A Comparison of EB^3 and B for Information System Specification

Benoît Fraikin, Marc Frappier

GRIL
Département de mathématiques et d'informatique
Université de Sherbrooke
Sherbrooke, Québec, Canada J1K 2R1
+1 819 821-8000x2096
{Fraikin,Frappier}@dmi.usherb.ca

Régine Laleau

Laboratoire CEDRIC
Institut d'Informatique d'Entreprise
Conservatoire National des Arts et Métiers
18 allée Jean Rostand, 91025 Évry Cedex France
+33 1 69 36 73 47
laleau@iie.cnam.fr

July 4, 2003

1 Introduction

The goal of this paper is to compare state-based and event-based specification paradigms of information systems (IS) for a number of view points. The first and most important one is the ease of describing the *functional behavior* of IS. That includes expressing event ordering, data structures, modularity, and enforcing integrity constraints. The second view point is specification validation, which includes checking the specification against the user requirements by either review, inspection, walkthrough, animation, or scenario analysis. The next view point is specification validation, which consists of 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 view point, we identify the relative strengths and the weaknesses of each paradigm. Ultimately, this comparison leads us to determine an *ideal* specification process which includes the best of both paradigms.

Our comparison is domain specific and restricted to information systems. Their main characteristics are complex event ordering constraints and several complex data structure with multiples relationships. Concurrency, distribution and real-time constraints are usually not an issue for the formal definition of IS user requirements; although they may be of concern at the design level, but we do not address design issues in this paper.

In the past, the reliability requirements for information systems where not as strong, as they were used in-house only. Faults did not usually have a significant impact on an organisation. With the rapid deployment of the world wide web, organisations are developing web access to their information systems, thereby increasing the need for high quality systems. Data integrity becomes a critical issue if ordinary clients can use an organisation's information systems.

To illustrate the state-based specification paradigm, we have chosen the B language [1]; for the event-based paradigm, we have selected the EB^3 language [13]. B has been shown to be appropriate for describing IS [15, 16]. It supports the whole life cycle, from requirements specification to implementation; it is supported by industrial-strength case tools which have been successfully used on large scale, safety critical industrial applications [4]. EB^3 has been defined for the purpose of specifying IS; it is based on entities, process algebra, traces, and recursive functions defined on traces. It is chiefly event-driven, but it also includes some state-oriented constructs, in order to facilitate IS specification. A purely event-based specification of IS (*e.g.,* with a classical process algebra like CSP) is not practical, because it is too cumbersome to express data structures and complex ordering constraints.

Our comparison is illustrated by specifying 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 provides an equivalent specification in the B language. Note

that in order to highlight some features of each language, we have deliberately inserted some *errors* in both specifications. The correction of these errors are addressed in Section 5, which compares the two specifications from the view points of functional behavior description, validation, verification, and evolution. We conclude in Section 6 by providing a summary of the relative strengths and weaknesses of each language, from which an ideal specification process is derived. We also provide summary answers to the questions raised in the workshop call for papers.
Warning: We have attempted to transform an extended abstract into a full paper, but we were short of time to properly polish it. Hence, this paper is rather in a state of work in progress. We apologize for any inconvenience it may cause to the reader.

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. 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. Requirements elements are numbered for further reference in the sequel.

1. The library system has to manage loans and reservations of books by members.
2. A book is acquired by the library. It can be discarded, but only if it is not lent and not reserved.
3. A member must join the library in order to borrow a book.
4. A book can be reserved if and only if it is lent or already reserved by another member.
5. A member can borrow or renew a book loan unless the book is already reserved.
6. If many members reserved a book, the first one who reserved it is allowed to take it when it is returned, unless this member decided to cancel his reservation.
7. Anyone who reserved a book can cancel the reservation at anytime before the reservation is used.
8. A member can leave the library membership only when all his loans are returned and all his reservations are either used or canceled.

3 The EB^3 Specification

3.1 An Overview of EB^3

The EB^3 language has been especially design to specify IS. The core of EB^3 [13] includes a specification process and a formal notation for progressing from use cases to a complete and precise specification of input-output traces. Under EB^3, the specifier performs various tasks summarized as follows:

1. Define a user requirements class diagram.
2. Declare actions for entity types and associations.
3. Define valid input traces using a process expression.
4. Write recursive functions on input traces that assign values to entity attributes.
5. Define an output for each input trace using input-output rules.

