Comparing Different Combinations of Event-Based and State-Based Specifications for IS Modelling

Frédéric Gervais, Marc Frappier [*], and Régine Laleau

Département d'informatique
Université de Sherbrooke
Sherbrooke (Qc), Canada, J1K 2R1
{frederic.gervais,marc.frappier}@usherbrooke.ca

- Technical Report No. 18 -

Abstract

Event-based and state-based specifications constitute two potential views of a system model. Formal languages are generally based on only one of these two aspects and therefore can only capture one side of the model. Nevertherless, the other view may bring complementary aspects to the specified model. Coupling of state-based and event-based specifications allows the designer to capture both sides, but integrating two formal languages is difficult. Two complementary views are not always strictly orthogonal, and couplings may be incorrect because of inconsistency or redundancy. The aim of this paper is to show, through a simple example, several approaches using different kinds of couplings of state-based and event-based formal specifications. In particular, we compare their benefits and limits for specifying information systems.

Keywords: State-based specification, event-based specification, coupling, consistency, verification.

[*]Corresponding author. Tel.: +819-821-8000 x62096; fax:+819-821-8200

Contents

List of Tables

List of Figures

1 Introduction

This paper deals with the couplings of event-based and state-based specifications with formal methods. This state of the art is first intented to analyse and compare the advantages and the drawbacks of such approaches to formally specify information systems. More generally, it addresses the main issues about the coupling of formal languages.

1.1 Scope and purpose: Why coupling?

Specifying a system with formal languages is not straightforward. The choice of a specification language is often difficult and depends on the characteristics of the system to be specified. Most often, several languages are good candidates, but none of them really fits well, because some aspects of the requirements would not be explicitely taken into account. In those circumstances, the coupling of specification language may bring a solution.

Coupling of specification languages may be used for different purposes. The main interest is to enhance the description given in a first language. Another is the verification of properties described with a second language. In this paper, verification denotes both theorem proving and model checking. One can also specify a system with several viewpoints, each described with a different language. A coupling not only depends on the syntax of the languages used, but also on their semantics. This is an important issue, since the benefits and the limits of a language with respect to another depend on their semantics. The semantics determines which kinds of properties or requirements may be supported by a language and not by another. For instance, safety properties are easier to prove in state-based semantics, while liveness are easier to express and verify in event-based models. Then, to formally reason on a system model, couplings of specifications require a "common" semantics to compare and to validate the whole system.

Our aim [22] is the formal specification of information systems. An *information system* is a software system that collects and exploits the data of an organization or a corporation [28]. It is mainly composed of a database and software applications and tools that query and modify the database, communicate query results to users, and allow administrators to control and modify the whole system. The use of formal methods to design information systems [19, 29, 32] is justified by the strategic value of data for organisations like banks, insurance companies, high-tech industries or government institutions.

One interesting issue about the design of information systems is the formal specification of transactions. *Transactions* are software applications that manipulate the database through simple query or update operations. On one hand, they must satisfy several integrity constraints in order to preserve the coherence of data. In state-based specifications, integrity constraints correspond to state invariant properties (static properties) that must be preserved by the execution of a transaction. On the other hand, transactions must satisfy scenarios, which define precise transaction ordering properties. Then event-based specifications are well suited to describe such kind of dynamic properties. That is why we are interested by couplings of event-based and state-based specifications in order to specify different kinds of properties (static properties and dynamic properties).

1.2 Which kinds of properties?

We can distinguish several kinds of properties to specify and/or to validate for information systems. *Static* properties are interesting for ensuring the coherence of the system and the integrity of data: they are defined with static data integrity constraints in database systems. In formal languages, such properties can be specified with invariant properties. Operations are then defined with guards or preconditions in order to preserve the invariant. *Dynamic* properties characterize occurrences and ordering properties of operations. This definition includes both safety and liveness properties, as well as properties of the form: "*a*, followed by an arbitrary number of *b*, followed by *c*", where *a*, *b* and *c* are operations.

A *safety* property means that "something bad will never happen". In formal state-based approaches, an invariant property is a safety property for the modelled system. In an event-based specification, a valid sequence of events (often called a valid *trace*) constitutes a safety property of the system as well. A *liveness* property means that "something good will eventually happen". With process algebra like languages, this kind of property can be verified by analysing the valid traces of the system. In state-based specifications, liveness properties are difficult to state and to verify, since they require an explicit description of the possible sequences of state transitions.

An operation (or event) ordering property is also not straightforward to verify for state-based models, compared to event-based ones. For instance, a property like "*a*, followed by an arbitrary number of *b*, followed by *c*" is easy to define in a trace-based language like EB^3 (its syntax will be presented in more details in Sect. 2.2.6):

$$a \;.\; b^* \;.\; c$$

In state-based specifications, operations are generally defined with substitutions on the state variables of the system. To ensure that operations *a*, *b* and *c* satisfy the simple ordering constraint above, the following statements must be satisfied.

- there exists a state in which *a* will be executed,
- the state after having executed *a* corresponds to a state in which either *b* or *c* can be executed,
- each state after having executed *b* corresponds to a state in which either *b* or *c* can be executed,
- there exists a state after having executed *a* or *b* in which *c* will be executed.

Thus, an ordering property is rather difficult to validate by analysing the state-based definitions of operations *a*, *b* and *c*.

Since the use of only one language clearly restricts the scope of the model to be specified, our purposes bring us to study couplings of event-based and state-based specifications in order to evaluate their strengths and weaknesses.

1.3 Outline of the Paper

Even if our primary aim is the specification of information systems, this survey illustrates couplings that are not only restricted to this specific area. In this paper, we show, through a simple example, six approaches using different kinds of couplings of state-based and event-based formal specifications. The paper is organized as follows. In Sect. 2, we introduce the languages used in the remainder of the paper and our example of comparison. The six examples of couplings are shown in Sect. 3. Then, we compare these approaches and show their strengths and weaknesses in Sect. 4. Finally, Sect. 5 concludes this paper with some remarks about couplings and issues related to the design of information systems.

2 Background

For the sake of understanding, we first define the labelled transition systems in Sect. 2.1. Then, the main basic languages used in the remainder of the paper are introduced in Sect. 2.2. Finally, our example and the criteria of comparison are presented in Sect. 2.3.

2.1 Labelled transition systems (LTS)

A *labelled transition system* (LTS) [31] is a transition system where the events are labelled. LTS are often used to define the operational semantics of state-based languages or to represent the behaviour of a system.

The labelled transition system $(A, S, \longrightarrow, R)$ of system P is defined by:

- A is the set of the possible events of system P; A is called the *alphabet* of the LTS,
- S is the set of the possible states,
- $\longrightarrow$ is the state transition relation, such that: $\longrightarrow \subseteq S \times A \times S$,
- R is the set of initial states, such that: $R \subseteq S$ and $R \neq \varnothing$.

Relation $\longrightarrow$ is often defined by a set of inference rules. A transition (q_0, E, q_1) is also denoted by:

$$q_0 \xrightarrow{E} q_1$$

Figure 2.1 shows an example of LTS. States are represented by circles and transitions, by arrows. The initial state is denoted by symbol “$>$”. The values of A, S and R for this example are the follwong ones.

$$A = \{B, C, D, E, F, G, H, I, J, K, L\}$$

$$S = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$
$$R = \{q_0\}$$

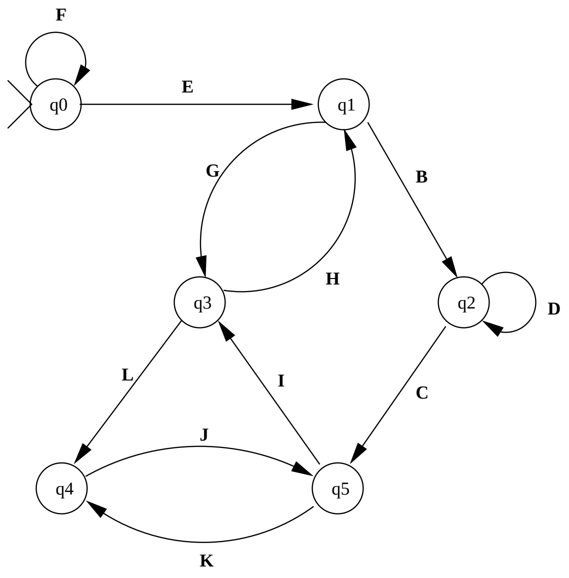

Figure 2.1: An example of LTS.

2.2 Basic languages

Couplings described in this paper use common basic languages. This subsection deals with these languages. Two kinds of languages are mainly coupled in these approaches:

- state-based languages,
- event-based languages.

State-based languages represent a system through two complementary models: the static part of the specification describes the entities of the system and their states, while the dynamic one describes the operations or events that modify the states, inducing state transitions. Invariant properties (or safety properties) ensure the coherence of the system. State-based languages used in the couplings are Z (Sect. 2.2.1), Object-Z (Sect. 2.2.2), B (Sect. 2.2.3) and Event B (Sect. 2.2.4).

Event-based languages represent a system through processes. A process is an independent entity that communicates with other processes and with the environment. Its semantics is represented either by traces, that is sequences of events, or by labelled transition systems, that is graphs where edges are labelled with events to describe the global behaviour of the system. Event-based languages used in the couplings are CSP (Sect. 2.2.5) and EB^3 (Sect. 2.2.6).

2.2.1 Z

The Z language [12] is a state-based formal specification language, based upon the concept of schema. A *schema* is a box containing descriptions using the Z notation. Schemas are used for defining the states of the system, the initial states and the operations.

The static part of the specification provides the definition of the states and of the invariant properties that are preserved during the state transitions. These descriptions are encapsulated in a state schema. For instance, the *Birthday Book* [41] is a well-known example. It describes a system recording birthdays. Types of this sytem are: $[NAME, DATE]$. State schema *BirthdayBook* is defined in Z by:

$$\begin{array}{|l}
\textit{BirthdayBook} \\
\hline
known : \mathbb{P}\, NAME \\
birthday : NAME \pfun DATE \\
\hline
known = \operatorname{dom} birthday \\
\hline
\end{array}$$

The first part of this schema is the declaration of state variables *known* and *birthday*. The second part is the definition of the invariant of the system. Variable *known* must be equal to the domain of variable *birthday*.

The dynamic part of the specification deals with operations, which are relationships between inputs, outputs, before states and after states. For instance, operation *AddBirthday* allows the addition of a birthday:

$$\begin{array}{|l}
\textit{AddBirthday} \\
\hline
\Delta BirthdayBook \\
n? : NAME \\
d? : DATE \\
\hline
n? \notin known \\
birthday' = birthday \cup \{n? \mapsto d?\} \\
\hline
\end{array}$$

State schema *BirthdayBook* represents the states space of the system. The Δ-notation means that the current state (ie, before state) of schema *BirthdayBook* will be changed by executing this operation. The notation $?$ means that variables $n?$ and $d?$ are input parameters of the operation. Predicate $n? \notin known$ is the precondition of the operation. Notation $birthday'$ is used for specifying the new value of state variable *birthday* after the execution of the operation (ie, after state). For instance, the effect of this operation is to add an element to *birthday*. An output parameter is denoted in Z by the symbol !.

The initial state of the system is defined by schema *InitBirthdayBook*:

$$\begin{array}{|l}
\textit{InitBirthdayBook} \\
\hline
BirthdayBook \\
\hline
known = \varnothing \\
\hline
\end{array}$$

Variable *known* is initialized to an empty set.

2.2.2 Object-Z

Object-Z [39] is an extension of the Z language which provides an object-oriented style of specification. In a Z description, it is often difficult to analyse the effects of all the operations on a specific state schema, since operations often act on several state schemas. In Object-Z, the concept of *class* is used for grouping in one schema all the operations depending on it. A class is defined by:

- the list of the inherited classes,
- the definition of types,
- the definition of constants,
- a state schema without name,
- an initial state schema,
- and operation schemas.

For instance, a first-in, first-out (FIFO) waiting line (see [39] for the complete specification) is defined in Object-Z in Fig. 2.2. This class represents sequences of elements of type T defined as parameter of the class. A waiting line has a maximal capacity *max*. Two operations are defined, *Join* and *Leave*. They respectively allow the addition and the suppression of an element from the line. *Inheritance* allows the definitions of the inherited class to become genuine definitions of the class that inherits. It is defined in a Object-Z class by keyword *inherits* (which has not been illustrated here, for the sake of concision).

2.2.3 B Language

The B language [1] is a formal specification language based upon the notion of abstract machine. An *abstract machine* represents a state defined by a static part (thanks to state variables and invariance properties) and modified by a dynamic part (the operations of the system).

The language used for the static part is based on set theory and first order logic. Variables are typed by sets and invariant properties are specified with first-order predicates. The state of the abstract machine can only be modified by operations. They are specified in a generalized substitution language. Table 2.1 presents the different clauses of an abstract machine in B.

For instance, in Fig. 2.3, *Example_Machine* is an example of B abstract machine extracted from [1]. In the body of the abstract operation **change**, the substitution is specified with keywork CHOICE which is not deterministic. Abstract variable x can be substitued either by $x+2$ or $x-2$. A preconditioned operation is always available, but its termination is guaranteed only if it is executed when the precondition is satisfied. If the precondition (PRE) of **change** is not satisfied by the state of the system before its execution, then its effect is not known and the system can diverge. Hence, the operation precondition is used for proving the preservation of the invariant (see clause INVARIANT) when operation **change** is executed.

