordering relation between input events is explicit, except perhaps for expressions combined with `||`, which perform a synchronization on common actions between the operands without explicitly listing these actions.

The B specification is closer to a program, except that its state space is defined with more abstract data types. The relationship between input events is not explicit; it is described via state variables, which induces a more complex form of coupling between specification elements than in EB^3. For instance, consider the `L_Discard` operation in the machine *B_Library*. Its precondition must refer to state variables from *B_Book*, *B_Loan*, and *B_Reservation*. Hence, an operation that seems, at first hand, to involve a book only, is, in fact, intimately related to state variables from other components. In the EB^3 specification, it is sufficient to say that the event `Discard` occurs after the execution of `reservation` and `loan`; there is no reference to the internal details of these processes.

In the EB^3 specification, guards referring to functions defined on the system trace are very close to preconditions of B operations. Hence, for input events subject to more complex ordering constraints, B and EB^3 are quite similar.

The same data attributes usually exist in both specifications, although the B specification may involve more attributes in order to express ordering constraints. As a first example, imagine that a book can be acquired only once; that is, it cannot be reacquired after it has been discarded. In EB^3, this change is made by removing the Kleene closure operator `^*` on the call to process `book` in the main process, as follows.

```
main =    ( ||| bId : BOOKID : book(bId) )
             ...
```

In B, there are a number of ways of expressing this constraint. One of them, which involves a minimal number of changes to the existing B specification, is to define a new state variable, *allBooks*, which contains the set of all books acquired so far.

INVARIANT
$allBooks \subseteq BOOK\ \wedge$
$books \subseteq allBooks$

When a book is acquired, it is added both to *books* and *allBooks*; when a book is discarded, it is removed from `books` but kept in *allBooks*. The precondition of `B_Acquire` is changed so that a book can be acquired when it doesn't belong to the set *allBooks*.

A second example is the specification of the attribute *numReserv* in the machine *B_Reservation*. This attribute is used in the precondition of `L_Take` to ensure that only the first member to reserve a book can take it (requirement 5). Of course, this attribute must be updated in operations that add or cancel a reservation (here `B_Reserve` and `B_Cancel`). In EB^3, the ordering constraint is specified by the guard `isFirst(trace,mId, bId)` in the process definition of `reservation`. Note that we could have defined *reservations* as an ordered association instead of defining a new attribute. In any case, the above-mentioned remarks would have always been valid. These two small examples illustrate that expressing some simple ordering constraints sometimes induces unexpected complexities in the state-based specification.

Modularity is also expressed very differently. In B, the state space is decomposed into a number of machines; operations encapsulate the description of what happens to the state variables when a transition occurs. In EB^3, behavior is encapsulated into process expressions and data values are encapsulated into a function defining the value of an attribute (of an entity or an association). Hence, it is very easy in B to determine what happens to state variables when an input event is processed. Conversely, it is very difficult to determine how a state variable evolves, because this information is scattered over all operations that modify it. In EB^3, it is exactly the opposite: it is difficult to determine the effect of an input event on attributes, because this information is scattered over several function definitions, whereas how an attribute is influenced by input events is immediately evident.

[2] This diagram would be similar to a requirements class diagram (e.g., of the EB^3 specification), with some minor differences for attributes.

Error management is implicit in EB^3 and defined in the semantics of the language. Error management in B is explicit; it takes place in the top-level (interface) machine where operations are coded with IF-THEN-ELSE rather than PRE-THEN. This approach provides the flexibility to define precise error messages, rather than the generic `ok` and `error` used in EB^3.

Overall, the connection between a B specification and an EB^3 specification is the following. The preconditions of B operations correspond to the process expressions of the EB^3 specification. The basic substitutions (i.e., :=) of B operations correspond to recursive functions on the system trace and contribute to the definition of ordering constraints.

These facts lead us to conclude that, in the general case, the structure of an EB^3 specification is closer to the structure of the user requirements than a B specification. From a user's point of view, the value of an IS lies in the information it provides and in the assurance that data integrity is preserved by event processing. The issue of checking data integrity is addressed in the next two sections. For now, we consider the issue of defining the data and stating how it is updated. In EB^3, each data attribute is defined on its own by a single function. This specification style is closer to the user view. Because it is the data that matters each data attribute can be described independently, one by one. In B, the user must consider a partial view of a number of attributes to describe what happens when an event is executed. For instance, it is easier for a user to say that the current borrower of a book is the last to have executed a `Lend` or `Take`, and that it becomes undefined when a book is returned, than to describe all preconditions and all modifications to attributes when a `Lend` occurs. Of course, there are cases in which an event is naturally seen by the user as a set of effects. For instance, a year closing transaction in an accounting system is easier to describe as a set of effects on various accounts, journals, and year-end reports.

5.2 Validation of the specification

We define validation as the activity of ensuring that the specification is an adequate formulation of the (textual) user requirements [5]. In other words, validation makes sure that the specification meets the client's expectations. Validation is usually conducted by human inspection and sometimes supported by specification animation tools to execute some scenarios.

In the EB^3 and B specifications of Sects. 3 and 4, some parts of the user requirements from Sect. 2 are not satisfied. They both contain the following *errors*:

1. A member can borrow a book that has been reserved by another member.
2. A member can renew a loan even if the book has been reserved by another member.
3. The borrower can reserve the book he borrowed.
4. A book can also be reserved without being lent or reserved by someone else.
5. A member can exceed his loan limit.
6. The action `Take` does not revoke its corresponding reservation.

Specification errors can be classified in several ways. From a state-based viewpoint, they can be classified on the basis of where they occur in the specification, i.e., in the precondition or in the postcondition of an operation. From a process algebraic viewpoint, notions of preconditions and postconditions translate into a single concept, namely event ordering.