The EB^3 language uses a process algebra similar to Lotos [5], CCS [17] or CSP [14]. It includes sequence (`.`), choice (`|`), Kleene closure (`^*`), interleaving (`|||`), parallel composition (`||`, *i.e.,* synchronization on shared actions), guard (`==>`), process call, and quantification of choice (`|x : T : ...`) and interleaving (`|||x : T : ...`). An action has input and output parameters; an input event is an action with its input parameters. Input events are elementary process expressions. They can be freely combined using the operators. A single *state variable* can be used in process expressions: the execution trace. It is usually accessed through recursive functions that extract relevant information from it.

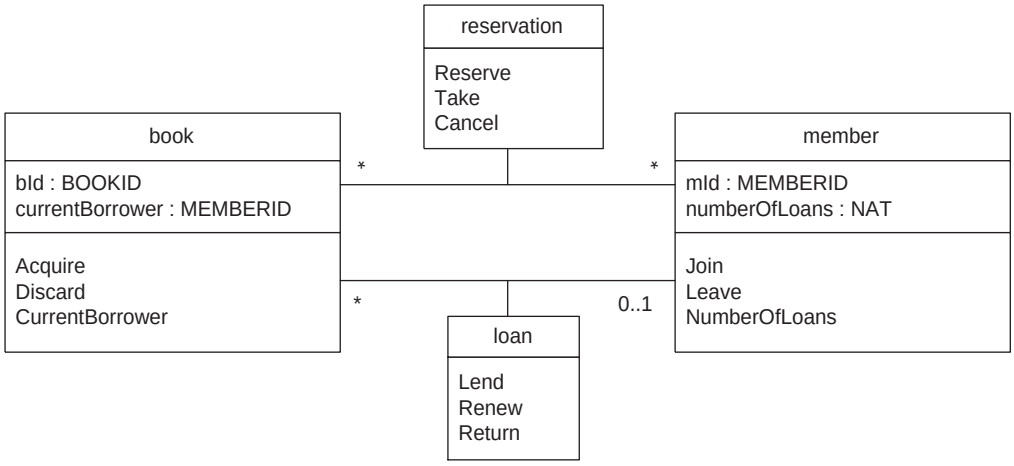

Figure 1: User requirements class diagram of the library IS

An ongoing project at the University of Sherbrooke aims at automatically generating IS from EB^3 specification. Details on this topic can be found in [10, 11]. The process algebra interpreter developed so far can execute process expressions with reasonable performance, such that it can be used in many cases as a substitute to a hand-written implementation of the specification.

3.2 The Library Specification

Figure 1 shows the user requirements class diagram used to construct the specification. By studying the requirements, we quickly find four process definitions that will represent the two entity types, member and book, and the two associations, reservation and loan. In [13], Frappier and St Denis describe a strategy to correctly and efficiently design an EB^3 specification. A more complete analysis of the specification creation process can be found in Section 5.1.1. By convention action names have uppercase initials and process calls and functions are all lowercase.

The Figure 2 describes the main process and the entity process definitions of `member` and `book`. The process `main` gives the global behavior of the system. All actions have no output, except for `CurrentBorrower(bId)` and `NumberOfLoans(mId)`. These last two actions output the result of recursive functions `currentBorrower(bId)` and `numberOfLoans(mId)`, respectively, applied to the current execution trace. These two functions are defined in Figure 3. The correspondance between actions and functions is defined by input-output rules, which are omitted here for the sake of concision. The Figure 2 also describes the association process definitions of `loan` and `reservation`. Expression `isFirst(trace,mId,bId)` is a function call returning a Boolean value. Description of this function and the one needed to compute it are the last two shown in Figure 3.

A complete description of the recursive functions and their uses can be found in [13]. A function is described by its parameters in parenthesis and their types. The output type value is written after the colons. The definition of a function is written in a CAML[1] style. We use a special symbol, `trace`, that denotes the

[1] CAML is a functional language.