A B abstract machine may be refined. This activity consists of the derivation of a concrete implementation from an abstract model through successive steps. With refinement, operations become more and more detailed and less and less non-deterministic. For instance, *Example_Machine* is refined by the B component represented in Fig. 2.4. A new concrete state variable is introduced: y. To link the abstract

$Queue[T]$

$max : \mathbb{N}$

$items : \operatorname{seq} T$

$\#items \leqslant max$

$\textsc{Init}$

$items = \langle\,\rangle$

$Join$

$\Delta(items)$

$item? : T$

$\#items < max$

$items' = \langle item? \rangle \frown items$

$Leave$

$\Delta(items)$

$item! : T$

$items \neq \langle\,\rangle$

$items = items' \frown \langle item! \rangle$

Figure 2.2: FIFO waiting line.

Table 2.1: Clauses of a B abstract machine.

Clauses	Description
MACHINE	Name and parameters of the machine
CONSTRAINTS	Definition of the properties of the machine parameters
SETS	List of the abstract sets and definition of the enumerated sets
CONSTANTS	List of the constants
PROPERTIES	Definition of the properties of constants and sets
VARIABLES	List of the state variables
INVARIANT	Definition of the types and the properties of variables
DEFINITIONS	List of abreviations for predicates, expressions or substitutions
INITIALISATION	Initialization of the state variables
OPERATIONS	List of the operations

```
MACHINE Example_Machine
VARIABLES x
INVARIANT $x \in 0..20$
INITIALISATION $x := 10$
OPERATIONS
    change =
    PRE
        $x + 2 \leq 20 \land$
        $x - 2 \geq 0$
    THEN
        CHOICE $x := x + 2$
        OR $x := x - 2$
        END
    END
END
```

Figure 2.3: B abstract machine *Example_Machine*.

```
REFINEMENT Example_Refinement
REFINES Example_Machine
VARIABLES y
INVARIANT
    $y \in 0..10 \wedge$
    $x = 2 \times y$
INITIALISATION $y := 5$
OPERATIONS
    change =
    BEGIN
        $y := y + 1$
    END
END
```

Figure 2.4: B refinement *Example_Refinement*.

and the concrete models, a gluing invariant is defined between abstract variable *x* and concrete variable *y*: $x = 2 \times y$. Operation **change** is now deterministic without any precondition.

B is more than a formal language. It is also a method. The refinement activity associated to a complete tool (Atelier B [10] or B-Toolkit [3]) allows complex systems to be developed in a formal way.

2.2.4 Event B

Event B [2] is an evolution of the B language to specify complex systems by using decomposition and event-based descriptions. In particular, new events can be added during refinement, as opposed to traditional B where new operations cannot be added.

In Event B, specifications describe “closed” event systems, in order to consider a system and its interactions with its environment as a whole. The behaviour is then modelled by events on the system.

An event is defined in Event B by a guard (e.g. a blocking condition that ensures the consistency of the system if the event is executed) and an action described by the generalized substitution language as in B. An event is of the form:

```
ANY $x, y, ...$ WHERE
    $P(x, y, ..., v, w, ...)$
THEN
    $S(x, y, ..., v, w, ...)$
END
```

where $x, y, ...$ are local variables and $v, w, ...$ are constants or state variables of the event system. In this case, P is the guard and S the action. If there is no local variable, an event is then of the following form:

```
SELECT $P(v, w, ...)$
THEN $S(v, w, ...)$
END
```

For instance, *Example_System* is an example of Event B system.

```
    EVENT SYSTEM Example_System
    VARIABLES $x$
    INVARIANT $x \in 0..2$
    INITIALISATION $x := 0$
    EVENTS
        change0to1 $=$
        SELECT
            $x = 0$
        THEN
            $x := 1$
        END;

        change1to2 $=$
        SELECT
            $x = 1$
        THEN
            CHOICE $x := 1$
            OR $x := 2$
            END
        END
    END
```

In the example, after the initialization of the system, only event **change0to1** is enabled to be executed because its guard is satisfied. Once it has been executed, state variable *x* is set to 1, and only one event can be executed: **change1to2**. If *x* remains unchanged by the execution of **change1to2**, then the event can be executed once again; otherwise, no event can be executed.

An event system may be refined. Refinement in Event B not only refines data structures like in B, but also allows new events to be added. However, only new concrete variables can be modified by new events. The state refinement is expressed, like in B, with a gluing invariant. For example, *Example_System* can be refined as follows.

```
    REFINEMENT Example_Refinement
    REFINES Example_System
    VARIABLES $y, count$
    INVARIANT $y = x \land count \in \mathbb{N}$
    INITIALISATION $y, count := 0, 0$
    EVENTS
        change0to1 $=$
        SELECT
            $y = 0 \land count \leq 10$
        THEN
            $y := 1$
        END;

        **change1to2** $=$
        SELECT
            $y = 1 \wedge count \geq 5$
        THEN
            $y := 2$
        END;

        **newcount** $=$
        SELECT
            $count \leq 10$
        THEN
            $count := count + 1$
        END
    END
```

In that refinement, a new state variable *count* is defined. Variable *count* can be modified only by a new event called **newcount** that increases it by 1. In Event B, refinement allows event guards to be strengthened. For instance, the guards of **change0to1** and **change1to2** are now strengthened by a condition on state variable *count*. Moreover, events become less and less non-deterministic with refinement. For instance, *y* is set to 2 in event **change1to2**, while it was set to 1 or 2 in its abstract specification.

2.2.5 CSP

CSP (Communicating Sequential Processes) [25] is a notation for describing concurrent systems.

In CSP, *processes* are independent entities that can communicate through synchronization. A process can execute *events* (also called *actions* when they are labelled). The events describe the behaviour of the system. The set of all the events that a process P can execute is its alphabet, denoted by $\alpha(P)$. In practice, there is no state variable as opposed to Z and B. The simplest CSP process is *SKIP*: it can do nothing.

The prefix operator ($\rightarrow$) allows the description of a process by describing the events that it can execute. Let a be an event and P a process,

$$a \rightarrow P$$

is the process that can execute a and then behaves as process P.

There exist several kinds of choice in CSP. The most simple is denoted by $|$ and only acts on prefixed process expressions. Process:

$$a \rightarrow P \mid b \rightarrow Q$$

can either execute event a and then behave as P, or execute event b and then behave as process Q. Prefixes must be different: $a \neq b$.

Two other choice operators exist in CSP: the external ($\Box$) and the internal ($\sqcap$) choices. Processes $P \Box Q$ and $P \sqcap Q$ can execute all the events that P or Q can execute. The external choice depends on the other processes and is deterministic, while the internal choice is non-deterministic. To define the internal choice, an internal event, denoted by τ, is used in CSP as an event that is not observable by the environment.

Parallel composition is used for describing several processes in parallel. Let P and Q be two processes, then

$$P \;||\; Q$$

is the parallel composition of P and Q. Let A and B be the alphabets of processes P and Q, respectively: $A = \alpha(P)$ and $B = \alpha(Q)$. Process $P \;||\; Q$ can execute all the events from A and B and, moreover, it executes the events that belong to the intersection of A and B in both processes in synchronization. To illustrate this definition, let us first introduce the concept of transition in CSP. Let a be an action, and let R and R' be two processes. Then, expression $R \xrightarrow{a} R'$ denotes the transition from process R to process R' by execution of action a. For instance, if processes P and Q are defined as follows:

$$\begin{aligned} P &= a \rightarrow c \rightarrow P' \\ Q &= b \rightarrow c \rightarrow Q' \end{aligned}$$

where $\alpha(P) = \{a, c\}$ and $\alpha(Q) = \{b, c\}$, then the following transitions are enabled:

$$P \;||\; Q \xrightarrow{a} (c \rightarrow P') \;||\; Q$$

and

$$P \;||\; Q \xrightarrow{b} P \;||\; (c \rightarrow Q')$$

We go on with the first transition. It is now possible to perform action b on process Q, for example:

$$(c \rightarrow P') \;||\; Q \xrightarrow{b} (c \rightarrow P') \;||\; (c \rightarrow Q')$$

Event c is now executed in synchronization by the two processes:

$$(c \rightarrow P') \;||\; (c \rightarrow Q') \xrightarrow{c} P' \;||\; Q'$$

The interleaving operator ($|||$) is similar to the parallel composition operator, but without synchronization.

A process can communicate messages to other processes by using communication channels. An event of the form:

$$c.v \rightarrow P$$

means that a message v is passing on the channel c and then the process behaves as P. To determine if the message is an input or an output, the CSP language uses the decorations “?” and “!”. Process

$$c?x : T \rightarrow P(x)$$

receives input parameter x of type T from channel c and then behaves as process P. Analogously, process

$$c!v \rightarrow P$$

outputs value v through channel c and then behaves as P.

Finally, external and internal choices operators, parallel composition and interleaving may be quantified in CSP.

CSP is supported by some model-checker tools, like FDR [15].

2.2.6 EB^3

The EB^3 ("Entity-Based Black Box") language [19] is a formal specification language created for the specification of information systems. It is based on the notion of trace, on process algebras and on the entity concept from the JSD method [27]. The EB^3 process algebra is inspired from regular expressions, CSP [25], CCS [30] and LOTOS [5]. The language has a trace semantics.

The classic object-oriented notions of class and object are respectively called in EB^3 *entity type* and *entity*. The actions of each entity type constitute the events of the information system. The behaviour of entities and associations are described in terms of traces, that is, sequences of input events. The outputs of the system are defined thanks to attributes, through recursive functions and input-output rules.

An information system is specified with EB^3 in four steps:

1. specify the business model in UML by defining the entity types, the associations, the actions and the attributes of the system.
2. describe the behaviour of the entity types and of the associations with process expressions that define the valid traces of input events.
3. specify a recursive function for each entity attribute of the system.
4. specify input-output rules for defining the outputs of the system depending on the valid input traces.

The EB^3 process algebra includes the following operators. The concatenation operator ($\centerdot$) allows two process expressions P and Q to be concatened (in the list sense). This is denoted by $P \centerdot Q$. Notation $G \Longrightarrow P$ is used for defining a guard G on process expression P. The Kleene closure on the action a, denoted by $a^\star$ means that a is executed a finite, arbitrary number of times. The choice ($|$), parallel composition ($||$) and interleaving ($|||$) operators are defined as in CSP. Let P_1 and P_2 be two process expressions and Δ a set of actions, $P_1|[\Delta]|P_2$ denotes the parallel composition of P_1 and P_2, parameterized by the set of actions Δ. That means that P_1 and P_2 are synchronized only for the actions of Δ. Operators $|$, $|[\Delta]|$ and $|||$ may be quantified. For instance, process expression $|_{i \in 1..3}\, P(i)$ stands for $P(1) \mid P(2) \mid P(3)$.

For instance, let a, b and c be three EB^3 actions. The following EB^3 process expression denotes the behaviour "a, followed by an arbitrary number of b, followed by c":

$$a \centerdot b^\star \centerdot c$$

An interpreter for the EB^3 process algebra, called EB^{3}PAI, can execute process expressions with reasonable efficiency [16].

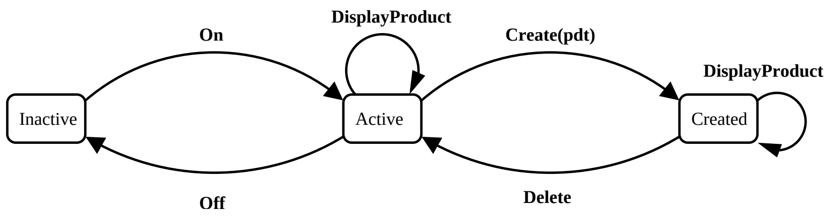

Figure 2.5: Transition system of our example.

2.3 Example presentation and comparison criteria

Our analysis is illustrated by an example. The example consists of a machine that can simply create or delete products. Figure 2.5 is the transition system of this machine. Let us note that a product may be deleted only if it has been created before, and that a new product can be created only if there is no other product. This strong event ordering property must be taken into account in the couplings. Moreover, operation `DisplayProduct`, which can be executed at any moment when the machine is active, outputs the current product or special value "NULL", when the last product has been deleted.

This example allows us to consider both state-based properties and event ordering constraints in the context of IS. Indeed, it satisfies the following requirements:

- some operations, like `Create` and `Delete`, modify the state variables of the system; they give us the opportunity of stating static properties like the preservation of state invariants.
- the system is characterized by a strong event ordering property; hence, we can consider dynamic features like liveness properties.
- operation `DisplayProduct` returns the value of a state variable; it can be considered as a request on an attribute value.

It is actually the example that contains the most IS requirements such that it can be specified by all the approaches described in Sect. 3. However, this example is not entirely convincing to represent all the features of IS, since it does not allow several products to exist at the same time. For each coupling, we try to deal with the creation of several products and, if possible, we show how to specify it.

Through this simple example, we show the benefits and the drawbacks of specifying an information system with formal approaches coupling event-based and state-based specifications. Our criteria of comparison are:

- specific syntax used in the approach: we describe the potential restrictions on the languages used and the requirements difficult to express.
- specification of our comparison example: for each approach, we try to specify the example and respect the transition system of Fig. 2.5.

