How to Verify Dynamic Properties of Information Systems

Neil Evans and Helen Treharne,
Department of Computer Science,
Royal Holloway,
University of London.

Régine Laleau,
IUT de Fontainebleau,
Département Informatique,
Université Paris 12.

Marc Frappier,
Département d'informatique,
Université de Sherbrooke,
Canada.

Abstract

EB^3 is an established formal technique, based on process algebra, for specifying Information Systems (IS) that have both complex state and event based features; as yet, EB^3 has no tool support. Another formal technique called CSP $\|$ B uses two existing analysis tools, FDR and the B-Toolkit, to support the verification of state/event based systems. However, the CSP $\|$ B approach has never been applied to this specialised domain. In this paper, we use a specification pattern of EB^3 to motivate a new style of specification in CSP $\|$ B appropriate for IS. We demonstrate this using an example system and show that the verification of its dynamic properties is now amenable to tool support.

1. Introduction

The main characteristic of information systems (IS) is the great number of complex, interrelated data structures which are subject to strong integrity constraints. Currently, the most widely used paradigm for specifying IS is the state transition. A state transition specification consists of a state space (defined via state variables), an invariant (a consistency property), and operations describing transitions by modification of the state variables. Upon receipt of an external trigger, an operation is called to compute the new state and to produce an output if necessary. Existing notations that have been used for state transition specifications of IS include UML with OCL [16], and model-based notations like B [17, 14], and Z [2].

Individually, the programs of an information system are not necessarily complex. However, there can be numerous distributed programs processing the data, and these may be subject to complex ordering constraints to fulfil the user requirements of the system. One difficulty with state transition specifications is the validation of dynamic properties such as ordering constraints and liveness requirements. For example, ordering can be expressed through conditions on state variables and, in our experience, the validation of user requirements against a formal specification can be hard to achieve. In this paper we are interested in developing techniques for formally checking dynamic properties of IS. The specification language EB^3 [10] has been defined for the purpose of specifying IS. It is chiefly event-driven, but it also includes some state-oriented constructs. Section 2 provides an overview of EB^3, and Section 4 illustrates, using a library example, the constituent parts of an EB^3 information system specification. A sample set of user requirements for the library system example is given in Section 3. Sections 2, 3 and 4 summarise previous work presented in [8] but also highlight the important aspects of this work to provide a basis for this paper's contribution.

Up to now, the EB^3 approach does not have any tools to support verification, and so it is not possible to demonstrate mechanically that the user requirements are met by the EB^3 specification. Another formal technique called CSP $\|$ B [20] uses two existing analysis tools, FDR [5] and the B-Toolkit [18], to support the verification of state/event based systems. In this paper, we use a specification pattern of EB^3 to motivate a new style of specification in CSP $\|$ B appropriate for IS, and this contribution of the paper is described in Section 5. The paper also shows, in Section 6, that dynamic properties of an information system can be captured in the CSP $\|$ B approach, and we demonstrate this by verifying, with the aid of tool support, that the specification of the system fulfils certain ordering constraints and, in addition, exhibits a liveness requirement. The paper concludes with some remarks about the results obtained and future directions for the EB^3 and CSP $\|$ B approaches.

2. An Overview of EB^3

An EB^3 specification [10] has the following elements:

1. a user requirements class diagram which includes entities, associations, and their respective actions and attributes. These diagrams are based on entity-relationship model concepts [3]. An entity is an object whose attributes (*i.e.,* both key and non-key at-

tributes) represent the properties that describe the object, and actions can change the values of attributes. The relationships between entities are called associations. For example, a member entity in a library system may be described by the member's identifier (its key attribute), name and address (its non-key attributes);

2. a process expression, 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 behaviour of `main` is defined as follows: `trace` denotes the system trace, which is a list comprised of *valid* input events accepted so far in the execution of the system, $t::\sigma$ denotes the right append of an input event σ to trace t, and `[ ]` denotes the empty trace.

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

An input event σ is an instantiation of (the input parameters of) an action. The signature of an action a is given by:

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

where $p_1, \ldots, p_n$ are input parameters of types $T_1, \ldots, T_n$ and $p_{n+1}, \ldots, p_m$ are output parameters of types $T_{n+1}, \ldots, T_m$. An instantiated action $\mathsf{a}(t_1, ..., t_n)$ also constitutes an elementary process expression.

Complex EB^3 process expressions can be constructed from elementary process expressions (instantiated actions) using the following operators: sequence (`.`), choice (`|`), Kleene closure (`^*`), interleaving (`|||`), parallel composition (`||`, *i.e.,* CSP's synchronisation on shared actions), guard (`==>`), process call, and quantification of choice (`|x : T : ...`) and interleaving (`|||x : T : ...`). The EB^3 notation is similar to CSP [11] but the main differences between EB^3 and CSP are: i) EB^3 allows one to use a single state variable, the system trace, in predicates of guard statements (as we shall see below); 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.

The process expressions are defined from the point of view of the actions that can be performed and the inputs that need to be provided. As such, any outputs can essentially be viewed as by-products of a system that performs valid input traces because, in such a system, they are not used to control the flow of execution. However, outputs do have an important role when invalid traces are taken into consideration. This is discussed in [10] but, since we are only interested in valid traces, output is beyond the scope of this paper. Therefore, we do not consider input-output rules (element 4 of an EB^3 specification).