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

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

member(mId : MEMBERID) =
     (Join(mId,_,_).
     (
         ( ||| bId : BOOKID : loan(mId,bId)^* )
       ||
         ( ||| bId : BOOKID : reservation(mId,bId)^* )
       ||
         NumberOfLoans(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)
    )
```

Figure 2: EB^3 specification: process definitions

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

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

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

Figure 3: EB^3 specification: function definitions

current execution trace; it is represented by a list. Operators `last` and `first` and `front` respectively return the last element, the first element and all but the first element of a list; they return the special value `nil` when the list is empty. Symbol `[]` denotes an empty list and symbol `::` is the append of an element to a list. Finally the symbol _ can match any value and is consequently used to provide default instructions.

The primary use of functions is to extract information from the execution trace. They are used to provide a response to some input events like the function `currentBorrower(bId)` which returns its output value when the action `CurrentBorrower(bId)` is executed. However they are also quite useful to handle constraints in the specification with the use of guard as with `isFirst` in process `reservation`. In this way the use of functions is quite similar to a precondition. More discussions about this subject are provided in Section 5.1.1.

4 The B Specification

We choose the B language to specify the IS with a state-based approach. B is a formal method developed by Abrial [1]. It is a complete method and supports a large segment of the development life cycle : specification, refinement and implementation. It ensures, thanks to refinement steps and proofs, that the code satisfies its specification. It has been used in significant industrial projects [4] and commercial case tools are available in order to help the specifier during the development process. These are the main arguments to use B rather than Z (type checking and tool assistance in proof, but limited tool support for refinement and no automated proof), VDM (standard semantics, good tool support, but relatively primitive structuring of

MACHINE B_Member

SETS $MEMBER$;

VARIABLES $members$

INVARIANT $members \subseteq MEMBER$

INITIALISATION $members := \{\}$

OPERATIONS

Join(mId) $\triangleq$
PRE $mId \in MEMBER - members \ \wedge$
$members \subset MEMBER$
THEN
$members := members \cup \{mId\}$
END;

B_Leave(mId) $\triangleq$
PRE $mId \in members$
THEN
$members := members - \{mId\}$
END

END

MACHINE B_Book

SETS $BOOK$;

VARIABLES $books$

INVARIANT $books \subseteq BOOK$

INITIALISATION $books := \{\}$

OPERATIONS

Acquire(bId) $\triangleq$
PRE $bId \in BOOK - books \ \wedge$
$books \subset BOOK$
THEN
$books := books \cup \{bId\}$
END;

B_Discard(bId) $\triangleq$
PRE $bId \in books$
THEN
$books := books - \{bId\}$
END

END

Figure 4: B specification: *B_Member* and *B_Book* machines

formal presentation). However, the approach described in this paper probably applies to these state-based languages, and others of the same family (such as ASM).

Each entity and association will be specify in separate machines. Figures 4, 5 and 6 respectively describe the `B_Member` and `B_Book` machines, `B_Reservation` and `B_Loan` machines. Figure 7 provides the top-level machine (or global machine) that works as the user interface. Operations available to users are either operations of this machine or promoted operations from included machines.

The `Renew` operation may look quite useless, since it only has a `skip` instruction in its body. A real IS would probably update the limit date of return, which we do not take into account, for the sake of simplicity. A more complete analysis of the specification creation process can be found in Section 5.1.1.

Note that we could have used the event B approach [1, 2, 3, 6], instead of EB^3, to specify the IS with an event-based style. However, event B doesn't provide explicit mechanisms to express event ordering constraints, which we want to model. Hence, an event B specification would be fairly similar to our B specification.