- consistency: since couplings may introduce inconsistency and redundancy, we show the techniques used to validate the integrated model. We also indicate what is the semantics considered for each approach.
- verification: this point is tightly associated to the semantics. Verification techniques are presented and we also indicate the existence of tools to help the designer.
- refinement: this is an important issue to support a full development process from specification to code.
- adequacy for information systems: we analyse the benefits and the limits of the couplings for specifying information systems.

3 Description of couplings

In this section, six examples of couplings of state-based and event-based languages are presented. We consider three main groups of couplings, according to the aim of each coupling:

1. **Specification in several parts:** In that case, two languages (one is state-based and the other one, event-based) are used to specify the behaviour of the system, like in csp2B (Sect. 3.1) and in CSP || B (Sect. 3.2). Each language provides one part or one aspect of the specification. Hence, both parts are required to form a complete specification. The integration preserves the characteristics of each language, allowing the reuse of existing tools for each language. A connection between the two languages is developped in order to support the coupling.
2. **Creation of a new language:** In that case, a new language integrates in a single semantics constructs of both paradigms (state-based and event-based). It aims at unifying their semantics and at reusing the most relevant operators and techniques. For instance, CSP-OZ (Sect. 3.3) is inspired from CSP and Object-Z, while Circus (Sect. 3.4) is a new language based on Z, CSP, and the refinement calculus.
3. **Verification of properties:** In the last kind of coupling, a complete specification is defined with one language and the second language is used to describe properties which must be proved or verified on the first. For instance, PLTL properties are verified on Event B systems (Sect. 3.5) and EB^3 event ordering properties are proved on B abstract machines (Sect. 3.6).

3.1 csp2B

csp2B [7] belongs to the first group of couplings (specification in several parts). The basic idea of csp2B is to translate a CSP specification (see Sect. 2.2.5) into a B machine (Sect. 2.2.3). A subset of CSP operators are allowed for semantics reasons. The CSP specification is used to specify the ordering of the operations of a B machine; the latter is said to be “conjoined” to the CSP specification. The tool csp2B can translate the CSP specification into a normal B machine which contains the conjoined B machine. The aim is to specify the control in CSP and the data in B. Thus, liveness properties can be verified by existing CSP tools and safety properties are proved by B tools. Refinement is supported in csp2B by translating the CSP specification into B and by using the B refinement relationship. The approach requires both CSP and B to model a system. The tool csp2B provides the one-way link between the two parts of the specification.

3.1.1 Syntax

The CSP part of the model is described as a machine with the following clauses:

- MACHINE: name of the specification
- CONJOIN: name of the "conjoined" machine, that is the B machine which is constrained by the CSP machine
- ALPHABET: list of events
- PROCESS: description of the main process

The following CSP operators are used in the definition of the process: $\rightarrow$ (prefix), $\Box$ (external choice), $||$ (parallel composition), $|||$ (interleaving), *SKIP* (process that does nothing). The first two operators are used without any restriction. Parallel composition is only used for outer expressions. Interleaving is only accepted for multiple instances of the same process. The powerful and the expressiveness of CSP is not entirely explored, since quantification is very limited and the use of operators is restricted.

The B part is modelled as a standard B machine without any restriction on the language.

3.1.2 Specification

The B machine *Example_Act* allows the designer to specify data types and operations:

```
MACHINE Example_Act
SETS PRODUCTS
CONSTANTS Null
PROPERTIES $Null \in PRODUCTS$
VARIABLES Product
INVARIANT $Product \in PRODUCTS$
INITIALISATION $Product := Null$
OPERATIONS
    Create_Act(pdt) =
    PRE $pdt \neq Null$
    THEN
        $Product := pdt$
    END;
```

```
        **Delete_Act** =
        BEGIN
            $Product := Null$
        END;

        $pdt \longleftarrow$ **DisplayProduct_Act** =
        BEGIN
            $pdt := Product$
        END
    END
```

This machine is indeed defined in a standard B-like style. The state of the system is represented by state variable *Product*. This variable represents the existing product, if it exists; otherwise, it is set to special value *Null*, defined as a constant of set *PRODUCTS* (see clauses CONSTANTS and PROPERTIES). State variable *Product* belongs to given set *PRODUCTS*, which represents all the possible products. The invariant of the machine must be preserved by each operation: **Create_Act**, **Delete_Act** and **DisplayProduct_Act**. The first one sets state variable *Product* to the new product *pdt*. Obviously, its input parameter *pdt* must not be equal to constant *Null*. The precondition of this operation verifies this is not the case. Operation **Delete_Act** sets variable *Product* to *Null*, because the product is deleted. Since there is no precondition for this operation, the clause for the precondition is replaced by keyword BEGIN. Finally, operation **DisplayProduct_Act** outputs the existing product, if any, that is *Product*.

Now a CSP-like description is used to constrain some operations of the B machine *Example_Act*. In CSP, the following process would be sufficient to describe the behaviour of the system:

$$\begin{aligned}
&Main \;=\; On \;\rightarrow\; Active\\
&Active \;=\; (ProductCycles \,|||\, Requests)\,\Box\,(Off \rightarrow Main)\\
&Requests \;=\; (DisplayProduct!pdt \;\rightarrow\; Requests)\,\Box\,SKIP\\
&ProductCycles \;=\; Create?pdt \rightarrow Delete \rightarrow Active
\end{aligned}$$

Three sub-processes are defined in *Main*: *Active*, *Requests* and *ProductCycles*. From process expression *Main*, only action *On* is enabled. Then, the process is on state[1]*Active*. From this state, it can choose between two alternatives: either it executes some creations of products, or it executes action *Off* and the system returns to the initial state. In the first case, requests executed by action *DisplayProduct* are interleaved with the life cycles of the products which are created, one after the other (process *ProductCycles*). Thus, process *Main* has the same state transition diagram as in Fig. 2.5. However, we cannot write such an expression in csp2B, since the interleaving operator can only be used as the most outer operator, interleaving purely sequential processes or multiple instances of the same process. Consequently, there exist two alternatives for modelling the behaviour of the system in this approach: either action *DisplayProduct* is not constrained by the CSP process, or it must be explicitely interleaved in the corresponding sequences of actions.

[1]A process expression is often called a “state” in CSP, in reference to the state transition diagram that can be associated to a process expression.

An alternative is to not constrain action *DisplayProduct* with the CSP process. Then, the CSP part of the specification is defined in csp2B by the following CSP machine:

MACHINE *Example*
CONJOINS *Example_Act*
SETS *PRODUCTS*
ALPHABET
$\quad On,\ Off, Create(pdt : PRODUCTS), Delete$
PROCESS $EX\ =\ Inactive$
CONSTRAINS $Create,\ Delete$
WHERE
$\quad Inactive\ =\ On\ \rightarrow\ Active$
$\quad Active\ =\ (Create?pdt \rightarrow Delete \rightarrow Active)\Box(Off \rightarrow Inactive)$
END

This CSP machine defines a process called *EX* to constrain B operations **Create_Act** and **Delete_Act**. Two other processes are defined: *Inactive* and *Active*. Process *EX* has exactly the same behaviour as process *Main* above, except for action *DisplayProduct* that does not appear in *EX*. Let us note that action *Create* is called in the CSP part as a parameterized channel since the corresponding operation from the B machine requires an input parameter. The input type is defined in the ALPHABET clause.

To consider action *DisplayProduct* in the CSP part without using the interleaving operator, then process *Active* must be redefined by:

$$\begin{aligned} Active\ &=\ (Create?pdt \rightarrow Created)\Box(DisplayProduct!pdt \rightarrow Active)\\ &\qquad\qquad \Box(Off \rightarrow Inactive)\\ Created\ &=\ (Delete \rightarrow Active)\Box(DisplayProduct!pdt \rightarrow Created)\end{aligned}$$

The intermediate state after the creation of a product, but before its removal is explicitely defined and action *DisplayProduct* is duplicated each time it can be executed. This solution would be inadequate for long sequences of events. Indeed, let us suppose for example that processes *P* and *Q* are defined as follows.

$$P = a_1 \rightarrow \ldots \rightarrow a_{10} \rightarrow P$$
$$Q = b_1 \rightarrow \ldots \rightarrow b_{10} \rightarrow Q$$

Then, the rewriting of $P\ |||\ Q$ by a process expression without any interleaving operator is very heavy.

3.1.3 csp2B tool

The csp2B tool allows a CSP-like description defined as above to be translated into a B machine. For instance, the translation for the CSP machine that does not constrain action *DisplayProduct* is the following:

MACHINE *Example*
INCLUDES *Example_Act*
SETS *PRODUCTS*; $EXState\ =\ \{Inactive, Active, Active\_1\}$
VARIABLES *EX*
INVARIANT $EX\ \in\ EXState$

```
INITIALISATION $EX := Inactive$
OPERATIONS
    **On** =
    SELECT $EX = Inactive$
    THEN
        $EX := Active$
    END;

    **Off** =
    SELECT $EX = Active$
    THEN
        $EX := Inactive$
    END;

    **Create**($pdt$) =
    PRE
        $pdt \neq Null$
    THEN
        SELECT $EX = Active$
        THEN $EX := Active\_1$
        END
        ||
        SELECT $EX = Active$
        THEN **Create_Act**($pdt$)
        END
    END;

    **Delete** =
    BEGIN
        SELECT $EX = Active\_1$
        THEN $EX := Active$
        END
        ||
        SELECT $EX = Active\_1$
        THEN **Delete_Act**
        END
    END
END
```

All the different states of process *EX* are defined in the enumerated set *EXState*: *Inactive*, *Active* and *Active_1*. The latter is an implicit state of process *EX* which is automatically generated by the tool. It corresponds to the state after executing *Create* and before enabling action *Delete* (i.e., state *Created* in Fig. 2.5). The current state of the process is represented by variable *EX*. Events are translated into guarded operations using the SELECT THEN END form. This allows the B operation to be executed only if the predicate defined in the SELECT clause is true.

The result of the translation also includes (clause INCLUDES) the conjoined B machine of the CSP-like description. This allows the sets and the variables of this B machine to be used and its operations to be called by the B machine specifying the translation. Let us note that the above-mentioned specifications are the result of an automatic translation and therefore could in many cases be simplified. For instance, one can simply specify **Create** with:

```
$\mathbf{Create}(pdt) \ =$
PRE $pdt \ \neq \ Null$
THEN
    SELECT $EX \ = \ Active$
    THEN
        $\mathbf{Create\_Act}(pdt) \ \|$
        $EX \ : \ = \ Active\_1$
    END
END
```

The automatic translation for the CSP machine that constrains action *DisplayProduct* is quite similar, but state *Active_1* is replaced by *Created* and operation **DisplayProduct** is defined by:

```
$pdt \longleftarrow \mathbf{DisplayProduct} \ =$
BEGIN
    SELECT $EX \ = \ Active \ \vee \ EX \ = \ Created$
    THEN $skip$
    END
    $\|$
    SELECT $EX \ = \ Active \ \vee \ EX \ = \ Created$
    THEN $pdt \longleftarrow \mathbf{DisplayProduct\_Act}$
    END
END
```

3.1.4 Verification and refinement

Since CSP-like descriptions are translated into B machines, the semantics of CSP operators is defined upon the semantics of B. Verification of the integrated model is possible by using B techniques and tools. However, the semantics of B does not take into account all the properties of a CSP process. This implies the syntactical restrictions of the CSP part. Since B is a state-oriented language, the generated B specification of the CSP process expression is validated using state-oriented proof techniques like invariant preservation.

Refinement in csp2B is also derived from the B refinement. Figure 3.1 is a summary of the method. CSP-like descriptions are translated by csp2B into B machines and the B refinement is then used to specify the corresponding gluing invariants in CSP.

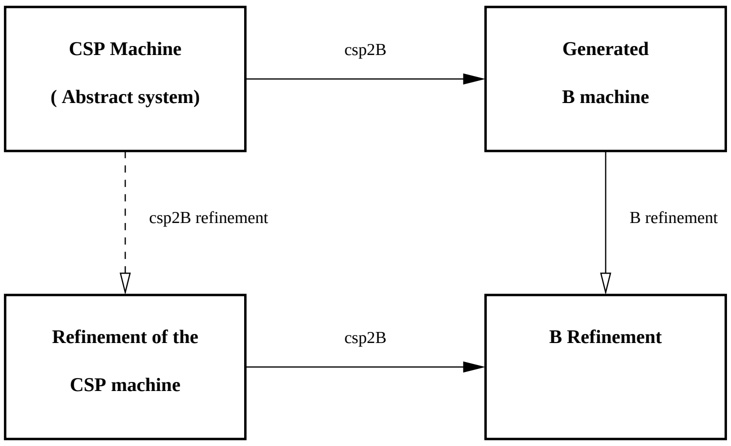

Figure 3.1: Refinement with csp2B tool.

3.1.5 Adequacy for IS

Apart from this particular approach, the CSP language is not very convenient for specifying IS. On one hand, IS data models require state-based structures like abstract data types or class attributes, while the use of state variables in CSP is limited to input-output parameters by means of communicating channels. Actually, variable assignment was originally provided by CSP [25], but it is no longer used in practice.