Errors can also be classified on the basis of properties about the specification. In the state-based paradigm, an invariant describes a property on state variables that must be preserved by each transition (operation call). It can be verified by proofs, as we shall see in Sect. 5.3. A temporal property relates several transitions (i.e., event sequences) and can be verified by model checking. In the process algebraic paradigm, only temporal properties make sense.

In IS, data integrity is a fundamental requirement related to both invariants and temporal properties, because the state must properly reflect the history of what has happened and must not enable incorrect future events. Invariants are also called *static constraints*; temporal properties are also called *dynamic constraints*. Data integrity is often identified with static constraints, as these were studied first in database theory.

Errors 1, 2, and 4 are precondition problems. They are not detected by proving an invariant, but they can be detected by verifying temporal properties. Errors 3 and 5 are also precondition errors, but they can be detected by proving an invariant. Error 6 is a postcondition problem; it can be detected by proving an invariant. Of course, to detect these errors, one must find the appropriate invariants or temporal properties, which is not a trivial task.

In this section, we want to analyze and rectify such errors both in EB^3 and B. Moreover we will compare again these languages and try to identify a reasoning style which will help an analyst in this task.

5.2.1 Error correction in EB^3

These errors can be detected by an experienced EB^3 specifier through a review or walk-through of the specification. The ordering constraints on `Lend`, `Renew`, and `Reserve` are expressed in a simple manner; the only potential difficulty to understanding them lies in the synchronization between `loan` and `reservation` over the action `Take` or in understanding the quantifications occurring in `book` or `member`.

The first four errors arise from the difficulty in a process algebra to express constraints involving several entities at the same time. Such constraints occur quite often in IS. For instance, to prevent the borrowing of a reserved book, the process expression `loan` must be "aware" that

```
loan(mId : MEMBERID, bId : BOOKID) =
      ( isNotReserved(trace,bId) ==> Lend(mId,bId) | isFirst(trace,mId,bId) ==> Take(mId,bId) )
    .   isNotReserved(trace,bId) ==> Renew(mId,bId) )^*
    .   Return(mId,bId)

isNotReserved(trace : VALID_TRACE, bId : BOOKID): BOOLEAN =
    reservationQueue(trace,bId) = []
```

Fig. 11. EB^3 specification: State-oriented solution to errors 1 and 2

```
main =    ( ||| bId : BOOKID : book(bId)^* )
       || ( ||| mId : MEMBERID : member(mId)^* )
       || DisplayBorrowerByCategory()^*
       || Controller1()

Controller1() = ||| bId : BOOKID :
      |[ Reserve, Take, Cancel ]| mId : MEMBERID :
           ( (Lend(mId,bId) | Renew(mId,bId))^* . (||| mId2 : MEMBERID : reservation(mId2,bId)^*) )^*
```

Fig. 12. EB^3 specification: Process-algebraic solution to errors 1 and 2

a Reserve has been executed on the book. In a pure process algebraic style, a process can communicate with another solely through synchronization, which is not always easy to achieve. To facilitate this task, EB^3 allows for the use of a single state variable, the system trace, in a process expression.

The first two errors contradict requirement 4 in Sect. 2. The actions causing the errors are Lend and Renew; Take is not a problem, since it is guarded with the function isFirst. We can provide two equivalent solutions for these two errors: one is in a pure process algebraic style; the other uses a guard and a function. As we already mentioned, the use of guards and functions is more state oriented; we try to avoid it as much as possible to make ordering constraints more explicit. Figure 11 provides the state-oriented solution, while Fig. 12 provides the purely process algebraic one.

It is interesting to look at the analysis that led to the creation of the process controller1 in Fig. 12. The problem was to write a process expression to prevent the borrowing or renewal of a reserved book. It seems natural to state this in the following terms: *"a reserved book cannot be lent out and, if on loan, the book cannot be renewed."* This formulation cannot, however, be easily translated into a pure process algebraic style. It is more suitable for constructing a guard, which would simply express the negation of this formulation. To proceed with a pure process algebraic style, one has to reason in terms of what event sequences are *allowed*, not in terms of what is *prohibited.* So it is better to find a positive formulation of the statement: *"a loan and a renewal only occur before any reservation or after* ***all*** *reservations have been consumed"*. This reveals the need to "spy" on the actions the other members, which is achieved by the process Controller1 using an interleave quantification ||| mId2 for each member mId. These quantifications are all wrapped in a parameterized parallel composition |[Reserve,Take,Cancel]| mId, which requires synchronization on Reserve, Take, and Cancel.

Errors 3 and 4 are both related to the Reserve action and contradict requirement 3. If we correct them with a guard, the modification is quite straightforward. Figure 13 provides the guard function and the modification to the process reservation. One may be surprised to see that we do not check if the reservation queue already contains the member. The current specification already ensures that a member cannot perform two consecutive reserve actions, just by using the classic operators of process algebra to express a basic reservation scenario.