5 A Comparison of the Two Specifications

5.1 Expression of Functional Behavior

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

MACHINE $B\_Reservation$

USES B_Member, B_Book

VARIABLES $reservations$, $numReserv$;

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

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

OPERATIONS

Reserve$(mId, bId) \triangleq$
PRE $mId \in members \ \wedge \ bId \in books \ \wedge$
$(bId \mapsto mId) \notin reservations$
THEN
$reservations := reservations \cup \{bId \mapsto mId\} \ ||$
$numReserv := numReserv \cup \{(bId, mId) \mapsto (max(ran(numReserv) \cup \{0\}) + 1)\}$
END;

Cancel$(mId, bId) \triangleq$
PRE $mId \in members \ \wedge \ bId \in books \ \wedge$
$(bId \mapsto mId) \in reservations$
THEN
$reservations := reservations - \{bId \mapsto mId\} \ ||$
$numReserv := \{(bId \mapsto mId)\} \lhd numReserv$
END

END

Figure 5: B specification: $B\_Reservation$ machine

5.1.1 The EB^3 Specification

The structure of the EB^3 specification is directly inspired from the entities and their associations. This structure entails a nicely modularized specification. Simple ordering constraints (*i.e.,* basic scenarios) can be easily expressed using the process algebra operators. To do so, one can take each entity type from the requirements class diagram (*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 $\|$. The behavior of an association is also described by a process expression which is called by each entity. The multiplicity of an association (*e.g.,* 1..*,*) 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; $|||$ when an entity is related to a number of entities). Several patterns have been defined to translate requirements class diagram into process expressions (see [13]).

Following the structure of the class diagram allows for the elements 2, 3, 7, and 8 of the user requirements to be taken into account, as well as implicit requirements. For instance, the fact that two members cannot

MACHINE B_Loan

USES B_Member,
B_Book

VARIABLES
$loans$;

INVARIANT
$loans \in books \pfun members$

INITIALISATION $loans := \{\}$

OPERATIONS

Lend$(mId, bId) \triangleq$
PRE $mId \in members \ \wedge$
$bId \in books \ \wedge$
$bId \notin dom(loans)$
THEN
$loans(bId) := mId$
END;

Return$(mId, bId) \triangleq$
PRE $mId \in members \ \wedge$
$bId \in books \ \wedge$
$(bId, mId) \in loans$
THEN
$loans := \{bId\} \ndres loans$
END;

$mId \leftarrow$ **CurrentBorrower** $(bId) \triangleq$
PRE $bId \in dom(loans)$
THEN
$mId := loans(bId)$
END

END

Figure 6: B specification: B_Loan machine

MACHINE $B\_Library$

INCLUDES B_Member, B_Book, B_Loan, $B\_Reservation$

PROMOTES $Join$, $Acquire$, $Cancel$, $Reserve$, $Lend$, $Return$

OPERATIONS

Leave$(mId) \triangleq$
PRE $mId \in members \ \wedge$
$mId \notin ran(reservations) \ \wedge$
$mId \notin ran(loans)$
THEN
$B\_Leave(mId)$
END;

Discard$(bId) \triangleq$
PRE $bId \in books \ \wedge$
$bId \notin dom(reservations) \ \wedge$
$bId \notin dom(loans)$
THEN
$B\_Discard(bId)$
END;

Take$(mId, bId) \triangleq$
PRE $mId \in members \ \wedge$
$bId \in books \ \wedge$
$(bId, mId) \in reservations \ \wedge$
$numReserv(bId, mId) = min(numReserv[\{bId\} \lhd dom(numReserv)]$
THEN
$Cancel(mId, bId) \ || \ Lend(mId, bId)$
END;

Renew$(mId, bId) \triangleq$
PRE $mId \in members \ \wedge$
$bId \in books \ \wedge$
$(bId, mId) \in loans$
THEN
$skip$
END

END

Figure 7: B specification: Global machine

borrow the same book at the same time is not stated in the requirements, but it is described in the process expression by the synchronization between books and members over the input events of the loan association.

The constraints which are not easy to express in a pure process algebraic style (*i.e.,* without using recursive functions defined on the execution trace) arise from conditions involving input events from the history of inputs and from a number of entities. For instance, to adress the user requirements element 6, one must deal with a queue of active reservations of a book. 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. One has to define a controller process which is synchronized with the reservation process. This controller process takes a queue as parameter, and this queue is updated by a recursive call to the process. Its definition is the following.