On the other hand, process is the fundamental concept in CSP, and each process is used to communicate with other processes. So the representation of distinct entities sharing the same properties is a main issue in CSP. In particular, if a process is used to define an entity type, then how are considered the communications between distinct entities of the same type? If a process is defined for each entity, then what is similar between the different entities of the same type and what does distinguish an entity type from another one?

In csp2B, there is no concept of entity type, and CSP's parallel composition and interleaving are only used for outer expressions. However, the interleaving operator can be used for several instances of the same process. Hence, several entities of the same type can be taken into account, if they have exactly the same behaviour and if there is no action that requires a synchronization with the other entities. For instance, to allow several products to exist at the same time, the following process may be defined in csp2B:

PROCESS $Main \ = \ ||| \ pdt.ProductCycles[pdt]$
WHERE
$ProductCycles[pdt] \ = \ Create.pdt \rightarrow Delete.pdt \rightarrow ProductCycles[pdt]$

Since the interleaving must be defined in the main process, the other constraints on actions *On* and *Off* cannot be taken into account. Moreover, the quantification parameter must be specified as an input parameter of actions *Create* and *Delete*.

The B language provides state variables and invariant properties to represent IS data models; however, dynamic properties are difficult to specify and analyse in B. In csp2B, CSP process expressions control the execution of B operations and the csp2B tool translates CSP descriptions into B specifications. Translation from CSP to B is based on the operational semantics of CSP, which is given by a labelled transition system (LTS). This LTS is encoded in B using appropriate data types. Therefore the expressiveness of CSP descriptions is limited in order to allow the translation.

The use of guarded operations in B is not convenient for IS operation specification. Indeed, a guarded operation (with the SELECT THEN END form) can be executed only if the predicate defined in the SELECT clause is true. Otherwise, the operation is infeasible. Nevertheless, an IS operation should always be available and output an error message when its execution is not allowed (for instance, because of a data integrity violation).

This way of integrating CSP and B is twofold. On one hand, the translation into B restricts the kinds of properties to verify on the CSP specification. On the other hand, the use of the B method allows the designer to verify the consistency and some state-oriented properties of the model. Moreover, the translation into B provides the semantics for the whole model and prevents the specifier from introducing inconsistencies in the model.

3.2 CSP || B

CSP || B [38] belongs to the first group of couplings (specification in several parts). A CSP process (Sect. 2.2.5) called *executive controller* is coupled with a B machine (Sect. 2.2.3). The execution of the B operations is then constrained by the CSP controller. This approach can be applied for several B machines in concurrency. Each B machine is then associated to its CSP controller, thus modelling communications between machines through their CSP processes. The idea is to decompose the system into several pairs B machine/CSP controller. The approach provides the sufficient conditions to verify the consistency of the whole specification. Liveness properties are verified by existing CSP tools on the CSP processes and safety properties are proved by B tools. CSP || B requires both the CSP and the B parts to model a system.

3.2.1 Syntax

The syntax for CSP controllers is:

$$\begin{array}{rcl} P & ::= & a \rightarrow P \mid c?x\langle E(x)\rangle \rightarrow P \mid d!v\{E(v)\} \rightarrow P \mid e\mathbf{?}v\mathbf{!}x\{E(x)\} \rightarrow P \\ & & \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \mid \sqcap_{x|E(x)} P \mid \textit{if } b \textit{ then } P_1 \textit{ else } P_2 \textit{ end} \end{array}$$

where a is an event, c an input communication channel, d an output communication channel, e an input/output channel, x represents a variable, v a data value, $E(x)$ is a predicate on x, b is a boolean expression and P a process expression. Most of the CSP operators are allowed: $\rightarrow$ (prefix), $\Box$ (external choice), $\sqcap$ (internal choice). An input x through channel c is denoted by $c?x$. Likewise, an output v through channel d is $d!v$. Predicate $E(x)$ is used for representing the guard (denoted by $\langle E(x)\rangle$) or the assertion ($\{E(x)\}$) of an event.

A guard is a predicate on the input parameter that must be satisfied to execute the event. If the guard is not satisfied, the event is not available. If no event can be executed, then the system is deadlocked. An assertion is a predicate on the output parameter that is assumed to be satisfied when the event has been executed. If the assertion is not satisfied, then the system may diverge. This means that:

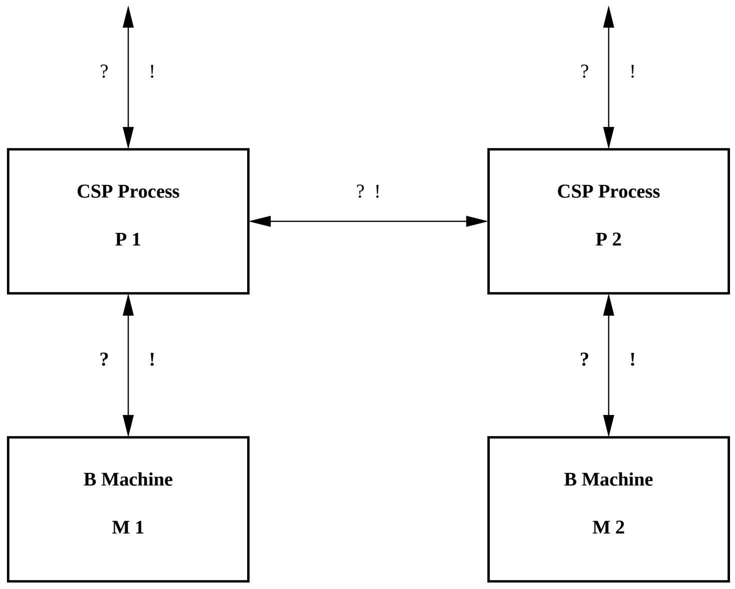

Figure 3.2: Interactions between B machines and CSP processes.

- either the event has aborted; in other words, the event has not been executed as specified, and the system is on a state that is not planned by the specification,
- or an infinite loop of internal events has taken the control of the system.

Parallel composition and interleaving are not considered at this level, since the concurrency is specified between the CSP controllers. Indeed, the CSP controllers are implicitely defined in parallel ($||$). Let $P_1, ..., P_n$ the list of all the CSP controllers, then the global behaviour of the system is specified by: $P_1 \mid\mid ... \mid\mid P_n$. Let us note that the syntax makes a distinction between inter-processes communications (! et ?) and communication between B machines and their associated CSP controllers (**!** et **?**). Figure 3.2 represents the different ways of communication.

The B machine is described as a standard B machine.

3.2.2 Specification

Our example (see Fig. 2.5) is modelled as follows. The B machine called *ExampleData* describes the states and the operations of the model, while the process *ExampleProc* is its associated controller. The links between the CSP controller and its associated B machine are the operations which are identified as communication channels of the CSP process. For instance, process expression $Create\_B\mathbf{?}x\{E(x)\} \rightarrow P$ first executes B operation **Create_B**(x) and then behaves as process P. In this paper, each B operation that corresponds to a CSP channel *op* is denoted by **op_B**. Let us note that this convention is not mandatory in the CSP $||$ B approach and is only used in this paper for the sake of understanding.

Figure 3.3 represents the interactions between *ExampleData* and *ExampleProc*. Events *On* and *Off* are here called from the environment. Operation *Create* is called by the environment in order to specify the

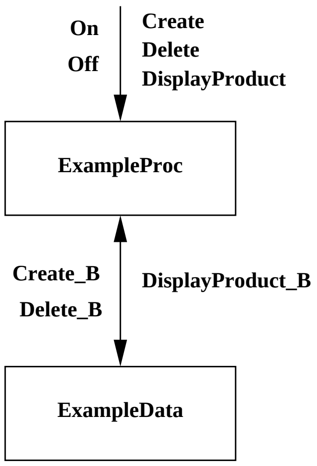

Figure 3.3: Links between the B machine and its associated CSP process.

product *pdt* to be created. Consequently, it constrains B operation **Create_B** in process *ExampleProc*. Likewise, operation *Delete*, that is called by the environment to remove the existing product, constrains operation **Delete_B** from the B machine. Operation *Display-Product* is called by the environment to display the current product, if it exists.

ExampleData is defined as follows. It is the same machine as *Example_Act* of the csp2B approach (Sect. 3.1).

```
MACHINE $ExampleData$
SETS $PRODUCTS$
CONSTANTS $Null$
PROPERTIES $Null \in PRODUCTS$
VARIABLES $Product$
INVARIANT $Product \in PRODUCTS$
INITIALISATION $Product := Null$
OPERATIONS
    **Create_B**$(pdt) =$
    PRE $pdt \neq Null$
    THEN
        $Product := pdt$
    END;

    **Delete_B** $=$
    BEGIN
        $Product := Null$
    END;

    $pdt \longleftarrow$ **DisplayProduct_B** $=$
    BEGIN
        $pdt :\ = Product$
    END
END
```

Like in csp2B (see Sect. 3.1), the interleaving operator is not allowed in the standard approach CSP || B, as described in [43, 42, 38]. However, interleaving and parallel composition have been used in CSP || B in [13] (see Sect. 3.2.4). The CSP controller of the B machine is then specified by:

$$\begin{array}{rcl}
ExampleProc & = & Inactive \\
Inactive & = & On \rightarrow Active(Null) \\
Active(P) & = & (ProductCycles(P) \mid\mid\mid Requests(P)) \\
 & & \Box(Off \rightarrow Inactive) \\
Requests(P) & = & (DisplayProduct\ !pdt\{pdt \in P\} \ \rightarrow \\
 & & DisplayProduct\_B\ \mathbf{!}pdt \ \rightarrow \ Requests(P))\Box SKIP \\
ProductCycles(P) & = & Create\ ?pdt < pdt \neq Null > \\
 & & \rightarrow Create\_B\mathbf{?}pdt \rightarrow ProductCycles1(pdt) \\
ProductCycles1(P) & = & Delete \rightarrow Delete\_B \rightarrow ProductCycles(Null)
\end{array}$$

This process has a parameter P that represents the existing product (if it does not exist, then P is set to *Null*). The initial state is *Inactive*. From state *Active*, the controller describes the accepted sequence of actions. Guards, like predicate $pdt \neq Null$ in expression $Create\ ?pdt < pdt \neq Null >$, are used for defining the enabled values of input parameters of operations. Assertions, like predicate $pdt \in P$ in expression $DisplayProduct\ !pdt\{pdt \in P\}$, are properties that the values of output parameters of operations must satisfy.

From state *Inactive*, only action *On* is enabled. Then, the system is on state *Active*. Once again, two alternatives are enabled. Either the environment asks for the creation of a new product with action *CreateProduct* (decoration "?") or the system returns to state *Inactive* with action *Off*. In the first case, B operation **Create_B** is executed and an intermediate state is defined with *ProductCycles1*. Let us note that parameter P is set to the created product *pdt*. Then, the only available action is *Delete* that constrains B operation **Delete_B**, and the system returns to state *ProductCycles* with $P = Null$. At any moment, action *DisplayProduct* can be interleaved by the environment such that B operation **DisplayProduct_B** can be executed.

3.2.3 Verification and refinement

The main issue of this approach, like in several combinated formalisms, is the consistency of the model. Since the CSP and the B specifications are sometimes redundant, the specifier must verify that the model does not introduce some contradictions. For instance, a controller should not introduce divergences or deadlocks while executing the operations. To avoid divergences, a *control loop invariant (CLI)* is defined in order to guarantee that the CSP controller does not introduce some divergences. Deadlock freedom is verified for each machine-controller pair in concurrency. Since there is only one pair in our example, we do not deal with deadlock freedom in this paper; details can be found in [38].

With operations of the PRE THEN END form, the system would diverge if the preconditions were not satisfied. So the invariant *CLI* ensures that, for each recursive call $S(p)$ in the CSP process, the following statement is verified:

$$CLI \wedge I \Rightarrow [\{BBODY_{S(p)}\}]CLI$$

where the sequence of operations $\{BBODY_{S(p)}\}$ is the result of the translation into B of the process expression related to $S(p)$ and where I is the invariant of the B machine. This ensures that, for each recursive call of the CSP controller, the sequence of the corresponding B operations preserve the control loop invariant *CLI*. The translation into B is defined in [43] for the CSP expressions satisfying the original syntax of CSP || B. Consequently, the B translation of CSP process expressions using the interleaving or the parallel composition operators is not provided.

To illustrate the verification of divergence freedom, let us now consider for our example the CSP process expression without interleaving as given in Sect. 3.1.2. The CLI property is then of the form $c \in 0..2$, where c is a control variable of the system which corresponds to the states 0 (*Inactive*), 1 (*Active*) or 2 (*Created*). Generally, the definition of invariant *CLI* is not straightforward and requires the analysis of CSP process expressions with existing tools.

The CSP || B approach defines proof obligations to check the consistency of the whole model. The implementation of tools to support the verification of both parts of the model is a work in progress. One can use existing tools for verifying the B and CSP parts separately. Refinement of integrated systems in CSP || B is also an open issue. Since B refinement is a data refinement which preserves the signature of the operations, one can refine the B machines, apart from the CSP controllers.

3.2.4 Adequacy for IS

We have already discussed about the drawbacks of CSP for specifying IS (see Sect. 3.1.5). In CSP || B, B machines are associated to CSP processes to control the execution of their operations. The specification is then divided into two parts: CSP and B. Like in the csp2B approach, the CSP language is restricted. In particular, interleaving, parallel composition and quantification are not considered. Contrary to csp2B, a B semantics is not defined to simulate CSP process expressions into B.