We do, however, need to consider recursive functions (element 3) since they are used in EB^3 to control which actions are enabled and hence which events can be performed in a `trace`. For example, if a recursive function is used within a predicate of a guarded expression, then the action is blocked until its guard becomes true (*i.e.,* when the function returns true, or a boolean expression which uses the return value of a function evaluates to true). The value of the guard evolves as new events are executed because these newly executed actions may change the value of the attributes used in the guards. Examples of the use of guards and their definitions will be given in Figures 4 and 5 (Section 4). Note, these recursive functions do not appear in the `trace` (since traces are made of input events only).

The definition of a recursive function is written in a functional language style and traverses the system trace. Standard list operators such as `first`, `last` and `front` are used to retrieve elements of the trace and truncate the trace appropriately. The special value `nil` is used to denote undefinedness (for example `first([])` returns `nil`). Finally, the symbol "_" is used to pattern match with any list element.

3. User Requirements of a Library System

The library system has to manage loans and reservations of books by members and was first presented in [8]. The requirements are stated below so that we can refer to them explicitly in Section 6.

1. A book is acquired by the library. It can be discarded, but only if it is not lent and not reserved.
2. A Member must join the library in order to borrow a book.
3. A book can be reserved if and only if it is lent or already reserved by another member.
4. A member can borrow or renew a book loan unless the book is already reserved.
5. 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.
6. Anyone who reserved a book can cancel the reservation at anytime before the reservation is used.

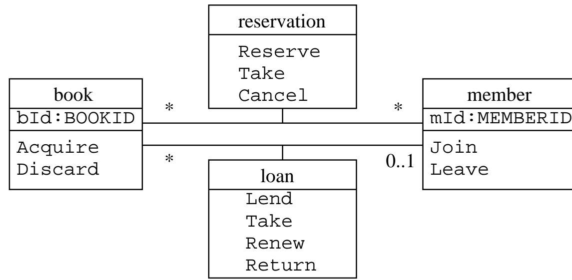

Figure 1. User requirements class diagram

7. A member can leave the library membership only when all his loans are returned and all his reservations are either used or cancelled.

It is possible to interpret the user requirements in two ways: *safety properties* state that certain bad things should not happen. For example, we can infer from requirement 1 that books on loan must not be discarded. *Liveness properties* state that certain good things should happen. For example, requirement 6 is a liveness property because it states that a member should be allowed to cancel a reservation before the book is taken. These kinds of properties have implications on the formal analyses presented in Section 6.

4. Component Parts of an EB^3 Specification

Figure 1 shows the user requirements class diagram used to construct the specification. We identify two main entity types, `member` and `book`, and two associations between them, `reservation` and `loan`, with their corresponding actions and attributes. Figure 2 provides the signature of actions (`void` denotes an action with no input-output rule).

```
Acquire(bId:BOOKID):void
Discard(bId:BOOKID):void
Join(mId:MEMBERID):void
Leave(mId:MEMBERID):void
Lend(bId:BOOKID,mId:MEMBERID):void
Renew(bId:BOOKID,mId:MEMBERID):void
Return(bId:BOOKID,mId:MEMBERID):void
Reserve(bId:BOOKID,mId:MEMBERID):void
Cancel(bId:BOOKID,mId:MEMBERID):void
Take(bId:BOOKID,mId:MEMBERID):void
```

Figure 2. EB^3 signature of actions

In [10], a strategy to design an EB^3 specification is described, in which each entity type and association is represented by a process. The interactions between entities (*e.g.,* when a member borrows a book) are expressed by composing entities in parallel in the process `main`:

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

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

Figure 3. EB^3 Entity processes

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

Simple ordering constraints can be expressed by using process expressions. Figure 3 illustrates these constraints on books and members.

The entity processes in turn call the processes `loan` and `reservation` which describe the behaviour of the corresponding associations (see Figure 4). The multiplicity of an association (*e.g.,* 1..*,*, 0..1, and so on) is expressed in the process expressions of the entities by selecting an appropriate quantification operator to encapsulate the call to the association process expression. For example, we use $|\ x$ when an entity is related to at most one entity, as is case in the loan relationship between books and members. We use $|||$ when an entity is related to a number of entities, as is the case of the loan relationship between members and books. Several patterns have been defined to translate requirements class diagrams into process expressions (see [10]).