```
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)) )
```

This process must be composed in parallel with the call to process `reservation` in process `book`. We have chosen a different solution, by using a guard invoking recursive function `isFirst`, as illustrated in Figure 3. Our motivations for such a choice will be more obvious when the issue of specification validation is discussed in Section 5.2. In particular, it is especially tricky to derive a pure process algebraic style to express user requirements elements 4 and 5.

Finally, each data attribute is defined by a recursive function on the execution trace. Since the execution trace contains the history of input events, any data attribute can be defined, usually quite easily. In [13], patterns are defined for attributes.

5.1.2 The B Specification

Our B specification is structured according to the style presented in [15], where a translation between UML diagrams and B specifications is defined. This style enforces modularity, by proposing to create one basic machine for each entity type and each association, and a top-level machine that defines one operation for each input event. It also simplifies the discharging of proof obligations required by the B method. Closely related styles have also been proposed [16, 18].

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 requirements class diagram (*e.g.,* of the EB^3 specification) is a good source to start with. Each class is represented by a set of instances, and each class attribute is represented by a function from this set to the attribute type. 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 input between the operands without explicitly listing these input events.

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 **Discard** operation in machine `B_Library`. Its precondition must refer to state variables from `B_Book`, `B_Loan`, and `B_Reservation`. Hence, an operation which 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 event `Discard` occurs after the execution of `loan` and `reservation`; there is no reference to the internal details of these processes.

In the EB^3 specification, guards referring to functions defined the execution 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. For instance, imagine that a book can be acquired only once, that is, it cannot be re-acquired after it has been discarded. In EB^3, this change is made by simply 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
$allBooks \in BOOK \ \wedge$
$books \subseteq allBooks$

When a book is acquired, it is added both to `books` and `allBooks`; when a book is discarded, it is removed from `books`, but it is kept in `allBooks`. The precondition of **Acquire** is changed so that a book can be acquired when it doesn't belong to set `allBooks`. This small example illustrates 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 occur. 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 what is the effect of an input event on attributes, because this information is scattered over several function definitions, whereas it is immediate to see how an attribute is influenced by input events.

Overall, the connexion 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 execution trace and contribute to the definition of ordering constraints.

These facts lead us to conclude that EB^3 is closer to the user requirements. From a user's point of view, the value of an IS is 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. In EB^3, each data attribute is defined on its own, by a single function. This specification style closely matches the view of a user. Because it is the data that matters, each data attribute can be described independently, one by one. In B, the user is forced to consider altogether 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 one who executed a `Lend` or a `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 where 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. In other words, validation makes sure that the specification meets the client's expectations. Validation is usually conducted by human inspection, sometimes supported by specification animation tools to execute some scenarios.

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

1. a member can borrow a book which is reserved by another member;
2. a member can renew a loan even if the book is 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.

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 extract some mental scheme in which an analyst will find help in this task.

5.2.1 Error Correction in EB^3

These four specification errors are ordering problems. They can be detected by an experimented EB^3 specifier with a simple review of the specification. The ordering constraints on `Lend`, `Renew` and `Reserve` are simply expressed; the only potential difficulty lies in the synchronization between `loan` and `reservation` over action `Take`, or in understanding the quantifications occurring in `book` or `member`.

The cause of these errors arise from the difficulty in a process algebra to express constraints involving several entities at the same time. For instance, to prevent the borrowing of a reserved book, process expression `loan` must be "aware" that a `Reserve` has been executed on the book. In a pure process algebraic style, a process can only communicate with another 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 execution trace, in a process expression.

The first two errors contradict requirement 5 of Section 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 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 8 provides a state-oriented solution, while Figure 9 provides a pure process algebraic solution.