CSP || B differs from csp2B in the way of controlling B operations:

- In csp2B, B operations are guarded (i.e. they are of the form SELECT THEN END). Hence, a B operation can be executed only if its guard is satisfied. Operation guards are generated thanks to the translation of the CSP process into B.

- In CSP || B, B operations are preconditioned (i.e. they are of the form PRE THEN END). Thus, a B operation is always available, but the system can diverge if an operation is executed outside its precondition. One must then verify that the precondition is consistent with respect to the CSP specification.

CSP || B provides techniques to verify the consistency of the model, by checking that each pair B machine/CSP controller is divergence-free. It also provides techniques to verify the deadlock freedom of the whole system.

CSP || B needs at least some extensions to specify IS, as described in [13], mainly because of the distinction between controllers and state machines. If each entity type is modelled by a B machine, then the interactions between each entity of the same type cannot be represented in the CSP part, since there is only one controller for each B machine. To consider interactions between entities of the same type, then interleaving (|||), parallel composition (||) and quantification of them must be allowed in CSP controllers. Such enhancements have been used in [13] to specify IS with CSP || B, but they require more consistency-checking obligations than those specific to the original CSP || B approach. In addition, IS may involve several thousands of entities for each kind of entity type; hence, combinatorial explosion of the state space is an important issue of IS model checking.

3.3 CSP-OZ

CSP-OZ [14] belongs to the second group of couplings (creation of a new language). CSP-OZ is a formal, object-oriented, specification language integrating most of the concepts of CSP (Sect. 2.2.5) and Object-Z (Sect. 2.2.2). The idea of CSP-OZ is to identify an Object-Z class with a CSP process. Hence, the operations of an Object-Z class can be controlled by a CSP-like description in CSP-OZ. Both liveness and safety properties can be verified. Refinement is also supported by the definition of simulation rules.

3.3.1 Syntax

CSP-OZ is an extension of CSP$_Z$, a CSP dialect which extends the Z syntax (see Sect. 2.2.1) for describing process expressions and data types. We first present CSP$_Z$ and then CSP-OZ.

CSP$_Z$ The CSP$_Z$ syntax and semantics are detailed in [14]. For the sake of concision, we present only the main features of the language. A Z specification is a sequence of paragraphs described under several schemas (see Sect. 2.2.1). CSP$_Z$ extends Z syntax with two new kinds of paragraphs for describing channels and processes:

$$\mathit{Paragraph}_{CSPZ} \quad ::= \quad \mathit{Paragraph}_Z \mid \mathit{Channel} \mid \mathit{Process}$$

Paragraph *Channel* provides the syntax for specifying communicating channels. A channel is defined as follows:

$$\mathit{Channel} \quad ::= \quad \mathbf{chan}\ \mathit{Name}\ [: \mathit{Expr}]$$

where *Expr* is a classic Z expression which defines data types for input and output parameters. Paragraph *Process* is used for specifying process expressions by using Z constructors and new operators defined in CSP$_Z$. A process is defined as follows:

$$\mathit{Process} \quad ::= \quad \mathit{DefProc} = \mathit{Expr}$$

where *DefProc* is the name of the process. The Z expressions are extended in CSP$_Z$ with new operators for representing process expressions. In particular, *ProcExpr* is the subset of *Expr* that is used to define process expressions. The basic processes are *STOP* (deadlock process), *SKIP* (process that does nothing) and *DIVER* (diverging process). *ProcExpr* expressions use most of the CSP operators: prefix ($\rightarrow$), external ($\Box$) and internal ($\sqcap$) choices, parallel composition ($||$) and interleaving ($|||$). The quantification of the binary operators is allowed.

CSP-OZ The CSP-OZ language extends CSP$_Z$ by adding the concept of class. The syntax adds two new paragraphs to CSP$_Z$ paragraphs:

$$Paragraph_{CSP-OZ} \quad ::= \quad Paragraph_{CSPZ} \mid Class_O \mid Class_C$$

where $Paragraph_{CSPZ}$ represents the previous paragraphs from CSP$_Z$, $Class_C$ is a CSP-OZ class and $Class_O$ an Object-Z class. A CSP-OZ class is an object-oriented view of a process, while an Object-Z class is used for describing data types of the system.

Object-Z classes ($Class_O$) encapsulate the following features:

- a visibility list : items visible from the environment of the class,
- a list of the classes it inherits,
- a list of local definitions,
- a schema defining the state of the objects,
- an initial schema specifying the initial state,
- a list of the operations which can modify the state of the objects.

Let us recall that Object-Z classes can be parameterized; hence, they can be instantiated. An Object-Z class can only inherit from another Object-Z class. CSP-OZ classes ($Class_C$) are similar to Object-Z classes, but they do not have a visibility list and they define an interface and a list of CSP$_Z$ processes. The interface of a CSP-OZ class provides the description of all kinds of communication links of the class, like its methods but also the methods from other objects that the class can use. The list of CSP$_Z$ processes is used for constraining the execution order of the methods.

In order to link methods from an Object-Z class to processes from a CSP-OZ class, the CSP-OZ language introduces several keywords. If *Op* denotes an operation, then:

- `enable`_*Op* denotes the guard of the operation,
- `effect`_*Op* denotes the effect of operation *Op*,
- and `com`_*Op* is used for defining both the guard and the effect of *Op*.

3.3.2 Specification

This approach first needs the definition of two classes: an Object-Z class and a CSP-OZ class. A third class is then used to associate methods to processes.

ExampleData

$Null : PRODUCTS$

$Product : PRODUCTS$

INIT

$Product = Null$

Create

$\Delta(Product)$
$pdt? : PRODUCTS$

$pdt? \neq Null$
$Product' = pdt?$

Delete

$\Delta(Product)$

$Product' = Null$

DisplayProduct

$pdt! : PRODUCTS$

$pdt! = Product$

Object-Z class *ExampleData* specifies the data types and the operations of the system. In this class, state variable *Product* represents the existing product, otherwise it is set to constant *Null*. The type of *Product* and *Null* is *PRODUCTS*. Three operation schemas are specified: *Create*, *Delete* and *DisplayProduct*. The Δ-notation is used if necessary to indicate the state variables that are modified by executing the operation. In our example, only *Product* is concerned in operations *Create* and *Delete*. Operation *Create* requires an input *pdt?*, whose type is *PRODUCTS*, while *DisplayProduct* requires an output *pdt!*. The preconditions and the effects of operations are nearly similar to the previous approaches.

CSP-OZ class *ExampleProc* defines the main CSP_Z process modelling the system. Operations *Create*, *Delete* and *DisplayProduct*, and communicating channels *On* and *Off* are declared in the interface of the class in order to be used by the process:

ExempleProc

`method` $Create[pdt? : PRODUCTS]$
`method` $Delete$
`method` $DisplayProduct[pdt! : PRODUCTS]$
`chan` On
`chan` Off

$Main = On \rightarrow Active$
$Active = (ProductCycles \mid\mid\mid Requests) \Box (Off \rightarrow Main)$
$Requests = (DisplayProduct!pdt \rightarrow Requests) \Box SKIP$
$ProductCycles = Create?pdt \rightarrow Delete \rightarrow Active$

A method like *Create* is an operation that is implemented by the class itself or by one of the classes that inherit from it, while a channel like *On* is an operation that is implemented by other classes, but is used by the class.

By itself, *ExempleProc* only constraints the ordering of the methods defined in its interface, but it does not provide a specification of them. A last Object-Z class called *Example* is then defined to implement the methods of *ExampleProc*:

Example

`method` $Create[pdt? : PRODUCTS]$
`method` $Delete$
`method` $DisplayProduct[pdt! : PRODUCTS]$
`chan` On
`chan` Off
`inherit` $ExempleData, ExempleProc$

`com`$\_Create = Create$
`com`$\_Delete = Delete$
`com`$\_DisplayProduct = DisplayProduct$

This class inherits (keyword `inherit`) from classes *ExampleData* and *ExampleProc*. This specification means that events *Create*, *Delete* and *DisplayProduct* from CSP-OZ class *ExampleProc* have the same guards and the same effects (keyword `com`) than the corresponding operations defined in class *ExampleData*.

3.3.3 Verification and refinement

A graphical editor and a type checker exist for CSP-OZ [21]. Object-Z and CSP specifications are easy to translate into CSP-OZ specifications. However, since CSP-OZ introduces new syntax constructors, some modifications are required to analyse CSP-OZ specifications by tools like CSP's FDR [15] and Object-Z's Moby/OZ [23].

Refinement in CSP-OZ is based on the refinement relations defined in Z and CSP. Fischer shows that if Object-Z classes are data refined following the rules defined in [14], then the associated processes are refined in the CSP sense. Thus, the two notions of refinement are consistent.

3.3.4 Adequacy for IS

CSP-OZ is an integrated formal specification language based on CSP and Object-Z. The main issue with such integrated languages is their usability. They require a new way of specifying systems, new tools to assist the designer and, even if they better capture some interesting properties, they are sometimes difficult to apply. The main benefit of CSP-OZ is the use of a unique language to specify both static and dynamic properties of the system. Roughly speaking, a CSP-OZ class is an Object-Z class with a CSP_Z process expression.

Even if the CSP part is not limited as in the previous approaches, we have already discussed the drawbacks of CSP for specifying IS. In CSP-OZ, the process mainly controls the interactions of the class with other CSP-OZ classes. However, it is possible to specify the functional behaviour between several objects of the same class by using the quantified interleaving operator. For instance, if a parameterized class $Product(pId)$ is defined to represent the set of product entities, then the constraints that must be verified by each entity of this type are specified by a CSP_Z process expression in that class, while the behaviour between the different entities is specified in another class, in which the methods of $Product(pId)$ are visible. Then, the process expression is of the following form:

$$Main = (\ \mid\mid\mid pId : PRODUCTS \bullet Product(pId)\ )\ \mid\mid_{\{m_1,\ldots,m_k\}}\ P$$

where $PRODUCTS$ is the set of product entities, P is another process expression and $\{m_1, \ldots, m_k\}$ is the set of methods on which the synchronization holds.

Object-Z seems to be a convenient formal language to specify IS data model in an object-oriented style. In Object-Z, the attributes of a class are updated only by means of the class methods. The visibility list is therefore an important feature of Object-Z classes, since it determines which methods can be accessed by other classes.

CSP-OZ differs from the other couplings identifying a state-based operation with events, by allowing two kinds of identification in its language:

- a single event view, where an operation is considered as a communicating channel (like in CSP $||$ B, Sect. 3.2, for example);
- a double event view, where each operation is associated to two events: one for reading the inputs, the other one for returning the outputs. For the sake of concision, we have omitted this view in our example.

The latter option is interesting to deal with IS outputs, in particular for the return of an error message, when an operation is not feasible, for instance.

3.4 Circus

Circus [44] belongs to the second group of couplings (creation of a new language). It is a formal language based on Z (Sect. 2.2.1) and CSP (Sect. 2.2.5), that integrates the refinement calculus of [4]. The semantics is inspired from the unifying theory of programming [26]. The idea of Circus is to distinguish state transitions from the communications of the main action system that represents the behaviour of the system. Liveness properties can be verified on the action system, while safety properties are proved on the state schemas. Circus also provides a refinement relation.

3.4.1 Syntax

A Circus specification is structured in terms of processes. A process is represented by a state described by a Z schema and by a behaviour described by an action system. Standard Z schemas are used to represent states, initializations and operations. Communicating channels (similar to CSP's) are defined with the clause **channel**. Processes are specified in the following way:

$$\textbf{process}\ \mathit{Name} \triangleq \mathit{ProcExpr}$$

where *Name* is the name of the process and *ProcExpr* is a process expression of the Circus language. Process expressions in Circus are defined as follows:

$$\textbf{begin}\ \mathit{Paragraph}^{*} \bullet \mathit{Action}\ \textbf{end}$$

where *Paragraph* stands for Z standard paragraphs or for process declarations. Notation * means that a process can be defined with a finite arbitrary number of paragraphs.

Actions in Circus are described by Z schemas, guarded commands and CSP operators. Basic actions are: *Stop* (the action which stops the execution of the process), *Skip* (the action which does nothing) and *Chaos* (the action which diverges). Guarded actions are of the form *Predicate*&*Action*. Operators on actions are: internal ($\sqcap$) and external ($\Box$) choices, parameterized synchronization ($|[\Delta]|$, similar to $||$, but synchronized on actions of the set Δ), interleaving ($|||$), restriction ($\setminus$), sequence ($;$), and communications (of the form $c?x \rightarrow A$ or $c!x \rightarrow A$). Actions can also be recursively defined with μ-expressions from Z of the form:

$$\mu\, N \bullet A(N)$$

where N is an identifier and $A(N)$ is an action expression depending on N. This expression represents the least fixed point such that $N = A(N)$.

Processes of the form *ProcExpr* can be composed with the following operators: internal ($\sqcap$) and external ($\Box$) choices, parameterized synchronization ($|[\Delta]|$), interleaving ($|||$), restriction ($\setminus$) and sequence ($;$). Processes can also be indexed by operator $\odot$ in the following way. Process $i : T \odot P$, where T is a set and P is a process expression, behaves like P, but operates in different channels; each occurrence of a communicating channel c is then indexed by $i \in T$, e.g. c_i.