It is also possible to correct these errors in a pure process algebraic style as we did with errors 1 and 2. Again, we first need to formulate the requirements in terms of which event sequences are allowed. Therefore, the following formulations of requirement 3 are preferred:

- For the third error, *"For a given member and book, the reservations always occur between loans."*.
- For the fourth, *"A book reservation occurs during another member's loan or reservation cycle."*

Figure 14 provides the Controller2 and Controller3 processes that, respectively, specify these two statements.

```
canBeReserved(trace : VALID_TRACE, mId : MEMBERID,
              bId : BOOKID): BOOLEAN =
     (   currentBorrower(trace,bId) /= nil
     and currentBorrower(trace,bId) /= mId
     )
 or  reservationQueue(trace,bId) /= []

reservation( mId : MEMBERID , bId : BOOKID ) =
        canBeReserved(trace,mId,bId)
    ==> Reserve(mId,bId)
    .
    (   isFirst(trace,mId,bId) ==> Take(mId,bId)
    |   Cancel(mId,bId)
    )
```

Fig. 13. EB^3 specification: State-oriented solution to errors 3 and 4

```
Controller2() = ||| bId : BOOKID : ||| mId : MEMBERID :
                      ( loan(mId,bId)^* . Reserve(mId,bId)^* )^*

Controller3() =
    ||| bId : BOOKID :
        |[ Lend, Reserve, Take, Cancel, Return ]| mId : MEMBERID :   /* part 0 */
                (   (||| mId2 : MEMBERID - {mId} :
                        ( (Lend(mId2,bId) | Take(mId2,bId)).
                          Reserve(mId,bId)^* .
                          Return(mId2,bId)
                        )^*)                                         /* part 1 */
                |[Take]|
                    (||| mId3 : MEMBERID - {mId} :
                        ( Reserve(mId3,bId) .
                          Reserve(mId,bId)^* .
                          (Take(mId3,bId) | Cancel(mId3,bId))
                        )^*)                                         /* part 2 */
                )
            |||
                ( Lend(mId,bId) | Take(mId,bId) |
                  Return(mId,bId) | Cancel(mId,bId) )^*              /* part 3 */
```

Fig. 14. EB^3 specification: Process algebraic solution to correct errors 3 and 4

They must be inserted in the `main` process in parallel with other processes.

Clearly, these process expressions are quite hard to understand, even for experienced EB^3 specifiers. Indeed, if `Controller2` is still understandable, `Controller3` is clearly quite complex. It involves two different spying processes (`||| mId2` and `||| mId3`), called part 1 and part 2, respectively. Part 1 ensures that a book is reserved during a loan. Part 2 ensures that a reservation occurs only when at least one reservation by another member is active. The use of the synchronization operator `|[Take]|` between these two processes acts somewhat like a disjunction operator: a book can be reserved if and only if it has been borrowed *or* reserved by another member. Part 3 is needed to avoid deadlocks with the rest of the `Library` specification, since the constraint expressed should only apply to the `Reserve` action of a given book `bId`. Hence, part 3 allows other loan and reservation actions of `bId` to be executed in any order.

Process `Controller3` is very complex (in comparison with `Controller2` and even `Controller1`), because the constraint involves the same action, `Reserve`, from different members, which can be initiated in two cases: during a loan or reservation by another member. The use of guards and functions seems definitely wiser (and safer) here.

Figure 15 describes a partial view of the execution of a sequence of actions in `Controller3`. It shows the execution threads of three members (`mId`=1, 2, and 3 in part 0) during several loans and reservations for `bId`=10 in the outermost interleave `||| bId`. A blue box denotes an execution in part 3. A blue triangle in the upper right corner is an execution in part 1 and a blue triangle in the upper left corner is an execution in part 2. A box with two upper blue triangles represents an execution of both parts 1 and 2: this is only possible for a `Take` action, due to the synchronization `|[Take]|` between these parts. Here is the sequence that concerns us:

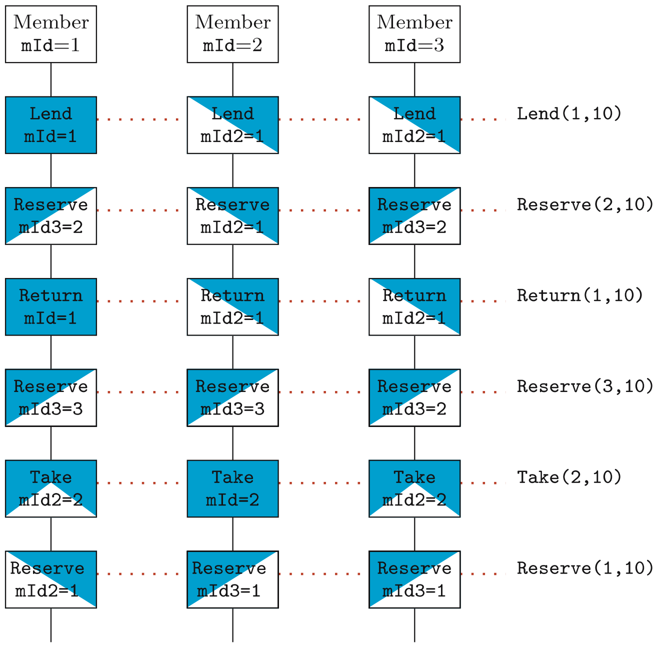


Fig. 15. Partial thread of execution for `controller3`

1. The first member borrows the book (`Lend(1,10)`). Its thread executes this action in part 3. Due to the synchronization on `Lend` imposed in part 0, the threads of the second and third members must also execute this action in part 1 with `mId2=1`.
2. The second member reserves the book. This is allowed because `Reserve(2,10)` is executable in part 1 of its thread (with `mId2=1`). The threads of the first and third members execute this action in part 2 with `mId3=2`.

3. The first member returns the book (`Returns(1,10)`).
4. The third member reserves the book (`Reserve(3, 10)`). This is allowed, even if the book has not been borrowed by anyone, because the second member has a reservation. So, this action is executed in part 2 in each thread with `mId3=2` for the third member and `mId3=3` for the first and second members.
5. The second member takes the book (`Take(2,10)`). In the threads of the first and the third members, the action is executed in part 1 and part 2, since these two parts must synchronize on a `Take` in order to close a reservation cycle and start a loan cycle.
6. The first member reserves the book (`Reserve(1,10)`). Since the book has already been reserved by member 3 and borrowed by member 2, this action can (non-deterministically) be executed in either part 1 (with `mId2=2`) or part 2 (with `mId3=3`). The threads of members 2 and 3 execute this action in part 2.