Notice, there is one common definition of the `loan` and `reservation` processes which means that these processes maintain the associations by taking the evolving behaviour of both books and members into account. The benefit of this common separation is that n-ary associations can be clearly defined. However, as is seen in [8], a simple definition of the associations is usually insufficient to determine all the causal dependencies between the actions of the associations that are needed to meet the user requirements. In order to remedy this problem, the EB^3 specification can be augmented in two distinct ways. One way is to use a pure process algebraic style, as presented in [8], where control processes are used to constrain the execution of actions via synchronisation. However, these process expressions can become complex and difficult to understand. This is especially true when the ordering constraints involve several entities and associations, which is the case for user re-

```
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) ;

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

Figure 4. EB^3 Association processes

quirements 3 and 4.

The alternative in EB^3 is to use recursive functions as guards to extract information from the system trace (examples of these are shown in Figure 4). The boolean functions `isNotReserved`, `isFirst` and `canBeReserved` are defined in Figure 5. These functions would typically be referenced in the requirements class diagram, for documentation purposes, although we have omitted them in Figure 1, for the sake of concision. They are used to control the execution of a process expression by enabling and disabling the actions that follow the guards. For example, given the system trace `<Acquire.b1,Join.m1,Join.m2,Lend.m1.b1>` as the first input parameter and `b1` as the second input parameter to the function `isNotReserved`, the function returns the result `true`. It provides a way of taking a snapshot of the dynamic state of the system as it has evolved. Therefore, if the system has evolved to a state in which the function `isNotReserved` returns `true` then the `Lend` action is enabled. Since the system trace contains all input events, any attribute can be represented by a recursive function.

The resulting EB^3 specification, which includes appropriate recursive functions to constrain the behaviour of the process expressions, is state-oriented in style because it uses the trace of the system, and thus implicitly the attribute values of the entities, to control the flow of execution. Currently, there is no tool for conducting verification in EB^3. Therefore, we cannot formally prove that the EB^3 specification (illustrated in Figures 3, 4, and 5) meets the user requirements described in Section 3. One solution is to develop bridges between EB^3 and process algebras supported by tools (such as FDR [5] and CADP [12]). The remainder of this paper is a basis for this approach using a formalism called CSP || B.

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

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

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

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

Figure 5. EB^3 Function definitions

5. Applying CSP || B to Information Systems

CSP || B is a general purpose approach to the formal analysis of systems with complex state and behavioural aspects. See [20] for details of the theory underlying this approach. However, in this paper we shall construct a CSP || B specification that is novel in several respects. Firstly, the CSP architecture of this model (shown in Figure 6) is different from those of other CSP || B analyses (see [4] for example). Traditionally, each B machine is accompanied by a unique CSP controller, whereas the definitions given in this section comprise of two controller processes ($BOOKCTRL$ and $MEMBERCTRL$) and three B machines ($BookEntity$, $MemberEntity$ and $Assoc$). The association machine $Assoc$ is controlled by the synchronising events of $BOOKCTRL$ and $MEMBERCTRL$. Secondly, the language of the controller processes has until now been restricted to a sequential subset of CSP. Our introduction of interleaving and sequential composition into controller processes has implications for the approach in general. These issues will be addressed as we consider the formal analysis of the user requirements in Section 6.

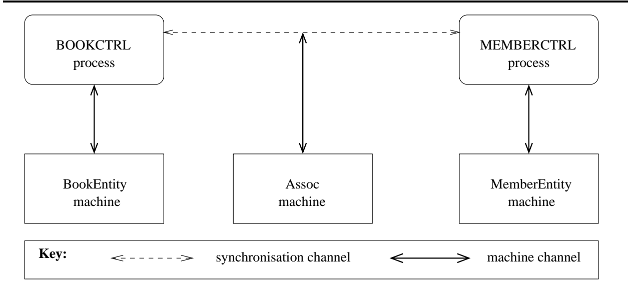

Figure 6. A CSP || B architecture for IS

$$BOOKCTRL(b) = acquire.b \rightarrow (ASSOCBL(b) \parallel \cdots);\ \cdots$$

$$ASSOCBL(b) = (\Box_{m \in MEMBERID} LOAN(m,b);\ ASSOCBL(b)) \Box Skip$$

$$MEMBERCTRL(m) = join.m \rightarrow ((|||_{b \in BOOKID} ASSOCML(m,b)) \parallel \cdots);\ \cdots$$

$$ASSOCML(m,b) = (LOAN(m,b);\ ASSOCML(m,b)) \Box Skip$$

Figure 7. Producer-modifier pattern in CSP

In applying CSP || B to IS, we use CSP to model the complex behaviour and, in contrast to the use of recursive functions in EB^3, the state of the information system (which corresponds to the values of entity attributes and the associations) is defined explicitly via the state variables of B machines. In order to maintain the state, each B operation corresponds to a CSP event so that whenever such an event is performed the corresponding B operation is executed.

Since EB^3 is a language based on process algebra, we begin by outlining an intuitive (but naïve) approach to defining the corresponding library system in CSP. The definitions of `book` and `member` in the EB^3 specification follow a producer-modifier-consumer pattern in which the `Acquire` and `Join` actions form the producer components of the pattern, the `loan` and `reservation` processes are the modifiers, and the `Discard` and `Leave` actions represent the consumer elements of the pattern (see Figure 3). In Figure 7, the definitions of these entity types (which we call $BOOKCTRL$ and $MEMBERCTRL$) give a flavour of the same pattern in CSP in which actions are modelled as CSP events (*e.g.,* the EB^3 action `Acquire` corresponds to the event $acquire$). The interleaving, parallel composition, and choice operators of EB^3 have direct counterparts in CSP. Sequencing in EB^3 can be achieved in CSP by using event prefixing ($\rightarrow$). The Kleene closure of an EB^3 process is modelled by using recursive calls to the corresponding CSP process and sequential composition of (terminating) processes (;). Termination is defined via the primitive process $Skip$.

Note the similarity between the EB^3 and CSP definitions for the different patterns of the 'one-to-many' and 'one-to-at-most-one' multiplicities in the loan association. Both $ASSOCBL(b)$ and $ASSOCML(m,b)$ model the zero or more iterations of the (translated) loan process, but the indexed external choice included in the process $ASSOCBL(b)$ ensures that only one member can borrow it at any one time. The interleaving in $MEMBERCTRL(m)$ allows the member m to borrow many different books at the same time.

As we discussed in Section 4, the initial specification of the common process `loan` in EB^3 is insufficient to yield all causal dependencies that are needed to meet the user requirements, and guards were introduced into the EB^3 specification to constrain the enabling of actions. A direct translation from the initial EB^3 specification to CSP would also be insufficient to meet the user requirements. Since we cannot use guards (based on the `trace`) in CSP, we have devised a new approach to specifying IS in CSP || B based on the different viewpoints of the entities with respect to their associations. That is, we choose to retain the produce-consumer element of the pattern but modifier components will be defined separately for each of the entities (this is described in Section 5.2).

5.1. Using B to maintain the state

The continuous update of the state in B means that we can simply extract state information as and when it is required. Another benefit of using B is that we have an explicit interface specification for the entities and associations; this is important for the CSP definitions.

We model the state of an information system as follows: the key attributes of the entity are used to define the set of instances, and non-key attributes are defined as functions from the set of keys to attribute values. An entity type definition contains operations which define the producer and consumer elements of an instance of that entity; the machine corresponding to the book entity type is given in Figure 8. Normally, in a B specification we might include the precondition $bid \notin bookInstances$ in the **acquire** producer operation. In this new CSP || B style we need only capture the preconditions that are appropriate for a particular instance. Since the CSP model, given in Section 5.2, will be an interleaving of individual instances of books, it would not be easy to discharge such a precondition because the CSP process expressions do not have any concept of a 'collection' of books. However, since we are adopting the producer-consumer pattern we can be assured that the producer operation will not be called again for a particular instance of book until it has been consumed. Thus, such preconditions are not relevant in this example. A similar informal justification applies to the member entity type.