3.4.2 Specification

Our example (see Fig. 2.5) is specified in Circus as follows. Type *STATES* is an enumerated set containing the three states of the system:

$$STATES = Stop \mid Active \mid Created$$

A communicating channel called *In* allows the declaration of the new product to be created. Channel *Out* is used to output the current product.

$$\mathbf{channel}\ In, Out : PRODUCTS$$

A constant *Null* is defined as follows:

$$\left|\ Null : PRODUCTS\right.$$

The main process is called *Machine*:

$$\mathbf{process}\ Machine \triangleq$$
begin …/* Z schemas and action system to follow */

The first two Z schemas, *State* and *StateInit*, define the state and the initialization of the system respectively. They are similar to the state schemas defined in CSP-OZ (Sect. 3.3). An additional state variable is defined however; thus, variable *state* denotes the state of the system.

State

$$\begin{array}{|l}
state : STATES \\
product : PRODUCTS
\end{array}$$

StateInit

$$\begin{array}{|l}
State \\
\hline
state' = Stop \\
product' = Null
\end{array}$$

The next schemas represent the operations: *On*, *Off*, *Create*, *Delete* and *DisplayProduct*.

On

$$\begin{array}{|l}
\Delta State \\
\hline
state = Stop \\
state' = Active \\
product' = product
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Off}} \\ \hline
\Delta \mathit{State} \\ \hline
\mathit{state} = \mathit{Active} \\
\mathit{state}' = \mathit{Stop} \\
\mathit{product}' = \mathit{product} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Create}} \\ \hline
\Delta \mathit{State} \\
\mathit{pdt}? : \mathit{PRODUCTS} \\ \hline
\mathit{state} = \mathit{Active} \ \wedge \ \mathit{pdt}? \neq \mathit{Null} \\
\mathit{state}' = \mathit{Created} \\
\mathit{product}' = \mathit{pdt}? \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Delete}} \\ \hline
\Delta \mathit{State} \\ \hline
\mathit{state} = \mathit{Created} \\
\mathit{state}' = \mathit{Active} \\
\mathit{product}' = \mathit{Null} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{DisplayProduct}} \\ \hline
\mathit{pdt}! : \mathit{PRODUCTS} \\ \hline
\mathit{pdt}! = \mathit{product} \\ \hline
\end{array}$$

An intermediate action, *ProductCycles*, represents the life cycles of the products which are created:

$$\begin{array}{l}
\mathit{ProductCycles} \triangleq \\
\mu X \bullet \mathit{In}?\mathit{pdt} : \mathit{PRODUCTS} \ \rightarrow \ ((\mathit{pdt} \neq \mathit{Null} \ \& \ \mathit{Create}; \ \mathit{Delete}) \\
\qquad \Box(\mathit{pdt} = \mathit{Null} \ \& \ \mathit{Skip})); \ X
\end{array}$$

Action *ProductCycles* first inputs one product *pdt* by channel *In*. If *pdt* is different from *Null*, then Z operations *Create* and *Delete* are consecutively executed; otherwise, nothing is done and the processus is iterated. This first intermediate action avoids a deadlock due to an undesired input value of parameter *pdt*. Another intermediate action, *Requests*, specifies an arbitrary finite number of executions of Z operation *DisplayProduct*:

$$\begin{array}{l}
\mathit{Requests} \triangleq \\
\mu Y \bullet (\mathit{DisplayProduct}; \ \mathit{Out}!\mathit{pdt} \ \rightarrow \ Y)\Box \mathit{Skip}
\end{array}$$

The result of operation *DisplayProduct* is output by communicating channel *Out*. Finally, the main action, described just after symbol •, defines the global behaviour of the system. This action uses all the above-mentioned Z paragraphs and actions.

> /* List of Z schemas and definitions of the intermediate actions */
> •
> $\mathit{StateInit};\ (\mu\, V \bullet (\mathit{On};\ ((\mathit{ProductCycles} \mathbin{|||} \mathit{Requests}) \Box (\mathit{Off};\ V))))$
> **end**

This system denotes the following behaviour. The initialization schema *StateInit* is first executed. Then, a μ-expression defines the action recursively. After having executed action *On*, there are two alternatives: either the process executes action *ProductCycles* interleaved with action *Requests*, or it executes *Off* and the system is ready to be actived once again.

3.4.3 Verification and refinement

The semantics is based on the unifying theory of programing [26]. In [44], Woodcock et al. use this theory to define a semantics for Circus, described with the Z language. A parser already exists for Circus descriptions. The development of a model-checker, based on CSP model-checher FDR [15] and on Z theorem prover ProofPower, in order to analyse Circus specifications, is currently a work in progress [9].

Circus defines a refinement relation that integrates both action, process and data refinement. In [37] and [8], refinement rules are defined in order to decompose a specification into several sub-components and then to verify all these components. These rules allow the definition of a strategy to derive a formal specification into a distributed system. Circus is the only approach in this paper that provides a refinement for integrated formal specifications, but its use is very difficult, since several refinement rules are defined and, up to now, no tool is available. The creation of tools should clearly improve this refinement strategy for combined systems.

3.4.4 Adequacy for IS

The Circus language has been created to unify refinement calculi from Z, CSP and Action Systems. Like for CSP-OZ (see Sect. 3.3), a new syntax and a new semantics have been defined. For the same reasons, the application of Circus requires a new way of specifying systems. In particular, since the semantics is unified, Circus requires new verification techniques and tools to analyse the specification.

Nevertheless, since the definition of an unified theory for refinement is the motivation behind Circus, the language dissociates state transitions from action systems to improve the refinement. Consequently, state-based properties can be checked separately. Then, action systems are mainly based on event ordering properties since the state transitions are encapsulated in the Z schemas. However, the action guards require an analysis in order to avoid deadlocks.

In order to refine a process into several processes in parallel, the expressiveness of actions is limited; in particular, interleaved actions of the same process share the same state space, but cannot modify the same state variables. So the concurrent behaviour between several entities of the same type is difficult to take into account by using the quantified interleaving operator on actions. However, the interleaving operator on processes, combined to operator $\odot$, can be used to represent the behaviour of entity type product. Indeed, if process *Product* is defined to specify the behaviour for a product, then expression $||| \; pdt : PRODUCTS \odot Product\lfloor pdt \rfloor$ represents the interleaving of several instances of *Product* indexed by *pdt*. On the other hand, an object-oriented extension of Circus, called OhCircus [9], has been defined. Once again, some restrictions exist, in particular for the method calls, in order to preserve the monotonicity property of refinement.

Actually, the main benefit of Circus is the definition of a refinement relation based on the unified semantics. Indeed, Circus is the only approach in this survey that provides a refinement for combinated specifications. The refinement strategy consists of iterated decompositions of processes. Circus refinement includes both process, action and data refinements. In particular, if a process is decomposed into several sub-processes in parallel, and if a state schema must be used by both sub-processes, then the schema is also split and communication channels are defined in order to share the state components of the original schema. Such a strategy could be used to address IS issues, like the modelling of concurrent behaviour or the implementation of a distributed system.

3.5 PLTL and Event B

PLTL-Event B belongs to the third group of couplings (verification of properties). Systems are specified with Event B (Sect. 2.2.4), which is an event-based extension of the B language (Sect. 2.2.3). The idea of this approach [11] is to add PLTL [36] temporal properties to an Event B specification. Proof techniques used in B are combined with model checking techniques to verify these properties on the model. Safety properties are proved as usual in Event B. The approach also provides a technique to reformulate PLTL properties on the refinement of an Event B system.

3.5.1 Syntax

Two existing languages are used in this approach: Event B and PLTL.

Event B An event system is described in B by a machine including these clauses:

- CONSTRAINTS : predicates of the machine,
- SETS : abstract and enumerated sets of the machine,
- INVARIANT : invariant properties of the system,
- INITIALISATION : initialization of the system,
- EVENTS : description of the events.

Dynamic constraints can be defined in B with dynamic invariants and modalities. A dynamic invariant is in such a form:

DYNAMICS $\mathcal{P}(V, V')$

where $\mathcal{P}(V, V')$ is a predicate that characterizes how the variables V may evolve to V' by the call of an event. A modality is either:

SELECT P Leadsto Q [WHILE $e_1, ..., e_n$]
INVARIANT J
VARIANT V
END

or:

SELECT P Until Q [WHILE $e_1, ..., e_n$]
INVARIANT J
VARIANT V
END

“P Leadsto Q” means that if P is true, then the sequence of events in the clause WHILE eventually leads to Q is true. “P Until Q” means that “P Leadsto Q” is satisfied and that P is true until Q becomes true.

PLTL The temporal logic introduced by [35] allows the description of dynamic properties that cannot be defined in B. Temporal logic operators used in this approach are: $\bigcirc$ (next state), $\Diamond$ (eventually), $\Box$ (always), Θ (previous state), $\mathcal{U}$ (until), $\mathcal{S}$ (since) and $\mathcal{W}$ (unless).

3.5.2 Specification

The following event system specifies the machine of Fig. 2.5:

EVENT SYSTEM *Example*
SETS $PRODUCTS; \; EXState = \{Inactive, Active, Created\}$
CONSTANTS $Null$
PROPERTIES $Null \in PRODUCTS$
VARIABLES $Product, \; Output, \; EX$
INVARIANT $Product \in PRODUCTS \wedge Output \in PRODUCTS$
$\quad \wedge \; EX \in EXState$
INITIALISATION
$\quad Product := Null \parallel Output := Null \parallel EX := Inactive$
EVENTS
$\quad$ **On** $=$
$\quad$ SELECT $EX = Inactive$
$\quad$ THEN $EX := Active$

```
    END;

    **Off** =
    SELECT $EX = Active$
    THEN $EX := Inactive$
    END;

    **Create** =
    SELECT $EX = Active$
    THEN
        ANY $xx$ WHERE $xx \in PRODUCTS \wedge xx \neq Null$
        THEN
            $EX := Created \parallel$
            $Product := xx$
        END
    END;

    **Delete** =
    SELECT $EX = Created$
    THEN
        $EX := Active \parallel$
        $Product := Null$
    END;

    **DisplayProduct** =
    SELECT $EX \in \{Active, Created\}$
    THEN
        $Output := Product$
    END
END
```

Contrary to the standard B approach, it is not possible to determine which product is effectively created, since events have no input and no output. In Event B, systems are considered as “closed”; hence, inputs and outputs are modelled as internal state variables of the event system. Through events **Create** and **Delete**, the user can only “observe” the state change of variable *Product* but cannot specify which products he wants to create. Nevertheless, the guard ensures that a product must be deleted before a new product can be created. Analogously, a state variable *Output* represents the output of event **DisplayProduct**.

Dynamic properties are specified in PLTL. For instance, the system is active while it does not stop and the only enabled next state after *Inactive* is *Active*:

$$\Box(((EX = Active)\mathcal{U}(EX = Inactive))\wedge$$
$$(EX = Inactive \Rightarrow \bigcirc(EX = Active)))$$

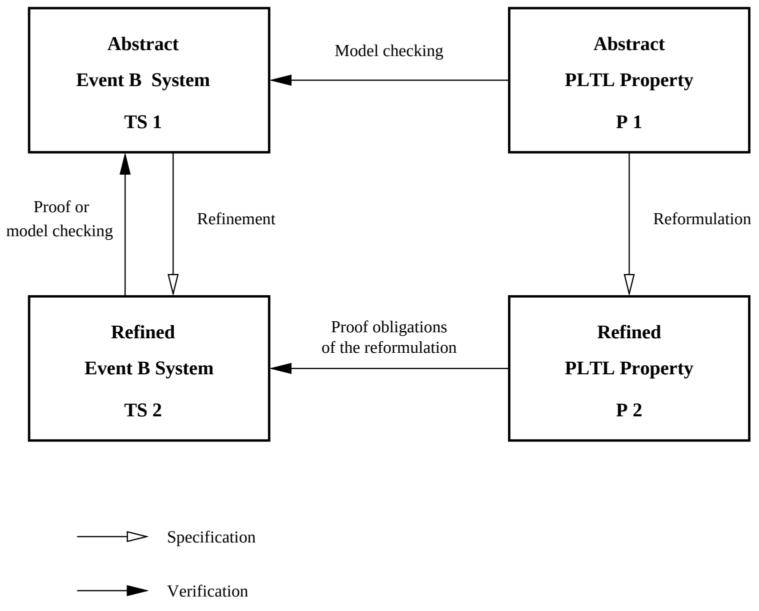

Figure 3.4: Verification of PLTL properties in a B model.

3.5.3 Verification and refinement

Temporal properties are preserved by refinement of Event B specifications. In his Ph.D. thesis [11], Darlot introduces reformulation rules to refine PLTL temporal properties. He defines a method combining theorem proving and model checking (see Fig. 3.4).

Firstly, PLTL temporal property $P1$ is defined on abstract system $TS1$. This property is verified by model checking. If abstract system $TS1$ is refined by concrete system $TS2$ by usual techniques of Event B, then property $P1$ is preserved by refinement. The approach consists in rewriting $P1$ with reformulation rules into a new property $P2$, which is correct with respect to $TS2$ if the conditions associated to the rules are satisfied.