Error 5 is caused by the absence of a guard for actions `Lend` and `Take`. This can be corrected by adding the following guard in the process `loan`:

```
numberOfLoans(trace,mId) < maxNbLoan ==>
  (  isNotReserved(trace,bId) ==> Lend(mId,bId)
   | isFirst(trace,mId,bId) ==> Take(mId,bId))
  ...
```

Error 6 is caused by an improper update of the attribute `reservationQueue` in its defining function. This can be corrected by adding a line in the recursive function that computes the value of the reservation queue. It must suppress the member `mId` from the reservation queue when a `Take` action is met (Fig. 16).

We have provided two kinds of solutions for several errors (1, 2, 3 and 4): a state-oriented and an event-oriented. For error 5, however, the event-oriented solution is so cumbersome that we have only provided a state-oriented solution. Error 6 is also solved this way. For the sake of clarity and maintainability, we suggest rearranging the process expressions of entities and associations in order to separate the general ordering behavior from specific constraints. In our example, only the `loan` and `reservation` associations are submitted to constraints, so that we obtain the new EB^3 specification of the library provided in Fig. 17. Here is a summary if its structure:

– The `main` part and the `book` and `member` entities are unchanged.
– Each of the `loan` and `reservation` processes are split into two processes, a nominal and a controller that synchronize on the common events.
– The two nominal processes `loanNominal` and `reservationNominal` are specified by a pure event process expression that describes the general behavior of the association.
– The two controllers `loanController` and `reservationController` include guards related to the constraints on the `loan` and the `reservation` associations.

The recursive functions are created and corrected as indicated in this section. We do not provide their new version since there is no major modification.

5.2.2 Error correction in B

Errors related to ordering constraints are detected differently in B. It requires checking preconditions of the operations to determine if they appropriately describe the desired ordering properties. This requires a sound understanding of the state variables. Moreover, a good understanding of basic set theory, functions, and relations is also necessary to express some constraints in preconditions and to compare them with the user requirements. It can be quite difficult to get a clear view of the possible execution order of operations. One solution is to use a specification animator, which is provided by some case tools supporting B.

Once an error is located, the correction process is nearly the same as for EB^3 with guards and functions. As mentioned, it is natural to write constraints as implications, making them easily translatable into preconditions. Nonetheless, as is shown in the comparison in the Sect. 5.1.3, some constraints expressed with preconditions may seem quite artificial in B whenever it is easy to express them in EB^3. This is the new definition of the operation `L_Renew` with the correct precondition.

```
L_Renew($mId, bId$) $\triangleq$
PRE $mId \in ran(memberId)$ $\wedge$
  $bId \in ran(bookId)$ $\wedge$
  $(theBook, theMember) \in loans$ $\wedge$
  $theBook \notin dom(reservations)$ **error** 2
THEN
  $skip$
END
```

Operations `B_Reserve` and `B_Lend` were promoted from *B_Reservation* and *B_Loan* in the previous specifica-

```
reservationQueue(trace : VALID_TRACE, bId : BOOKID): LIST of MEMBERID =
   match last(trace) with
        nil -> []                                                    |
        Reserve(mId,bId) -> reservationQueue(front(trace),bId)::mId    |
        Take(mId,bId) -> reservationQueue(front(trace),bId) - {mId}    |
        Cancel(mId,bId) -> reservationQueue(front(trace),bId) - {mId} |
        _ -> reservationQueue(front(trace),bId)
```

Fig. 16. Correction of `reservationQueue` (error 6)

```
main =    ( ||| bId : BOOKID : book(bId)^* )
       || ( ||| mId : MEMBERID : member(mId)^* )
       || DisplayBorrowerByCategory()^*

book(bId : BOOKID) =     Acquire(bId,_)
                       . (   ( | mId : MEMBERID : loan(mId,bId) )^*
                         ||  ( ||| mId : MEMBERID : reservation(mId,bId)^* )
                         ||  DisplayCurrentBorrower(bId)^*
                         )
                       . Discard(bId)

member(mId : MEMBERID) =     Join(mId).
                           . (
                                 ( ||| bId : BOOKID : loan(mId,bId) )^*
                             ||  ( ||| bId : BOOKID : reservation(mId,bId)^* )
                             ||  DisplayNumberOfLoans(mId)^*
                             )
                           . Leave(mId)

loan(mId : MEMBERID, bId : BOOKID) =
            loanNominal(mId,bId) ||  loanController(mId,bId)^*

loanNominal(mId : MEMBERID, bId : BOOKID) =
            ( Lend(mId,bId) | Take(mId,bId) ) . Renew(mId,bId)^* . Return(mId,bId)

loanController(mId : MEMBERID, bId : BOOKID) =
                    numberOfLoans(trace, mId) < maxNbLoan
                and isNotReserved(trace, bId) ==>  Lend(mId,bId)
            |   isNotReserved(trace, bId) ==> Renew(mId,bId)
            |   numberOfLoans(trace, mId) < maxNbLoan ==> Take(mId,bId)

reservation(mId : MEMBERID, bId : BOOKID) =
            reservationNominal(mId,bId) || reservationController(mId,bId)^*

reservationNominal(mId : MEMBERID, bId : BOOKID) =
            Reserve(mId,bId) . ( Take(mId,bId) | Cancel(mId,bId) )

reservationController(mId : MEMBERID, bId : BOOKID) =
                canBeReserved(trace,mId,bId) ==> Reserve(mId,bId)
            |   isFirst(trace,mId,bId) ) ==> Take(mId,bId)
```