```
MACHINE BookEntity
VARIABLES bookInstances
INVARIANT bookInstances ⊆ BOOKID
INITIALISATION bookInstances := ∅
OPERATIONS
  acquire ( bid ) ≙ PRE bid ∈ BOOKID THEN
     bookInstances := bookInstances ∪ { bid }
   END ;
  discard ( bid ) ≙ PRE bid ∈ BOOKID THEN
     bookInstances := bookInstances − { bid }
   END
END
```

Figure 8. Book Entity B machine

The associations between the entities are captured in another separate machine, as shown in Figure 9. The functions $loanRel$ and $reservSeq$ capture the two associations, and there are standard patterns for mapping multiplicities to quantified predicates [15]. Therefore, any operation in the association machine must preserve this invariant. The $loanRel$ keeps track of members who have borrowed particular books, and so it is entirely appropriate for the **lend** operation to update this function (this corresponds to the `Lend(mId,bId) -> mId` case in the definition of `currentBorrower` in Figure 5). The action `Take` is also referred to in `currentBorrower` but, in addition, it occurs in the `reservQueue` function to update the list of members that are waiting to reserve a particular book. The sequence of members associated with book bb, $reservSeq(bb)$, corresponds to the list returned by `reservQueue`. Thus, it is no coincidence that $reservSeq$ is updated along with $loanRel$ in the **take** operation.

As well as typing information, the preconditions of the **take** operation include the predicate *reservSeq*(*bb*) $\neq$ []. This explicit assumption gives a clear indication to the CSP controllers that a $take$ event should not occur when there are no reservations for the book bb. The other preconditions are required to preserve the invariant. Performing a CSP event triggers the corresponding machine operation, and it is the responsibility of the specifier to ensure that the operation's precondition holds.

Interestingly, the **renew** operation takes two parameters of the appropriate type but does not update the B state. This is consistent with the EB^3 recursive functions because the $Renew$ action is never used as an explicit pattern matching expression and thus it does not manipulate the entity and association values. It is included so that the operations match the actions of the association class in Figure 1. (The operation would update state if we tracked the renewal dates of books as non-key attributes.) Note, the signatures of the operations of the B machines are identical those of the actions given in Figure 2. In addition to these operations the **anyReserved** operation is provided. It queries the state of the B machine to determine if the book has any associated reser-