```
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) = [] )
```

Figure 8: State-oriented modification to process `loan` to correct errors 1 and 2

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

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

Figure 9: Process-algebraic solution to errors 1 and 2

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

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

Figure 10: State-oriented modification to process `reservation` to correct errors 3 and 4

It is interesting to analyze back our reflexion that has led to the creation of process `controller1` in Figure 9. The problem was to write a process expression to prevent the borrowing or renewal of a reserved book. It seems natural to enunciate this assertion in the following terms: *"if a book is reserved, it cannot be lent and a loan of this book cannot be renewed."* However, this formulation cannot be easily translated into a pure process algebraic style. It is more suitable for the construction of 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 assertion: *"a lend and a renew only occur before any reservation or after* **all** *reservations are consumed"*. Therefore, we can see the need to "spy" the other members' action with the last part of the sentence; this spying is achieved in process `Controller1` by 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`.

The last two errors (3 and 4) are both related to the `Reserve` action and contradict requirements element 4. If we correct them with a guard, the modification is quite straightforward. Figure 10 provides this function and the modification in 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 do two consecutive `reserve`, 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 like we did with previous errors. Again, we first need to formulate the requirements in terms of which event sequences are allowed. Therefore, the following formulations of requirements element 4 is preferred:

- for the third error, *"a reservation of a book always takes place between loans."*;
- and for the fourth, *"A reservation of a book takes place during a loan cycle or a reservation cycle of another member."*

Figure 11 provides `Controller2` and `Controller3` processes that respectively specify to these two assertions. They must be inserted in the `main` process in parallel with other processes.

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

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

Figure 11: Process algebraic solution to correct errors 3 and 4

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`). The first ensures that a book is reserved during a loan. The second ensures that a reservation occurs when at least a reservation of another member is active. The use of the interleave operators between these two processes, combined with the synchronization on the spied actions (`Lend, Reserve, Take, Cancel, Return`), behaves like a choice. So a `Reserve` action can only occur during a reservation or a loan cycle. The last interleaved process (last two lines) allows the member `mId` to execute these actions without constraints.

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

5.2.2 Error Correction in B

In B, errors are more difficult to find, since one has to check preconditions of the operations to determine if they appropriately describe the desired ordering properties; it requires a good understanding of the state variable. A good understanding of basic set theory, functions and relations is necessary in order 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.

Once an error is located, the correction process is nearly the same as what has been done for EB^3 with guards and functions. As we already mentioned, it is natural to write constraints as implications. Therefore, they are easily translated into preconditions. Nonetheless, as it is shown in the comparison of the Section 5.1.3, some constraints expressed with preconditions may seem quite artificial in B whenever it is easy to express it in EB^3.

Here is the new definition of the operation *Renew* with the correct precondition.

$$
\begin{array}{l}
\textbf{Renew}(mId, bId) \triangleq \\
\text{PRE} \quad mid \in members \quad \wedge \\
\qquad bId \in books \quad \wedge \\
\qquad (bId, mId) \in loans \quad \wedge \\
\qquad bId \notin dom(reservations) \ \textbf{error}\ 2 \\
\text{THEN} \\
\qquad skip \\
\text{END}
\end{array}
$$

Operations *Reserve* and *Lend* were promoted from `B_Reservation` and `B_Loan` in the previous specification. To correct them, we must now refer to variables from each other; it would imply a circular USE relationship between `B_Reservation` and `B_Loan`, which is not allowed in B; we must therefore create new operations in `B_Library`.

$$
\begin{array}{l}
\textbf{NewReserve}(mId, bId) \triangleq \\
\text{PRE} \quad mid \in members \quad \wedge \\
\qquad\quad bId \in books \quad \wedge \\
\qquad\quad (bId, mId) \notin reservations \quad \wedge \\
\qquad \big(\quad bId \in dom(reservations) \ \vee \\
\qquad\quad \big(\ bId \in dom(loans) \quad \wedge \quad loans(bId) \neq mId \ \big) \quad \big) \\
\text{THEN} \\
\qquad Reserve(mId, bId) \\
\text{END};
\end{array}
$$

$$
\begin{array}{l}
\textbf{NewLend}(mId, bId) \triangleq \\
\text{PRE} \quad mid \in members \quad \wedge \quad bId \in books \quad \wedge \quad bId \notin dom(loans) \quad \wedge \\
\qquad bId \in dom(reservations) \\
\text{THEN} \\
\qquad Lend(mId, bId) \\
\text{END};
\end{array}
$$

5.2.3 Conclusion

It seems to us that the validation is easier to achieve in the event-oriented EB^3 rather than with state-oriented B. Moreover, the specification creation process is more natural with a process algebra since it streamlines the specification of ordering constraints, at least when the constraints do not involve many entities or associations.