Fig. 17. EB^3 specification: Corrected process definitions

tion. To correct them, we must now refer to variables from each. This would imply a circular USES relationship between *B_Reservation* and *B_Loan*, which is not allowed in B; we must therefore create new operations in *B_Library*.

```
L_Reserve$(mId, bId) \triangleq$
PRE $mId \in ran(memberId) \wedge$
   $bId \in ran(bookId) \wedge$
   $(theBook, theMember) \notin reservations \wedge$
   $\big( theBook \in dom(reservations) \vee$
    $( theBook \in dom(loans) \wedge$
     $loans(theBook) \neq theMember$  \* error 3 *\
    $) \big)$ \* error 4*\
THEN
   $B\_Reserve(mId, bId)$
END;

L_Lend$(mId, bId) \triangleq$
PRE $mId \in ran(memberId) \wedge$
   $bId \in ran(bookId) \wedge$
   $theBook \notin dom(loans) \wedge$
   $theBook \notin dom(reservations) \wedge$  \*error 1*\
   $card(loans^{-1}[\{theMember\}]) <$
         maxNbLoan \*error 5*\
THEN
   $B\_Lend(mId, bId)$
END;
```

Error 3 is also an ordering constraint that is corrected by adding the conjunct:

$$loans(theBook) \neq theMember \tag{2}$$

to the precondition of operation L_Reserve. Moreover, we can also strengthen the invariant of the machine *B_Library* by adding the conjunct:

$$loans \cap reservations = \emptyset \tag{3}$$

```
L_Take($mId, bId$) $\triangleq$
PRE  $mId \in ran(memberId) \ \wedge \ bId \in ran(bookId) \ \wedge$
  $theBook \notin dom(loans) \ \wedge \ (theBook, theMember) \in reservations \ \wedge$
  $numReserv(theBook, theMember) = min(numReserv[\{theBook\} \lhd dom(numReserv)]) \ \wedge$
  $card(loans^{-1}[\{theMember\}]) <$ maxNbLoan /*error 5*/
THEN
  $B\_Lend(mId, bId) \ \| \ B\_Cancel(mId, bId)$ /*error 6*/
END;
```

Fig. 18. Correction of operation L_Take (error 6)

Error 3 could therefore be detected, because proof obligations of the operation L_Reserve would fail if the conjunct (2) were missing in its precondition.

Invariant (3) also allows error 6 to be detected. Indeed, proof obligations of the operation L_Take will fail and we must add the substitution that deletes the pair of objects of the variable reservations that is added in the variable loans by the operation B_Lend. Figure 18 shows the new operation L_Take.

Error 5 comes from the requirement 9 that expresses an integrity constraint, which requires the definition of an additional invariant:

$$\forall m \cdot m \in members \Rightarrow card(loans^{-1}[\{m\}]) \leq \texttt{maxNbLoan} \tag{4}$$

in the machine *B_Loan* and a new constant maxNbLoan. To discharge the proof obligations associated with this integrity constraint, the following precondition has been added to the operation B_Lend, and then to L_Take and L_Lend:

$$card(loans^{-1}[\{theMember\}]) < \texttt{maxNbLoan} \ . \tag{5}$$

Of course, all these modifications induce modifications in the corresponding operations of the *B_Library_Interface* machine.

5.2.3 Conclusion

For precondition errors, specification validation against user requirements is easier to achieve in event-oriented EB^3 than in state-oriented B for two reasons. First, the understanding of an EB^3 specification can be local, that is, each process expression can be understood independently of the others and thus can be individually validated. Second, user requirements are expressed more naturally with a process algebra, since it streamlines the specification of ordering constraints, at least, when the constraints do not involve many entities or associations. Indeed, the use of guards in an EB^3 specification tends to blur the global readability of the specification as preconditions do in a B specification. As an example, consider the following process expression:

```
    a.b.c
```

written in a pure process algebra style. It can be expressed in a guard-oriented style in EB^3 as:

```
  (g1 ==> a) || (g2 ==> b) || (g3 ==> c)
```

where the guard g1 expresses that action a is the first action to be performed and the guard g2 expresses (respectively g3) that action a (resp. actions a and b) has already been performed. The ordering constraint between actions a, b, and c is explicitly formulated in the first expression whereas the guard definitions must be explored to discover the constraint in the second expression.

It seems, however, that ordering constraints involving several properties of several entities (e.g., the last error correction) are quite difficult to express in EB^3 without guard, and are definitely less readable than an equivalent guard-oriented solution. In this case, the guard-oriented style is the most natural and easiest to write and understand. EB^3 has a slight advantage over B in this case, because a data attribute is completely defined by a single function, which makes it easier to understand. Nevertheless, this study has shown that some constraints look more natural in one paradigm while others are more natural in the other.

Some postcondition errors are not easier to detect in either method. For instance, error 6 is as difficult to find in B as EB^3, because this kind of error is intrinsically related to the state-based paradigm.

5.3 Specification verification

We define verification as the activity of checking that a specification satisfies some properties stated in a formal language. A property can be checked either by proving it or by checking it on a finite model of the specification. In contrast, validation is conducted using informal requirements or by the user [5].

5.3.1 Verification in EB^3

Since EB^3 is founded on a process algebra, an EB^3 specification could be verified using model-checking techniques. Theorem proving is seldom used for the verification of process expressions. There is currently no model-checking tool for EB^3. Bridges could be defined, however, to tools developed for other process algebras (e.g., FDR [16] for CSP, CADP [23] for LOTOS).