```
MACHINE Assoc
USES BookEntity , MemberEntity
VARIABLES loanRel , reservSeq
INVARIANT
loanRel ∈ bookInstances ⇸ memberInstances ∧
reservSeq ∈ bookInstances ⇸ iseq ( memberInstances ) ∧
∀ ( bb , mm ) . ( bb ∈ bookInstances ∧
mm ∈ memberInstances ∧
bb ↦ mm ∈ loanRel ⇒ mm ∉ ran ( reservSeq ( bb ) ) )
INITIALISATION loanRel := ∅ || reservSeq := ∅
OPERATIONS
 lend ( mm , bb ) ≙
  PRE mm ∈ memberInstances ∧ bb ∈ bookInstances
   ∧bb ∉ dom ( loanRel ) ∧ mm ∉ ran ( reservSeq ( bb ) )
  THEN
   loanRel := loanRel ∪ { bb ↦ mm }
  END ;
 renew ( mm , bb ) ≙ PRE ...THEN skip END ;
 return ( mm , bb ) ≙
  PRE ... ∧ bb ↦ mm ∈ loanRel THEN
   loanRel := loanRel − { bb ↦ mm }
  END ;
 reserve ( mm , bb ) ≙
  PRE ... ∧ bb ↦ mm ∉ loanRel THEN
   reservSeq ( bb ) := reservSeq ( bb ) ← mm
  END ;
 take ( mm , bb ) ≙
  PRE ... ∧ reservSeq ( bb ) ≠ [ ] ∧ mm = first ( reservSeq ( bb ) )
   ∧ bb ∉ dom ( loanRel )
  THEN
   reservSeq ( bb ) := tail ( reservSeq ( bb ) ) ||
   loanRel := loanRel ∪ { bb ↦ mm }
  END ;
 cancel ( mm , bb ) ≙
  PRE ...THEN
   LET xx BE xx = ( reservSeq ( bb ) ) ⁻¹ ( mm ) ∧ xx ∈ ℕ IN
     reservSeq ( bb ) := reservSeq ( bb ) ↑ xx ⌢
                         ( reservSeq ( bb ) ↓ xx )
   END
  END ;
rep , mm ⟵ anyReserved ( bb ) ≙
  PRE bb ∈ bookInstances THEN
   IF reservSeq ( bb ) ≠ [ ] THEN
     rep := yes || mm := first ( reservSeq ( bb ) )
   ELSE rep := no || mm := defaultMember
   END
  END
END
```

Figure 9. Association B machine

vations and, if this is the case, the output mm is the head of the reservation queue associated with the book; this is comparable to the `isFirst` function in the EB^3 specification. Its relevance will become clear when we examine the behavioural view of the system.

5.2. Using CSP to provide a behavioural view

Definitions based on different viewpoints (a book's viewpoint and a member's viewpoint in this example) give a more behavioural view of the system as a whole. This is orthogonal to the EB^3 definition which is de-

$$MEMBERCTRL(m) = join.m \rightarrow (|||_{b \in BOOKID} ASSOCM(m, b));\ leave.m \rightarrow MEMBERCTRL(m)$$

$$\begin{aligned} ASSOCM(m, b) = & (lend.m.b \rightarrow BORROWEDM(m, b);\ ASSOCM(m, b)) \,\Box \\ & (reserve.m.b \rightarrow RESERVEDM(m, b);\ ASSOCM(m, b)) \,\Box\, Skip \end{aligned}$$

$$BORROWEDM(m, b) = (renew.m.b \rightarrow BORROWEDM(m, b)) \,\Box\, (return.m.b \rightarrow Skip)$$

$$RESERVEDM(m, b) = (take.m.b \rightarrow BORROWEDM(m, b)) \,\Box\, (cancel.m.b \rightarrow Skip)$$

Figure 10. CSP definition of member behaviour

rived from a requirements class diagram of the information system. We therefore dispense with the skeleton CSP processes shown in Figure 7 and, with a behavioural view in mind, we can construct a new CSP representation of the library system. We consider the actions (events) of the system from two different points of view: books and members. From a member's point of view, any book can be lent or reserved. If a book has been lent to the member then the loan can be renewed or else the book can be returned. Once a book has been reserved by a member, it can either be taken or the reservation can be cancelled by the member. This is reflected in the definition of $MEMBERCTRL$ shown in Figure 10.

When a book has been acquired, it can be borrowed either by performing a $take$ event or a $lend$ event depending on whether it has been reserved or not. Once a book has been borrowed, it can be returned (by the borrower) and reserved (by anyone). Any reservation can be cancelled prior to taking the book and, in addition, a loan can be renewed only if there are no reservations of the book. Only when the book is no longer borrowed and there are no outstanding reservations can it be discarded. This motivates the definition of $BOOKCTRL$ in Figure 11.

Note that from the members' point of view, they are always prepared to $take$ a book that they have reserved. However, the book controller only allows the member at the head of its reservation queue to take the book once it has been returned. This information is retrieved from the association machine by using the $anyReserved$ query. These constraints are enforced by the parallel composition of $BOOKCTRL$ and $MEMBERCTRL$ as follows:

$$\begin{aligned} MAIN \ = \ & (|||_{b \in BOOKID} BOOKCTRL(b)) \\ & \parallel \\ & (|||_{m \in MEMBERID} MEMBERCTRL(m)) \end{aligned}$$

Note that CSP uses multiway parallelism and it is therefore possible to define n-ary associations between entity types in addition to the binary associations that are defined in this example.

6. Verifying User Requirements in CSP || B

Theoretical results of the CSP $\parallel$ B approach, [20], enable us to verify properties of this specification of the library system using the refinement checking capabilities of FDR. *Process-oriented specifications* [19] are CSP processes that are used to formalise the user requirements given in Section 3. We aim to show that a process-oriented specification is refined by the CSP $\parallel$ B model, thereby proving that the model does not violate these constraints.

In order to use FDR to prove refinement, we must consider the CSP controllers of the model in isolation. However, by ignoring the B machines, we lose vital state information that affects the behaviour of the controllers, and this can determine whether the refinement holds. The solution to this problem is to augment the controllers with enough state information to mimic the effects of the B machines on their controllers.