3.5.4 Adequacy for IS

Event B is a state-based language using guarded operations to represent action systems. The use of guards is not well adapted for IS specification, because each IS operation requires an answer, possibly an error message. Actually, guarded operations cannot be executed if their guard is not satisfied.

In Event B, systems are “closed” and events (as opposed to the state-based “operations” from B in Sect. 2.2.3) have neither input nor output parameters. The specification of IS with closed models seems to be a main issue, since an IS can involve multi-users, transactions with a large number of inputs, distributed database systems, etc. However, Event B refinement allows new events to be added and existing systems to be decomposed into concurrent systems. Moreover, a refinement step is used to split the model in two parts: the system software and its environment. Hence, the future input and output parameters of an event can be modelled as internal state variables of the event system. Hence, IS specification with Event B is probably a big challenge, but it should be possible.

The creation of several products in the example can be taken into account in Event B. In that case, state *Created* is no longer considered in set *EXState*. Hence, $EXState = \{Inactive, Active\}$. State variable *Product* is defined as a subset of *PRODUCTS*. Events **Create** and **Delete** are specified as follows.

```
**Create** =
SELECT $EX = Active \wedge Product \neq PRODUCTS$
THEN
    ANY $xx$ WHERE $xx \in PRODUCTS - Product \wedge xx \neq Null$
    THEN
        $Product := Product \cup \{xx\}$
    END
END;

**Delete** =
SELECT $EX = Active \wedge Product \neq \varnothing$
THEN
    ANY $xx$ WHERE $xx \in Product$
    THEN
        $Product := Product - \{xx\}$
    END
END;
```

The new guards allow several products to exist at the same time. In particular, the guard on local variable *xx* does not allow an existing product to be created by event **Create**. Similarly, the guard on *xx* prevents event **Delete** from removing a product that does not exist.

The approach presented in this section allows PLTL properties to be verified on Event B systems. B proof techniques are combined with model checking techniques: safety properties are then proved as usual in Event B, while PLTL properties are rewritten for each refinement step in order to take the Event B data refinement into account. The use of a temporal logic like PLTL may be very useful to verify dynamic properties of IS. The ability of refining such properties to take into account the new state variables and events introduced by the refinement of the event system is the main benefit of this approach.

3.6 EB^3-B

EB^3-B belongs to the third group of couplings (verification of properties). In this approach [18], the event ordering properties of a B machine (Sect. 2.2.3) are described with EB^3 (Sect. 2.2.6), a trace-based language, inspired from CSP [25], CCS [30] and LOTOS [5] and created for the specification of information systems. An EB^3 specification can be refined into a B machine [18] by using the B refinement relation. The idea behind this approach is to use EB^3 to specify the functional behaviour of the system, while B is used to verify the data model. By proving that the B model is a refinement of the EB^3 one, then the event ordering properties described in EB^3 are satisfied by the B operations. Tools supporting the B method are used for the B part, while there is no tool for EB^3.

3.6.1 Specification

The system of Fig. 2.5 is specified in B as follows:

```
MACHINE $Product$
SETS $PRODUCTS$; $STATES = \{0, 1, 2\}$
CONSTANTS $Null$
PROPERTIES $Null \in PRODUCTS$
VARIABLES $Products, State$
INVARIANT $Products \in PRODUCTS \land State \in STATES$
INITIALISATION
    $Products := Null \parallel State := 0$
OPERATIONS
    $res \longleftarrow$ **On** $=$
    IF $State = 0$
    THEN $State := 1 \parallel res := $ ‘ok’
    ELSE $res := $ ‘error’
    END;

    $res \longleftarrow$ **Off** $=$
    IF $State = 1$
    THEN $State := 0 \parallel res := $ ‘ok’
    ELSE $res := $ ‘error’
    END;

    $res \longleftarrow$ **Create**$(pdt)$ $=$
    PRE $pdt \in PRODUCTS$
    THEN
        IF $pdt \neq Null \land State = 1$
        THEN
            $Products := pdt \parallel State := 2 \parallel$
            $res := $ ‘ok’
        ELSE $res := $ ‘error’
        END
    END;

    $res \longleftarrow$ **Delete** $=$
    IF $State = 2$
    THEN
        $Products := Null \parallel State := 1 \parallel$
```

```
            $res\ :\ =\ $‘ok’
        ELSE $res\ :\ =\ $‘error’
        END;

        $pdt, res \longleftarrow$ **DisplayProduct** $=$
        IF $State\ \in\ \{1,2\}$
        THEN
            $pdt\ :\ =\ Product\ \|res\ :\ =\ $‘ok’
        ELSE
            $pdt\ :\ =\ Product\ \|res\ :\ =\ $‘error’
        END
    END
```

Operations are defined on a IF THEN ELSE END form to represent the deterministic transitions of the system and preconditions are used as typing constraints on the input parameters. All the operations have at least one output parameter, *res*, that indicates whether the operation has been successfully executed (‘*ok*’) or not (‘*error*’). Thus, an operation is always available and is successfully executed when the IF predicate is satisfied. Otherwise, an error message is output. Our example is represented with two state variables, *Products* and *State,* and five operations are defined: **On**, **Off**, **Create**, **Delete** and **DisplayProduct**.

In EB^3, the input-output behaviour of the system is specified as follows. The signature of the events of the system are:

```
On() : void
Off() : void
Create(pid : PRODUCTS) : void
Delete() : void
DisplayProduct() : (pdt : PRODUCTS)
```

The EB^3 actions correspond to the homonymic operations from the B machine. Output parameter *res* is not explicitely specified in the signature. Actions `On`, `Off` and `Delete` have neither input nor output parameter (`void`). The inputs of the system are the events received by the system. EB^3 process expressions specify the functional behaviour on the inputs of the system. Only one entity type is considered in our example, that is `Product`, and the following process expression is defined to describe its behaviour:

```
Product(pid : PRODUCTS) =
   Create(pid) . Delete
```

This specification denotes the sequence of the two events. The global behaviour of the system is specified by:

```
main =
(
   On .
   (
       ( | pid : PRODUCTS : Product(pid) )^*
```

```
        |||
        DisplayProduct^*
    ) .
    Off
)^*
```

Process `main` represents the set of the valid input traces (ie, the sequences of input events) of the system. It denotes the transition system described in Fig. 2.5. Let us recall that the Kleene closure (*expr**) denotes an arbitrary number of executions of expression *expr*.

The outputs of the system are evaluated from attribute definitions in answer to an input event. In EB^3, they are specified with recursive functions on the current trace of the system. For instance, the current product is output by the following recursive function:

```
CurrentProduct(trace : VALID-TRACE) : PRODUCTS =
   match last(trace) with
      null -> null
      Create(pid) -> pid
      Delete -> null
      _ -> CurrentProduct(front(trace))
```

where `VALID-TRACE` is the set of the valid input traces accepted by process `main`. Standard list operators are used, such as `last` and `front` which respectively return the last element and all but the last element of the list; they return the special value `null` when the list is empty. When a recursive function is executed, then the last input event received by the system is compared to the different patterns indicated before symbol `->`; the first pattern that holds is the one executed and the corresponding expression just after symbol `->` is computed. If there is no match with a pattern, then the function is recursively called with the first elements of `trace` except the last one; this case corresponds to the last line with symbol '_'. Symbol `null` matches with the empty trace. An input-output rule is defined in order to associate input event `DisplayProduct` to recursive function `CurrentTrace`:

```
Rule R1 :
Input DisplayProduct
Output CurrentProduct(current-trace)
EndRule
```

Hence, each time that `DisplayProduct` is a valid input event of the system, then recursive function `CurrentProduct` is called to evaluate the current product.

3.6.2 Verification and refinement

The verification of the dynamic property expressed in EB^3 by the B machine is proved by first translating the EB^3 process expression into a B machine and then by proving that the latter is refined by the former. For instance, the translation of the previous EB^3 process expression is of the following form:

MACHINE *TranslationEB*3
SEES ...

```
VARIABLES $t$
INVARIANT $t \in \tau(main)$
INITIALISATION $t := []$
OPERATIONS
    $res \longleftarrow$ **On** $=$
    IF $t \leftarrow On \in \tau(main)$
    THEN $t := t \leftarrow On \,\|\, res := \text{‘}ok\text{’}$
    ELSE $res := \text{‘}error\text{’}$
    END;
...
END
```

State variable *t* represents the current trace of the system. The set of valid input traces is represented by $\tau(main)$. The invariant ensures that the current trace is always valid. Each operation corresponds to an action of the process expression. Clause SEES provides the definition of set $\tau(main)$ and of function *CurrentProduct* in another B machine. Expression $t \leftarrow u$ denotes the right append of element *u* to sequence *t*. For the sake of concision, only operation **On** has been translated. For each operation *op*, the pattern is the following. The IF predicate verifies that $t \leftarrow op$ is a valid input trace. If so, then trace *t* is augmented with *op* and message ‘*ok* ’ is returned. Otherwise, an error message is sent.

The verification of the event ordering property consists in proving that machine *Product* refines machine *TranslationEB3* in the B semantics. The main issue is the definition of a gluing invariant between the abstract and the concrete state spaces. Concrete state variable *Products* represents the set of existing products. This set is represented in the EB^3 part of the specification by the output of recursive function `CurrentProduct` applied to the current trace, *t*. Then, the gluing invariant between *Products* and *t* is therefore:

$$Products = \{\mathtt{CurrentProduct(t)}\}$$

By analysing the LTS (see Sect. 2.1) of the example, the following gluing invariants are determined for state variables *State* and *t*:

$$\begin{array}{lcl} State = 0 & \Leftrightarrow & t = [] \lor last(t) = Off \\ State = 1 & \Leftrightarrow & last(t) \in \{On, Delete, DisplayProduct\} \\ State = 2 & \Leftrightarrow & last(t) \in \{Create, DisplayProduct\} \end{array}$$

Up to now, no refinement relationship is defined in EB^3.

3.6.3 Adequacy for IS

Contrary to CSP, EB^3 has been specially created for IS specification. Hence, the specification of the inputs and outputs of the system is divided into two parts. Process expressions represent the valid input traces of the IS, while outputs are computed from valid EB^3 traces. In particular, EB^3 process expressions are close to regular expressions, with the use of the sequence operator and the Kleene closure. For instance, the quantified interleaving operator is used to consider several products at the same time; the main process then becomes:

```
main =
(
   On .
   (
       ( ||| pid :  PRODUCTS : Product(pid)^* )
       |||
       DisplayProduct^*
   ) .
   Off
)^*
```

where the choice operator has been changed to an interleaving.

The approach presented in this section uses both EB^3 and B, in order to verify event ordering properties on the B model. The use of EB^3 to specify event ordering properties is interesting, because it would be difficult to specify this kind of property with a state-based language like B. Nevertheless, there is no verification techniques and tools in EB^3 to verify liveness properties on the EB^3 model. Since the EB^3 process expressions are then used to verify dynamic properties on the B model, one should first verify that the EB^3 specification itself is correct with respect to the requirements. Thus, the implementation of a model checker like CSP's FDR [15] would clearly improve the approach.

The use of preconditioned operations of the IF THEN ELSE END form in the B part is radically different from csp2B and CSP || B's specification styles. Preconditions are here used for typing constraints on the input parameters and IF THEN ELSE END substitutions distinguish valid calls from invalid calls to the operations. Hence, the ELSE part represents the error message returned by the IS when an operation is invalid. Moreover, IF THEN ELSE END operations guarantees that the IF predicate cannot be strengthened or weakened by refinement. Consequently, by proving the refinement between EB^3 and B, the LTS of EB^3 and B operations are exactly equivalent. Actually, the main benefit of this approach consists in proving the equivalence between the two parts.

4 Comparison of approaches

In Sect. 3, we have introduced the different couplings by classifying them according to their objectives. Thus, we have considered three groups: specification in several parts, creation of a new language and verification of properties. In this section, we are now interested in the way the couplings are defined. We can classify the approaches by defining three levels of coupling between the specifications, according to their underlying semantics:

1. **Unification:** This level of coupling is the strongest one, since it involves the definition of a new syntax and of a new semantics; in other words, a new language is defined to reuse the main operators from the composed languages and to unify their semantics. The new language benefits

from new interesting concepts like integrated refinement or verification rules, but the existing tools or the existing specifications cannot be reused. Two examples in this paper belong to this category: CSP-OZ and Circus.

2. **Embedding:** This level of coupling consists in defining the subset of one language in the semantics of the other one; it can be considered as a "vertical combination" of specification languages. At this level, some existing notations are reused, but the redefined subset of language may loose much of its expressiveness. If a specification from one language is translated into the other one, some information may be lost because of the restrictions of the target language. This kind of coupling is also used for verifying some temporal or event ordering properties. Three examples belong to this level of coupling: csp2B, PLTL and Event B, and EB^3-B.

3. **Juxtaposition:** This level of coupling allows different views of the same system to be expressed; it can be considered as a "horizontal combination" of specification languages. The link between the semantics is mainly an identification of the structure of one language to a structure of another language. Hence, it is rather difficult to understand the whole model. This level implies many issues like redundancy or inconsistency of specifications. CSP || B belongs to this category of coupling.