Model checking is based on exploration of a model of the specification. A process expression must first be translated into a transition system (the model). This transition system is then explored in order to verify temporal properties or to compare two process expressions for equivalence or refinement. The verification is usually automatic, but it can suffer from combinatorial explosion. For IS specifications, this is particularly severe since unbounded quantifications are used to represent entities and associations. Hence, the transition system of an IS is usually unbounded. For practical verification, the model must be reduced (e.g., by considering a small number of books and members) to check a property. When a property is satisfied on a reduced model, there is no guarantee that it is satisfied in the unbounded model. Dually, if a property fails on a reduced model, then it is usually not satisfied in the unbounded model, because the verification has identified a counterexample. Nevertheless, one could imagine properties that fail on a reduced model but succeed in an larger one (e.g., stating that a member can have up to n active loans). Recent developments in model checking allow for verification of unbounded models using a reduced model, in some specific cases (e.g., [13]). These techniques, however, sometimes require the identification of an invariant that must be proved for the specification. Consequently, the procedure is no longer automatic; it requires creativity and the use of a theorem prover. For an example of temporal property verification in process algebraic specifications of IS, the reader is referred to [14].

Expression and verification of static constraints are more complex in EB^3 than in B. One approach is to show that, for each integrity constraint `C`, $\mathcal{T}(\texttt{main||C}) = \mathcal{T}(\texttt{main})$, where `main` is the main process expression and `C` is the process expression defining the integrity constraint. For example, error 5 could be detected by using the following constraint:

```
C = ||| mId : MEMBERID :
          numberOfLoans(trace,mId) < maxNbLoan
      ==> (Lend(mId,_) | Take(mId,_))^*
```

To satisfy this constraint, the actions `Lend` and `Take` in the process `loan` must be guarded by:

```
      numberOfLoans(trace,mId) < maxNbLoan
```

Combinatorial explosion may prevent a model checker from detecting the violation of this condition, depending on the value assigned to `maxNbLoan`.

Dynamic constraints can also be specified using trace inclusion. For instance, a *safety* constraint C can be written such that $\mathcal{T}(main) \subseteq \mathcal{T}(C)$, which shows that something bad (i.e., a trace not allowed by C) cannot happen. Dually, a *liveness* property C may be stated such that $\mathcal{T}(main) \supseteq \mathcal{T}(C)$, which shows that something good (i.e., a trace allowed by C) can happen.

Temporal logic could also be used for dynamic constraints [29]. For instance, the following formula states that it is always the case ($\Box$) that, when a Lend occurs, the next events ($\bigcirc$) must not be ($\neg$) another Lend unless ($\mathcal{W}$) a Return occurs before.

$$\forall bId \cdot \Box(\mathsf{Lend}(bId,\_) \Rightarrow \bigcirc(\neg\mathsf{Lend}(bId,\_)\ \mathcal{W}\ \mathsf{Return}(bId,\_)))$$

5.3.2 Verification in B

Case tools such as Atelier B [10] provide a prover to handle proof obligations associated with machines and their refinements. Among proof obligations, the preservation of the invariant by operations is essential for verifying properties. B is very nice for specifying static properties about the data structures. The Atelier B prover was able to automatically discharge all proof obligations associated with our B specifications.

In IS, there exist a great number of static constraints expressed on the state of the system, such as requirement 9 that can be easily specified with the B language in the invariant clause. Discharging proof obligations associated with a specification ensures that integrity constraints are satisfied by the specification. In order to discharge these proof obligations, suitable preconditions must be defined in the operations involved by a constraint.

Dynamic constraints are very difficult to express and verify in B, as explained above. A solution for checking such constraints is to use the refinement mechanism and to prove that the B specification is a refinement of the EB^3 specification, as presented in [19].

Another approach, presented by Darlot in [11], combines the Event B method with the temporal logic PLTL [33]. Event B allows an event system to be specified on which temporal properties are expressed. In addition, a verification method is proposed: invariant properties of the system can be checked by theorem proving, using Atelier B, whereas temporal properties are verified by model checking. The approach does require some adaptation for application to IS [21].

5.3.3 Conclusion

Static integrity constraints can be explicitly stated within the invariant of a B machine. They induce proof obligations that must be discharged by the specifier. When it is difficult to discharge them, it is usually a good sign that the invariant is not preserved by an operation; specification errors are then uncovered.

The same invariant property could be stated in EB^3 using functions defining the entity attributes. Proving that they are preserved by every operation consists in proving that they hold for any trace accepted by the main process. These proofs are more difficult to achieve in EB^3, because there is no explicit formulation of event preconditions. B has a definite advantage over EB^3 in this regard.

B and EB^3 currently do not offer any proper support for the specification and verification of dynamic constraints. Current model-checking techniques could be applied to both, with the same level of difficulty and effectiveness, since verification is conducted on a transition system.

5.4 Specification evolution

We may classify modifications to specification as either event-ordering modification or data-requirements modification. The latter involves the definition or modification of data attributes, which may be used for defining ordering constraints or simply for providing information to the user.

Most of the errors identified in Sect. 5.2 were related to event-ordering constraints; their correction illustrates that changing the B specification and the EB^3 specification were roughly equivalent in complexity. These changes did not require the definition of new data attributes. The slight modification to the user requirements presented in Sect. 5.1.1 (to prohibit the reacquisition of a book) showed that the B specification required more changes than the EB^3 one. It is difficult to generalize this principle to arbitrary event-ordering requirements modifications. Nevertheless, an event-ordering modification in EB^3 can be defined either by providing a new combination of operators or by defining new attributes and using them in a guard. In the first case, the modification is simpler than in B; in the second case, it is roughly equivalent to the modification in B.