6.1. Safety Requirements

By characterising all desirable patterns of behaviour as a process-oriented specification we can verify a safety property by showing that the system (i.e. the CSP $\parallel$ B specification, in our case) is a trace refinement of the specification. A failure of the refinement check demonstrates that the system is capable of bad behaviour.

Rather than attempt to express all desirable patterns of behaviour in a single process-oriented specification, we define several independent specifications whose 'conjunction' captures an overall notion of desirable behaviour. The advantage of this approach is to enable the isolation of the state information that needs to be lifted in order to verify each property, and it also makes it easier to identify the source of any unexpected behaviour.

The 'conjunction' of process-oriented specifications in the traces model corresponds to parallel composition. As described in [4], for specifications R and S, and system P:

$$\text{if } R \sqsubseteq_T P \text{ and } S \sqsubseteq_T P \text{ then } R \parallel S \sqsubseteq_T P$$

where $\sqsubseteq_T$ denotes trace refinement.

User requirements 1, 3 and 4 concern the ordering constraints of the events from the moment a book is acquired to

$$BOOKCTRL(b) = acquire.b \rightarrow ASSOCB(b);\ discard.b \rightarrow BOOKCTRL(b)$$

$$\begin{aligned} ASSOCB(b) = {} & anyReserved.b?x?y \rightarrow \\ & if\ x = no\ then\ (\Box_{m \in MEMBERID}\ lend.m.b \rightarrow BORROWEDB(m,b)) \Box\ Skip \\ & else\ (take.y.b \rightarrow BORROWEDB(y,b)) \Box (\Box_{m \in MEMBERID}\ cancel.m.b \rightarrow ASSOCB(b)) \end{aligned}$$

$$\begin{aligned} & BORROWEDB(m,b) = anyReserved.b?x?y \rightarrow \\ & \quad if\ x = no\ then\ (return.m.b \rightarrow ASSOCB(b)) \Box (renew.m.b \rightarrow BORROWEDB(m,b)) \Box \\ & \qquad (\Box_{m' \in MEMBERID}\ reserve.m'.b \rightarrow BORROWEDB(m,b)) \\ & \quad else\ (return.m.b \rightarrow ASSOCB(b)) \Box \\ & \qquad (\Box_{m' \in MEMBERID}\ reserve.m'.b \rightarrow BORROWEDB(m,b)) \Box (\Box_{m' \in MEMBERID}\ cancel.m'.b \rightarrow BORROWEDB(m,b)) \end{aligned}$$

Figure 11. CSP definition of the library from the books' viewpoint

the moment it is discarded. Requirements 2 and 7 concern the ordering constraints of the events from the moment a member joins the library to the moment the member leaves.

User requirement 5 is more specialised because it concerns the order in which a book is taken based on the order in which it is reserved. In addition, it also demands that the cancellation of reservations needs to be taken into consideration. Note that this requirement only concerns a subset of the events in $MAIN$. Since we are indifferent to the other events, they must be allowed to occur freely. This is achieved by interleaving the process $RUN(sigma - specsigma)$ with $SPEC5$ to give the complete specification called $SPEC$ ($sigma$ is the set of all events and $specsigma$ is the set of events used in $SPEC5$):

$$SPEC5(b) = reserve?m.b \rightarrow SPEC5'(\langle m \rangle, b)$$

$$\begin{aligned} & SPEC5'(\langle m \rangle, b) = \\ & \ (take.m.b \rightarrow SPEC5(b)) \Box \\ & \ (cancel.m.b \rightarrow SPEC5(b)) \Box \\ & \ reserve?m' \in (MEMBERID - \{m\}).b \rightarrow \\ & \qquad SPEC5'(\langle m, m' \rangle, b) \end{aligned}$$

$$\begin{aligned} & SPEC5'(M, b) = \\ & \ (take.head(M).b \rightarrow SPEC5'(tail(M), b)) \Box \\ & \ (cancel?m' \in list2set(M).b \rightarrow \\ & \qquad SPEC5'(remove(M, m'), b)) \Box \\ & \ (reserve?m' \in (MEMBERID - list2set(M)).b \rightarrow \\ & \qquad SPEC5'(M \frown \langle m' \rangle, b)) \end{aligned}$$

$$\begin{aligned} & SPEC = \\ & \ RUN(sigma - specsigma) \mid\mid\mid (\mid\mid\mid_{b \in BOOKID} SPEC5(b)) \end{aligned}$$

The process $SPEC5'$ is parameterised by a sequence of $MEMBERID$ and a book. This sequence (which is either a singleton sequence $\langle m \rangle$, or else has at least two elements M) corresponds to the members who have reserved the book. Note that the inputs of the $cancel$ and $reserve$ events have constraints that depend on the sequence. (The function $list2set$ converts a sequence to its set of elements, and $remove$ takes a particular element from a sequence.) We check whether this specification is met by performing the refinement check $SPEC \sqsubseteq_T MAIN$ in FDR. If this succeeds then we can infer, using a result given in [20]:

$$SPEC \sqsubseteq_T MAIN \parallel BookEntity \parallel Assoc \parallel MemberEntity$$

Unfortunately, this check fails because the query $anyReserved.b?x?y$, in the context of the B machine $Assoc$, assigns the value yes to the variable x only when book b has been reserved, and it assigns the head of the book's reservation queue to y. By checking the process in isolation, both yes and no are possible assignments to x regardless of any reservations, and any member can be assigned to y. As a consequence, it is possible for b to be taken by anyone even if there are no reservations (since this is dependent on the query). In order to mask this impossible behaviour, it is necessary to constrain the values assigned by this query. We can state this explicitly within the process body only by referring to the reservations associated with this book: the value yes is assigned to x when the reservation queue is non-empty, and no when the queue is empty. In addition, whenever the queue is non-empty, the value assigned to y is the head of this queue. Thus, we augment the process $ASSOCB(b)$ and its subprocess $BORROWEDB(m, b)$ with a parameter n that tracks b's reservation queue (see Figure 12).