Preconditions in B can be useful but they lack readability for the validation process. The use of guards to correct EB^3 specification tends to blur the global readability of the specification as preconditions do in a B specification. Unfortunately, the last error correction reminds us that some properties can be quite difficult to express in EB^3 without guards, and they are definitely less readable than an equivalent guard-oriented solution. It seems that when an integrity constraint involves several properties of several entities (*e.g.,* a book being reserved, lent), 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 whereas other constraints are more natural in the other.

5.3 Specification Verification

We define verification as the activity of checking that the specification satisfies some properties, which are are usually stated in a precise language. A property can be checked either by proving it or by checking it on a finite model of the specification.

5.3.1 Verification in EB^3

Currently, there is no tool for conducting verification in EB^3. Since it is similar to several process algebras, we plan to develop bridges in order to use tools develop for these process algebras (*e.g.,* FDR [9], CADP [7]). These tools are based on state-space exploration, which quickly suffers from combinatorial explosion. Proving properties about process expressions is not as common, and not very well supported by case tools. Hence, verifying an EB^3 specification is difficult.

5.3.2 Verification in B

Case tools like Atelier B [8] provide a prover to handle proofs obligations associated to machines and their refinements. Among proof obligations, the preservation of the invariant by operations is quite important for verifying properties. B is very nice for specifying static properties about the data structures. The Atelier B prover was able to automatically discharged all proof obligations associated to our B specifications.

5.3.3 Conclusion

In our example, we did not provide errors where static data integrity constraints were violated. These integrity constraints can be explicitly stated within the invariant of a B machine. A priori, it seems that such errors are easier to find in a B specification. When it is difficult to discharge a proof obligation, it is usually a good sign that the invariant is not preserved by an operation.

The same invariant property could be stated in EB^3 using functions defining the entity attributes. Proving that they are preserve by every operation consist 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 the precondition of an event.

5.4 Specification Evolution

The error correction examples we have shown illustrate 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 Section 5.1.1 (to forbid the re-acquisition of a book) showed that the B specification required more changes than the EB^3 one.

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. Both in 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 loans) are 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, but this solution introduces some redundancy in the state space, and there is a risk that future modifications to the specification 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: the Ideal Specification Process

The ideal specification process would consist of the following steps.

1. Specify the system using EB^3.

   Justification: Ordering constraints are easier to specify. Incremental specification. Less coupling between specification elements. Attributes are easier to understand. Specification is easier to validate and modify.

2. Write an equivalent specification in B, and add static data integrity constraints.

   Justification: Easier to prove the preservation of static data integrity constraints in B. Moreover, it seems quite feasible to generate some parts of the B specification, *i.e.,* the basic substitutions modifying the state variables, from the definitions of the recursive functions in EB^3.

3. Prove that the B specification is equivalent to the EB^3 specification.

   Justification: This ensures that all ordering constraints are taken into account in the B specification. An approach to realize this proof is presented in [12].

This ideal process has the advantage of exploiting the orthogonality between EB^3 and B in order to catch (or avoid) as many errors as possible. In counterpart, it is probably very expensive to use, and it would need to be supported by tools to automate the proof of equivalence between the two specifications.

Finally, here are short summary answers to the questions raised in the workshop call for papers, from an information systems perspective.

1. *What is the shape of your mental landscape, when you start conceiving a complex (concurrent, reactive, distributed) system? Is it a structure of state variables (relations, functions), or a pattern of events in time?*

   Response: Start with an event-oriented specification (*i.e.,* EB^3). Delay as much as possible the introduction of state-oriented constructs. Refine into a state-oriented specification.

2. *Is the choice between a state-oriented and an event-oriented approach dependent on the type of system to be described? How?*

   Response: Simple ordering constraints are better described in an event-oriented style; complex ordering constraints are better described in a state-oriented style.

3. *How does a system description in natural language affect the choice between state-oriented and event-oriented formalisation? How does the requirements analysis process affect the choice?*

   Response: Basic scenarios are easy to translate into an event-oriented style; complex ordering constraints are often naturally stated in a state-oriented style (*e.g.,* see the correction of errors in Section 5.2.1).

4. *The two approaches don't have to be mutually exclusive. Is it easy/desirable to move from one to the other? At which stage of development would one do that?*

   Response: Indeed, the two approaches are not mutually exclusive. We favor the use of both, starting with event-oriented, and refining into state-oriented, in order to take full advantage of each paradigm.