Adding new data requirements, without changing the ordering constraints, usually involves more work in B than in EB^3. For instance, consider that we need to store the history of loans for a book, instead of only the current loan. In both B and EB^3, a new attribute must be created. In EB^3, it is quite straightforward: a new definition has to be created; there is no change to the rest of the specification. In the B specification, one must decide if the two attributes (current loan and history of loans) should be kept, or if the specification is rewritten to use only the history of loans as a variable. If two attributes are kept, several modifications are avoided. This solution introduces, however, some redundancy in the state space, along with the risk that future modifications to the specification will introduce some inconsistency between them. Stating the relationship between the two redundant attributes in an invariant avoids that, but it induces additional work for discharging invariant preservation proof obligations.

6 Conclusion

We have studied two orthogonal specifications method for IS. The B method is illustrative of the state-based paradigm, whereas the EB^3 method is illustrative of both the event-based and the state-based paradigms, since it is an hybrid method.

The EB^3 method is based on a separation of concerns between input processing and output processing, which distinguishes it from traditional process algebraic methods and state-oriented methods. Its process algebraic facet fosters the explicit definition of intra-entity constraints, making them easy to review and understand for a human being. Complex inter-entity constraints involving the history of input events are better expressed through its state-oriented facets, using guards and recursive functions on the system trace. The price paid for the hybrid nature of EB^3 is the strong difficulty of proving the preservation of static data integrity constraints, because the process algebraic facet does not include an explicit definition of action preconditions and postconditions.

The B method fosters modularity through machine encapsulation and operation encapsulation. It is very powerful for expressing complex ordering constraints and proving the preservation of static data integrity constraints. The price paid for this powerful feature is the difficulty of understanding, for a human being, the ordering relationship between input events, because of the strong data coupling between operations.

Dynamic constraints are as hard to verify in both methods. The ease of understanding and the weak coupling between actions makes EB^3 specifications slightly easier to maintain than B specifications.

One way of exploiting the strengths of each method is to use EB^3 to provide an explicit definition of the user requirements and their dynamic constraints and to use B to provide a powerful mechanism for specifying and verifying static data integrity constraints. Moreover, B can be used as a design method to proceed from a requirements specification to a complete implementation of the system [26, 28]. The consistency between the EB^3 specification and the B specification can be checked by proving refinement between them, using B's refinement relation. This refinement proof would ensure that the B specification satisfies all the ordering constraints prescribed by the EB^3 specification. It constitutes an interesting alternative to model checking, which is often limited by combinatorial explosion. A strategy for such a refinement proof has been proposed in [19]. Unfortunately, this refinement proof is not a trivial task. The idea of combining the state-based paradigm and the event-based paradigm has been studied under several forms: CSP OZ [15], Circus [37], CSP2B [9], and CSP $\|$ B [35].

Our future work will address the refinement of an EB^3 specification into a B specification, on the basis of the strategy outlined in [19]. We wish to automate as much as possible the generation of the B specification from the EB^3 one. It should be possible to derive from the EB^3 specification the variables, the gluing invariant and the postconditions of the operations. The most difficult part is to generate operation preconditions from process expressions using the variables corresponding to entity attributes. We will first investigate the EB^3 patterns identified in [20].

Acknowledgements. The authors wish to thank the anonymous referees for their insightful comments and suggestions, which lead to significant improvements to the paper. The work of Marc Frappier is supported by NSERC (Natural Sciences and Engineering Research Council) of Canada and by FQRNT (Fonds québécois de la recherche sur la nature et les technologies) of Québec.

References

1. Abrial J-R (1996) The B-Book. Cambridge University Press, Cambridge, UK
2. Abrial J-R (1996) Extending B without Changing it. In: Habrias H (ed) First Conference on the B Method, pp 169–190, November 1996