The reservation queues for all books are recorded as the single state variable $reservSeq$ in the $Assoc$ machine. We must ensure that the augmenting parameter n in Figure 12 corresponds to the reservation queue for book b in $reservSeq$. Since the book controller is made up of an interleaving of (independent) subprocesses (one for each book), this can only be achieved if the reservation queues of $reservSeq$ are independent of each other. In other words, the operations of $Assoc$ that update $reservSeq$ should only affect one reservation queue at a time. This is enforced by parameterising each operation with a single book. This, in

$$BOOKCTRL(b) = acquire.b \rightarrow ASSOCB(b, \langle\rangle);\ discard.b \rightarrow BOOKCTRL(b)$$

$$ASSOCB(b,n) = anyReserved.b?x?y\{(\#n > 0 \Rightarrow (x = yes \wedge y = head(n))) \wedge (\#n = 0 \Rightarrow x = no)\} \rightarrow$$
$$if\ x = no\ then\ (\square_{m \in MEMBERID}\ lend.m.b \rightarrow BORROWEDB(m, b, \langle\rangle)) \square\ Skip$$
$$else\ (take.y.b \rightarrow BORROWEDB(y, b, tail(n))) \square (\square_{m \in MEMBERID}\ cancel.m.b \rightarrow ASSOCB(b, remove(n, m)))$$

$$BORROWEDB(m, b, n) = anyReserved.b?x?y\{(\#n > 0 \Rightarrow x = yes) \wedge (\#n = 0 \Rightarrow x = 0)\} \rightarrow$$
$$if\ x = no\ then\ (return.m.b \rightarrow ASSOCB(b, \langle\rangle)) \square (renew.m.b \rightarrow BORROWEDB(m, b, \langle\rangle)) \square$$
$$(\square_{m' \in MEMBERID}\ reserve.m'.b \rightarrow BORROWEDB(m, b, \langle m' \rangle))$$
$$else\ (return.m.b \rightarrow ASSOCB(b, n)) \square (\square_{m' \in MEMBERID}\ reserve.m'.b \rightarrow BORROWEDB(m, b, n \frown \langle m' \rangle)) \square$$
$$(\square_{m' \in MEMBERID}\ cancel.m'.b \rightarrow BORROWEDB(m, b, remove(n, m')))$$

Figure 12. Augmented definition of book controller for user requirement 6

conjunction with the use of interleaving in the controller processes, achieves the independence requirement.

6.2. A Liveness Requirement

User requirement 6 states that a reservation can be cancelled before the book is taken, *i.e.,* the system cannot refuse to perform a cancellation from the moment a book is reserved to the moment the book is taken. We use FDR and refinement in the stable failures model to check this liveness property. However, as in Section 6.1, by analysing the CSP controllers in isolation we lose vital information held within the B machines. In this instance, it would be possible for $anyReserved$ to assign no to the variable x even if there are outstanding reservations. As such, the conditional branching in $BOOKCTRL$ (which depends on the value of x) results in all cancellations being refused. Once again, we must lift enough state from the B machines into the CSP in order to prevent FDR returning such false negatives. (Also note that refinement checking in the stable failures model discovers any traces that violate a corresponding refinement check in the traces model. Hence, lifting state to prevent such violations is equally important here.)

The process-oriented specification corresponding to this property is defined as follows:

$$SPEC6(m, b) = Stop \sqcap (reserve.m.b \rightarrow SPEC6'(m, b))$$

$$SPEC6'(m, b) = (cancel.m.b \rightarrow SPEC6(m, b)) \sqcap (take.m.b \rightarrow SPEC6(m, b))$$

$$LIVESPEC = CHAOS(sigma - specsigma) ||| (|||_{m \in MEMBERID} (|||_{b \in BOOKID}\ SPEC6(m, b)))$$

The process $CHAOS$ allows any event in its parameter set to be nondeterministically accepted and refused. In $LIVESPEC$, this corresponds to all events except $reserve$, $cancel$, and $take$. $SPEC6$ allows $reserve$ to be nondeterministically accepted and refused. However, if it is performed, $SPEC6'$ means that $cancel$ cannot be refused until the book is taken by the member.

We perform this refinement check of $LIVESPEC$ using the augmented specification given in Figure 12. However, we can take advantage of an observation made in Section 6.1, namely the independent behaviour of each book. As the reservations (and cancellations) of one book do not affect the other books, we only have to check that the property holds for a single book to show that the property holds for all books.

7. Remarks and Conclusion

Up to now, EB^3 has no tool support for verification of its specifications. One proposed solution presented in [9] is to write a corresponding model-oriented specification in the B Method and prove that this is a refinement of the EB^3 specification. One obvious drawback is that it is difficult to model the flow of control accurately in a purely state-based notation. The solution presented in this paper gives insights into the relationship between EB^3 and CSP || B, which could be used as the basis for a set of translation rules between the two approaches.

This paper demonstrates the applicability of the CSP || B approach in a new application domain, and in doing so we are required to extend the approach. In order to capture the complex interactions and data structures of an information system, a new architecture has been identified which relaxes a constraint in [20] that requires a unique coupling between controllers and machines. We now permit more than one controller to update the state of a B machine, provided that they synchronise on operation calls. Nevertheless, such controllers are still free to perform query operations independently. This extension provides further opportunities for compositional verification in the CSP || B approach. Gener-