5. *Can one integrate the two approaches, keeping their individual advantages? If so, can one formally refine a purely state-oriented or purely event-oriented description into a hybrid one?*

   Response: EB^3 offers some integration of both paradigms, through the use of functions on traces and guards. However, one should avoid abusing of the state-oriented style in EB^3.

References

[1] Abrial, J.-R.: *The B-Book*. Cambridge University Press, Cambridge, UK, 1996.

[2] Abrial, J.-R.: Extending B without Changing it. In *First Conference on the B Method*, H. Habrias, ed., pp 169–190, November 1996.

[3] Abrial, J.-R., Mussat, L.: Introducing Dynamic Constraints in B. In *Second International B Conference*, D. Bert, ed., Lecture Notes in Computer Science 1393, Springer-Verlag, 83–128, April 1998.

[4] P. Behm, P. Benoit, A. Faivre, and J.M. Meynadier. Météor: A Successful Application of B in a Large Project. In *FM99: World Congress on Formal Methods, Toulouse, France*, Lecture Notes in Computer Science 1708, Springer-Verlag, pp 369–387. Springer-Verlag, September 1999.

[5] Bolognesi, T. and Brinksma, E.: Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, **14**(1):25–59, 1987.

[6] Butler, M. J., Waldén, M.: Distributed System Development in B. In In *First Conference on the B Method*, H. Habrias, ed., November 1996.

[7] INRIA Rhône-Alpes: CADP (Caesar/Aldebaran Development Package), `http://www.inrialpes.fr/vasy/cadp/`

[8] CLEARSY System Engineering: Aix-en-Provence, France, `http://www.clearsy.com/`

[9] Formal Systems (Europe) Ltd.: Failures-Divergences Refinement: FDR2 User Man - ual (1997), `http://www.formal.demon.co.uk`

[10] Fraikin, B., and Frappier, M.: EB^3**PAI**: an interpreter for the EB^3 specification language *Proc. of FM-TOOLS 2002, The 5th Workshop on Tools for System Design and Verification*, 2002.

[11] Fraikin, B., and Frappier, M.: Optimizing memory space in the EB^3 process algebra interpreter. *Proc. ICCSSEA 2002, Software and Systemes Engineering and their Applications, Volume I, Session 4*, 2002.

[12] Frappier, M., Laleau, R.: Verifying Event Ordering Properties for Information Systems. *The 3rd International Conference of B and Z Users*, Lecture Notes in Computer Science 2651, Springer-Verlag, Turku, Finland, 4-6 June 2003, pp 421–436.

[13] Frappier, M., St-Denis, R.: EB^3: an Entity-Based Black-Box Specification Method for Information Systems, Software and System Modeling, to appear.

[14] Hoare, C. A. R.: *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, 1985.

[15] Laleau, R. Mammar, A.: 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, IEEE Computer Society Press, September 2000.

[16] Meyer, E., Souquières, J.: A Systematic approach to Transform OMT Diagrams to a B specification. In *Formal Methods (FM'99)*, J.M. Wing, J. Woodcook, J. Davies, eds., Lecture Notes in Computer Science 1708 vol. 1, Springer-Verlag, September 1999, pp 875–895.

[17] Milner, R.: *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, 1989.

[18] Snook, C., Butler, M.: Tool-Supported Use of UML for Constructing B Specifications, Declarative Systems and Software Engineering Research Group, Department of Electronics and Computer Science, University of Southampton, United Kingdom.
`http://www.ecs.soton.ac.uk/ mjb/U2Bpaper2.pdf`

Share sensitive information only on official, secure websites. Menu. Sign In.Â Family of EB-3 Visa Holders. If your I-140 petition is approved, your spouse and unmarried children under the age of 21 may be eligible to apply for admission to the United States in E34 (spouse of a â€œskilled workerâ€ or â€œprofessionalâ€) or EW4 (spouse of an â€œother workerâ€). and E35 (child of a â€œskilled workerâ€ or â€œprofessionalâ€) or EW5 (child of an â€œother. workerâ€). More Information. Special Immigrant Religious Workers. Health Care Worker Certification. The EB3 visa is a powerful way for a foreign worker to get a U.S. green card. In this guide, I explain the EB3 processing time.Â As a general overview the EB3 processing time is anywhere from 1 to 3 years, but nationals of some countries may wait up to 10+ years. Overview. EB3 Visa Overview. EB3 Visa Processing Time (Step-by-Step). PERM labor certification. I-140 Immigrant Petition. Immigrant Visa Processing / Adjustment of Status. The behavioural comparison of systems is an important concern of software engineering research. For example, the areas of specification discovery and specification mining are concerned with measuring the consistency between a collection of execution traces and a program specification. This problem is also tackled in process mining with the help of measures that describe the quality of a process specification automatically discovered from execution logs.Â analysis of information systems based on recorded executions of a process. Given a specification. and a log, process mining strives for quantifying the share of recorded behaviour that is in line.Â behavioural comparison of systems, specifications of systems, and logs. Yet, they are useful only if. The following tables compare general and technical information for a number of relational database management systems. Please see the individual products' articles for further information. Unless otherwise specified in footnotes, comparisons are based on the stable versions without any add-ons, extensions or external programs. The operating systems that the RDBMSes can run on. Information about what fundamental RDBMS features are implemented natively. Information systems data integrity constraints eb3 process expressions refinement. This is a preview of subscription content, log in to check access. Preview.Â Fraikin, B., Frappier, M., Laleau, R.: State-based versus event-based specifications for information systems: a comparison of B and EB3. Software and Systems Modeling 4(3), 236â€“257 (2005)CrossRefGoogle Scholar. 7. Frappier, M., Laleau, R.: Proving event ordering properties for information systems.