3. Abrial J-R, Mussat L (1998) Introducing Dynamic Constraints in B. In: Bert D (ed) Second International B Conference, Lecture Notes in Computer Science, vol 1393. Springer-Verlag, pp 83–128, April 1998
4. Behm P, Benoit P, Faivre A, Meynadier JM (1999) Météor: A Successful Application of B in a Large Project. In: FM99: World Congress on Formal Methods, Toulouse, France, Lecture Notes in Computer Science, vol 1708. Springer-Verlag, pp 369–387, September 1999
5. Boehm BW (1984) Verifying and Validating Software Requirements and Design Specifications. IEEE Software 1(1):75–88, January 1984
6. Boerger E, Staerk R (2003) Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag, ISBN 3-540-00702-4
7. Bolognesi T, Brinksma E (1987) Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14(1):25–59
8. Butler MJ, Waldén M (1996) Distributed System Development in B. In: Habrias H (ed) First Conference on the B Method, November 1996
9. Butler M (2000) csp2B: A Practical Approach to Combining CSP and B. Formal Aspects of Computing 12(4):182–198
10. CLEARSY System Engineering: Aix-en-Provence, France, `http://www.clearsy.com/`
11. Darlot C (2002) Reformulation et vérification de propriétés temporelles dans le cadre du raffinement de systèmes d'événements. Ph.D. thesis, Université de Franche-Comté, France
12. Elmasri R, Navathe SB (2004) Fundamentals of Database Systems. 4th edition, Addison-Wesley
13. Emerson EA, Kahlon V (2000) Reducing model checking of the many to the few. In: Proceedings of CADE'2000, Lecture Notes in Computer Science, vol 1831. Springer-Verlag, pp 236–354
14. Evans N, Treharne H, Laleau R, Frappier M (2004) How to Verify Dynamic Properties of Information Systems. In: Cuellar JR, Liu Z (eds) 2nd IEEE International Conference on Software Engineering and Formal Methods, Beijing, China, 26–30 September 2004. IEEE Computer Society Press, pp 416–425.
15. Fischer C (2000) Combination and Implementation of Processes and Data: from CSP-OZ to Java. PhD thesis, University of Oldenburg
16. Formal Systems (Europe) Ltd. (1997) Failures-Divergences Refinement: FDR2 User Manual. `http://www.formal.demon.co.uk`
17. Fraikin B, Frappier M (2002) EB^{3}PAI: an interpreter for the EB^3 specification language. In: FM-TOOLS 2002, The 5th Workshop on Tools for System Design and Verification, Reisensburg Castle, Günzburg, Germany, 15–17 July 2002
18. Fraikin B, Frappier M (2002) Optimizing memory space in the EB^3 process algebra interpreter. In: ICCSSEA 2002, Software and Systemes Engineering and their Applications, vol I, Session 4
19. Frappier M, Laleau R (2003) Proving Event Ordering Properties for Information Systems. In: Zb 2003: Formal Specification and Development in Z and B, Turku, Finland, 4–6 June 2003, Lecture Notes in Computer Science, vol 2651. Springer-Verlag, pp 421–436
20. Frappier M, St-Denis R (2003) EB^3: an Entity-Based Black-Box Specification Method for Information Systems. Software and System Modeling 2(2):134–149, July 2003
21. Gervais F (2004) EB^4: Vers une méthode combinée de spécification formelle des systèmes d'information. Examen de spécialité, Doctorat Informatique, Université de Sherbrooke, June 2004
22. Hoare CAR (1985) Communicating Sequential Processes. Prentice Hall, Englewood Cliffs
23. INRIA Rhône-Alpes: CADP (Caesar/Aldebaran Development Package), `http://www.inrialpes.fr/vasy/cadp/`
24. Jarke M, Mylopoulos J, Schmidt JW, Vassiliou Y (1992) DAIDA: An Environment for Evolving Information Systems. ACM Transactions on Information Systems 10(1):1–50, January 1992
25. Laleau R Mammar A (2000) An Overview of a Method and its Support Tool for Generating B Specifications from UML Notations. In: ASE: 15th IEEE Conference on Automated Software Engineering, Grenoble, France, September 2000, IEEE Computer Society Press
26. Laleau R, Mammar A (2000) A Generic Process to Refine a B Specification into a Relational Database Implementation. In: ZB2000: Formal Specification and Development in Z and B, Lecture Notes in Computer Science, vol 1878, Springer-Verlag, York
27. Laleau R (2002) Conception et développement formels d'applications bases de données. Habilitation Thesis, CEDRIC Laboratory, Évry, France. Available at `http://cedric.cnam.fr/PUBLIS/RC424.ps.gz`
28. Mammar A (2002) Un environnement formel pour le développement d'applications bases de données. Ph.D. thesis, CEDRIC Laboratory, CNAM, Evry, France, November 2002. Available at `http://cedric.cnam.fr/PUBLIS/RC392.ps.gz`
29. Manna M, Pnueli A (1992) The temporal logic of reactive and concurrent systems. Springer-Verlag
30. Meyer E, Souquières J (1999) A Systematic approach to Transform OMT Diagrams to a B specification. In: Wing JM, Woodcook J, Davies J (eds) Formal Methods (FM'99), September 1999, Lecture Notes in Computer Science, vol 1708(1), Springer-Verlag, pp 875–895
31. Mills HD, Linger R.C., Hevner AR (1986) Principles of Information Systems Analysis and Design. Academic Press, Orlando, FL
32. Milner R (1989) Communication and Concurrency. Prentice Hall, Englewood Cliffs
33. Pnueli A (1981) The temporal semantics of concurrent programs. Theoretical Computer Science 13:45–60
34. Snook C, Butler M (2004) UML-B: Formal modelling and design aided by UML. Technical Report, Department of Electronics and Computer Science, University of Southampton, United Kingdom. `http://www.ecs.soton.ac.uk/people/mjb/`
35. Treharne H, Schneider S (2000) How to drive a B machine. In: Bowen JP, Dunne S, Galloway A, King S (eds) ZB2000: Formal Specification and Development in Z and B, LNCS vol 1878, Springer-Verlag, pp 188–208
36. Vissers CA, Scollo G, van Sinderen M (1988) Architecture and specification style in formal descriptions of distributed systems. In: Aggarwal S, Sabnani K (eds) Protocol Specification, Testing and Verification, VIII, North-Holland, Amsterdam, pp 189–204
37. Woodcock JCP, Cavalcanti ALC (2002) The Semantics of Circus. In: ZB 2002: Formal Specification and Development in Z and B, Grenoble, France, 2002, LNCS vol 2272. Springer-Verlag

Benoit Fraikin is a Ph.D. student in computer science at the Université de Sherbrooke. He graduated in mathematics and obtained a DEA in logic and computer science, both at Université Denis Diderot (Paris VII). During his DEA, he worked on the B method at Matra transport. His Ph.D. thesis concerns the development of an efficient interpreter for the EB^3 process algebra.

Marc Frappier is a professor of software engineering at the Université de Sherbrooke. He earned a Ph.D. in computer science from the University of Ottawa in 1995. His research interests include software specification and synthesis, software measurement, and project management. He held several positions in industry prior to his academic career, both at the technical and management levels.

Régine Laleau is professor at the french University Paris 12 and member of the LACL research laboratory since 2003. Before she was member of the CEDRIC-CNAM laboratory where she has obtained an habilitation thesis in 2002. Her research domain concerns the use of formal methods for the analysis and design of databases applications.