alising the results of [20] to accommodate this new architecture is the subject of ongoing research.

This new architecture for CSP || B has implications on the style of the B operations. The $lend$, $renew$ and $reserve$ operations are all well behaved because of the synchronisation between the controller and the association machine. However, the controllers in isolation offer the $take$ event for a set of members who are ready to take the book (*i.e.,* there is no constraint in the CSP definitions to indicate that it must be the member at the head of the reserve queue). For example, the event $take.m2.b1$ could be performed even when $reservSeq(b1) = \langle m1, m2 \rangle$. This information is held in the B machine and needs to be conveyed to the controllers. One way is to augment the $anyReserved$ query to return the member at the head of the reserve queue. This value can then be used later in the $BOOKCTRL$ definition so that $take$ is offered only to this member (as is shown in Figure 11).

$MEMBERCTRL$ precludes the reservation of a book by a member who already has it on loan. This property is stated clearly in the third predicate of the invariant of $Assoc$. Since each operation maintains its machine's invariant, we know that this property is preserved by each action of the information system.

We have seen from the library system example that following the structure of a class diagram allows the user requirements to be taken into account as well as the implicit requirements which are present in the class diagram. Using the CSP || B approach such implicit requirements are made explicit in the definitions of the B association machine. For instance, the fact that two members cannot borrow the same book at the same time is not stated in the requirements. In EB^3 this information is also captured by the synchronisation between books and members over the input events of the loan association but this detail is somewhat obscured within the process expressions.

This work has highlighted the fundamental differences between EB^3 and CSP. Aside from the syntactic distinctions, EB^3 differs from CSP in its 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, semantically they do not distinguish between input and output. However an IS must provide an output for each input received from the environment because the user must be informed of the result of his request. It is very difficult to specify complex output computations using only a process algebra, and this is why mechanisms to handle outputs have been defined in EB^3. In this paper we have focused on ordering constraints but an interesting future direction would be to investigate a mapping of the output model of EB^3 within CSP || B.

References

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

[2] Dupuy, S., Ledru, Y., and Chabre-Peccoud, M.: An Overview of Roz: a Tool for Integrating UML and Z Specifications. *12th Int. Conf CAISE'00*, Stockhom, Sweden, June 2000.

[3] Elmasri, R., Navathe, S. B.: *Fundamentals of Database Systems*. 4$^{\text{th}}$ edition, Addison-Wesley, 2004.

[4] Evans, N., Treharne, H.: *Investigating a File Transfer Protocol Using CSP and B*. Technical Report CSD-TR-03-14, Royal Holloway, University of London, 2003.

[5] Formal Systems (Europe) Ltd.: Failures-Divergences Refinement: FDR2 User Manual 1997.

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

[7] Fraikin, B., and Frappier, M.: Optimizing memory space in the EB^3 process algebra interpreter. In *ICCSSEA 2002, Vol I, Session 4*, 2002.

[8] Fraikin, B., Frappier, M., Laleau, R.: A comparison of EB^3 and B for information system specification. In *StEve Workshop of FM'03*, 2003.

[9] Frappier, M., Laleau, R.: Proving Event Ordering Properties for Information Systems. In *ZB2003*, LNCS 2651, Springer-Verlag, Finland, 2003.

[10] Frappier, M., St-Denis, R.: EB^3 : an Entity-Based Black-Box Specification Method for Information Systems. *Software and System Modeling*, **2**(2), July 2003, pp 134-149.

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

[12] INRIA Rhône-Alpes: CADP (Caesar/Aldebaran Development Package).

[13] Jarke, M., Mylopoulos, J., Schmidt, J.W., Vassiliou, Y.: DAIDA: An Environment for Evolving Information Systems. *ACM Trans. on Information Systems*, **10**(1), January 1992.

[14] 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*, IEEE Computer Society Press, Grenoble, France, September 2000.

[15] Laleau, R., Polack, F.: *Coming and Going from UML to B: A Proposal to Support Traceability in Rigorous IS Development*, In ZB2002, LNCS 2272, Springer-Verlag, Grenoble, 2002.

[16] Marcano-Kamenoff, R.: *Spécification Formelle à Objets en UML/OCL et B: Une Approche Transformationnelle*. PhD thesis, PRISM Laboratory, December 2002.

[17] Meyer, E., Souquières, J.: A Systematic approach to Transform OMT Diagrams to a B specification. In *FM'99*, LNCS 1708 vol. 1, Springer-Verlag, 1999.

[18] Neilson, D., Sorensen, I. H.: *The B-Technologies: a system for computer aided programming*, B-Core (UK) Ltd, 1999.

[19] Schneider, S. A.: *Concurrent and Real-Time Systems: the CSP Approach*, John Wiley 1999.

[20] Schneider, S., Treharne, H.: *CSP Theorems for Communicating B Machines*. In Proceedings of IFM 2004, LNCS 2999, Springer-Verlag, University of Kent, 2004.