The complementarity between the descriptions is very important in the coupling of specifications. If the two languages are orthogonal in specification styles, then the whole model is difficult to understand and/or to verify. If two languages have the same capabilities of representation and expressiveness, the specifications can be redundant or inconsistent. On the other hand, specification errors can be found by stating the same properties twice, in two different languages, and then by proving that the two models are consistent.

When some subset of one language is defined in the semantics of the other one, some interesting features of the first language semantics are lost by translation. For instance, it is difficult to define the event ordering properties in B, whereas it is easy in CSP. The translation from CSP to B in the csp2B approach reduces the expressiveness of the CSP descriptions to be translated. When just an identification between structures is made, the verification requires more analysis to consider the whole system and to avoid inconsistencies between specifications.

4.1 Synthesis of the presented approaches

Tables 4.1 and 4.2 sum up the characteristics of the different approaches. For each coupling, we present in these tables:

- its name;
- the section of the paper in which it is presented;
- the languages it combines or from which it is inspired;
- the original aim of its authors;

Table 4.1: Comparison of couplings of state-based and event-based specifications - part 1.

Approach	csp2B [7]	CSP || B [38]	CSP-OZ [14]
Section	3.1	3.2	3.3
Languages	CSP B	CSP B	CSP Object-Z
Goal	tool from CSP to B	CSP process = controller of B operations	new language
Semantics	operational with LTS	denotational stable failures failures-divergences	operational and denotational
Verification	m.c. in CSP, proof in B	m.c. in CSP, proof in B	model checking
Refinement	B refinement	no	data refinement
Tools	FDR, B tools, csp2B	FDR, B tools	type-checker, m.c. to develop
Readability	$+$	$+$	$-$
Adequacy for IS	$-$	$-$	$+$
Classification: aim	specification in several parts	specification in several parts	creation of a new language
Classification: semantics	embedding	juxtaposition	unification

- the semantics on which it is based;
- the verification techniques it provides or uses, if any. Expression “m.c.” stands for model checking.
- the refinement techniques it provides or uses, if any;
- the tools it provides or uses, if any;
- our evaluation of its readability: symbol $+$ means that a specification described in this approach is well readable, while $-$ means that the understanding of the whole model is more difficult;
- our evaluation of its adequacy for IS: symbol $+$ means that the main requirements of IS can be captured by the approach, while $-$ means that IS modelling is more difficult with this approach;
- the classification according to its aim;
- the classification according to its underlying semantics.

Table 4.2: Comparison of couplings of state-based and event-based specifications - part 2.

Approach	Circus [44]	PLTL and Event B [11]	EB^3-B [18]
Section	3.4	3.5	3.6
Languages	CSP Z	PLTL Event B	EB^3 B
Goal	unify theory	PLTL properties on Event B systems	EB^3 traces on B systems
Semantics	denotational	operational with LTS	operational with LTS
Verification	model checking	proof in B, m.c. for PLTL	proof in B
Refinement	unified refinement	Event B refinement	B refinement
Tools	m.c. to develop	B tools, model checker	B tools, no tool in EB^3
Readability	$-$	$+$	$+$
Adequacy for IS	$+$	$-$	$+$
Classification: aim	creation of a new language	verification of properties	verification of properties
Classification: semantics	unification	embedding	embedding

4.2 Discussion

In each approach, it is possible to exhibit at least one feature that is of interest for IS. CSP-OZ is a formal object-oriented language that allows class methods to be constrained by process expressions. Moreover, class methods and process events can be identified in two ways: in a single or in a double event view. In the latter case, the execution of the method and its output can be both considered. Circus is the only approach presented in the paper that defines an unified refinement that integrates both process and data refinements. For the other examples, refinement, if existing, involves only one part of the specification and the other part remains correct by consistency. More generally, CSP-OZ and Circus are interesting for IS since static and dynamic properties are specified together in a unique model. However, the weakness of these couplings is the specification readability and the high level of expertise required for applying and using them. On the other hand, they can be used to specify and analyse very complex systems.

csp2B provides an efficient tool to translate CSP descriptions into B specifications. Hence, it prevents inconsistencies between CSP and B, since the B specification is obtained from the CSP part. Moreover, the resulting B specification satisfies the CSP properties by translation. The main weakness of csp2B is the restriction on the use of the interleaving and parallel composition operators. In EB^3-B, the B specification satisfies the EB^3 event ordering properties, because it is a refinement of the EB^3 specification. Contrary to csp2B, EB^3-B requires a formal proof to verify the properties. Nevertheless, a proof often requires some creativity, and consequently a high expertise in theorem proving. On the other hand, EB^3 has been specially created for IS specification and B has already been used to formalize UML descriptions for IS.

In PLTL-Event B, temporal logic is used to verify dynamic properties on Event B systems. The use of temporal logic is very interesting for IS; in particular, one could verify dynamic data integrity constraints that are very difficult to capture with state-based languages like B or Z. The weakness of this approach is the modelling of closed systems in Event B. This implies to model the behaviour of all the IS end-users, which is very difficult, because it can be unpredictable. CSP || B provides an original way of dissociating the state-based specification from the event-based description. Each B machine is controlled by a CSP process called controller, and the different controllers are composed in parallel to communicate between them. The weakness of this approach is the limited expressiveness of CSP controllers that prevents the main process from specifying quantified interleaving in order to consider the dynamic constraints on several entities of the same type.

4.3 Related works

Many works deal with the couplings of paradigms. Here are some other examples that we can classify according to their semantics.

Abstract State Machines (ASM) [24], also called Evolving Algebras (EA), associates algebraical specifications to their semantical model. These machines are built such that the correctness of specifications holds by simple observations. ASM can modify the semantics of some functions or relationships of the model and tools simulate the behaviour. Since ASM integrate several features into an unique semantics, this approach can be classified in the **unification** group of couplings, along with Circus and CSP-OZ.

An approach for coupling CSP and Z is proposed in [6]. It identifies a CSP process with a Z abstract data type. This work is close to other works on CSP and Object-Z (like in Sect. 3.3). For instance, the approach described in [40] identifies a CSP process to a Object-Z class. Since the two viewpoints are only juxtaposed and not integrated, the CSP process has a limited control on the operations of the associated class or data type. Consequently, verification and refinement are not obvious and require a strong analysis of the combined model. These couplings between CSP and Z or Object-Z belong to the **juxtaposition** class of couplings, like CSP || B.

Z + Petri Nets [33] associates Petri nets [34] to the Z language [41]. Petri nets are composed of places and transitions. In this approach, two kinds of places are defined: standard places and *ZPlaces*. When a token is in a ZPlace, this means that the corresponding Z operation is enabled. Even if Petri nets can be seen as an event-based approach, this coupling is restricted by the scope of such models. Petri nets are often used to verify properties on specific parts of a whole system. This approach is similar to the couplings of CSP with Z or Object-Z where a CSP process is identified to a Z abstract data type or a Object-Z class. Z + Petri Nets belongs to the **juxtaposition** class of couplings.

ZCCS [20] is a CCS [30] specification where the value-passing of data is defined by Z schemas. This work deals with issues on CCS semantics and it can be classified in the **unification** class of couplings as an unifying semantics.

5 Conclusion

Requirements analysis and modelling are primary but difficult tasks to achieve in IS. Event-based and state-based specifications constitute two potential views of a system model. In event-based views, the required behaviour of the system is specified with liveness properties, explicit valid sequence of events, and/or sequence of events with temporal logic operators, while in state-based specifications, the model focuses on valid state transitions and safety properties of the system.

Formal languages are generally based on only one of these two aspects and therefore can only capture one side of the model [17]. Nevertherless, the other view may bring complementary aspects to the specified model. Some formal approaches use coupling of state-based and event-based specifications to capture both sides, but integrating two formal languages raises some problems. Formal notations require the definition of a common semantics to unify both parts of the model. Two complementary views are not always strictly orthogonal, and couplings may therefore be incorrect because of inconsistency or redundancy of specifications. On the other hand, specifying twice the same properties and proving that the two specifications are consistent allow us to detect specification errors.

Through a small example but typical in IS, we have presented six major couplings and shown the strengths and the weaknesses of each one. Even if the illustration by a single example cannot be generalized to every kind of systems, we have shown some interesting features to give us criteria of comparison. Specification of event or operation ordering properties is one of them. Verification and refinement have also been taken into account. Existence of tools has been checked. By analysing the examples of couplings presented in the paper, we think that the coupling of event-based and state-based formal specifications is an interesting way to consider both data integrity constraints and behavioural properties in IS modelling. Indeed, a state-based specification is well adapted to represent the IS data model, while event ordering properties can be specified and/or verified with event-based formal languages. Finally, by using formal languages, one can check that the whole specification is consistent.

References

[1] J.R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, Cambridge, U.K., 1996.

[2] J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *Second Conference on the B Method*, volume 1393 of *LNCS*, pages 83–128, Montpellier, France, 1998. Springer-Verlag.

[3] B-Core (UK) Ltd. B-Toolkit. `http://www.b-core.com/btoolkit.html`, 2007.

[4] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, New York, U.S.A., 1998.

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

[6] C. Bolton, J. Davies, and J. Woodcock. On the refinement and simulation of data types and processes. In *IFM*, pages 273–292, York, U.K., 1999. Springer-Verlag.

[7] M. Butler. csp2B : A practical approach to combining CSP and B. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 World Congress on Formal Methods*, LNCS, pages 490–508, Toulouse, France, 22-24th Sept 1999. Springer-Verlag.

[8] A.L.C. Cavalcanti and J.C.P. Woodcock. Refinement of Actions in Circus. In *REFINE'2002*, Copenhagen, Denmark, 2002. Electronic Notes in Theoretical Computer Science.

[9] A.L.C. Cavalcanti, A.C.A. Sampaio, and J.C.P. Woodcock. Unifying classes and processes. *Software and Systems Modeling*, 4(3):277–296, 2005.

[10] Clearsy. Atelier B. `http://www.atelierb-societe.com`, 2007.

[11] C. Darlot. *Reformulation et vérification de propriétés temporelles dans le cadre du raffinement de systèmes d'événements*. PhD thesis, Université de Franche-Comté, 2002.

[12] J. Davies and J.C.P. Woodcock. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, International, 1996.

[13] N. Evans, H. Treharne, R. Laleau, and M. Frappier. How to verify dynamic properties of information systems. In *Proceedings of the Second IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 416–425, Beijing, China, 2004. IEEE Computer Society Press.

[14] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

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

[16] B. Fraikin and M. Frappier. EB3PAI: an interpreter for the EB3 specification language. In *15th International Conference on Software and Systems Engineering and their Applications*, Paris, France, December 3-5 2002. CMSL.

[17] B. Fraikin, M. Frappier, and R. Laleau. State-based versus event-based specifications for information systems: a comparison of B and EB^3. *Software and Systems Modeling*, 4(3):236–257, 2005.

[18] M. Frappier and R. Laleau. Proving event ordering properties for Information Systems. In *ZB2003: Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[19] M. Frappier and R. St-Denis. EB^3: an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2):134–149, July 2003.

[20] A.J. Galloway and W.J. Stoddart. Integrated formal methods. In *INFORSID'97*, Toulouse, France, 11-13 June 1997. INFORSID.

[21] J.v. Garrel. Typechecking und transformation von CSP-OZ spezifikationen nach Jass. Master's thesis, University of Oldenburg, 1999.

[22] F. Gervais. EB^4 : Vers une méthode combinée de spécification formelle des systèmes d'information. Dissertation for the general examination, Doctorate degree program in computer science, Université de Sherbrooke (Québec), Canada, June 2004.

[23] Correct System Design Group. `http://csd.informatik.uni-oldenburg.de/ moby/`, 2007.

[24] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In *Specification and Validation Methods*, pages 9–36, Oxford, U.K., 1995. Oxford University Press.

[25] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, International, 1985.

[26] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, International, 1998.

[27] M. Jackson. *System Development*. Prentice-Hall, International, 1983.

[28] R. Laleau. Conception et développement formels d'applications bases de données. Habilitation à diriger des recherches, Université d'Évry Val d'Essonne, 2002.

[29] A. Mammar. *Un environnement formel pour le développement d'applications base de données*. PhD thesis, CNAM, 2002.

[30] R. Milner. *Communication and Concurrency*. Prentice-Hall, International, 1989.

[31] R. Milner. *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter Operational and algebraic semantics of concurrent processes. MIT Press, 1990.

[32] H.P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM, 1998.

[33] F. Peschanski and D. Julien. When concurrent control meets functional requirements, or Z + Petri-Nets. In *ZB2003: Formal Specification and Development in Z and B*, volume 2651 of *LNCS*, Turku, Finland, 4-6 June 2003. Springer-Verlag.

[34] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, International, 1981.

[35] A. Pnueli. The temporal logic of programs. In *18th annual symposium on the Foundations of Computer Science*, pages 46–57, Providence, Rhode Island, U.S.A., 1977. IEEE.

[36] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13: 45–60, 1981.

[37] A.C.A. Sampaio, J.C.P. Woodcock, and A.L.C. Cavalcanti. Refinement in Circus. In *FME 2002: Formal Methods - Getting IT Right*, volume 2391 of *LNCS*, Copenhagen, Denmark, 2002. Springer-Verlag.

[38] S. Schneider and H. Treharne. Communicating B Machines. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *ZB2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 416–435, Grenoble, France, 2002. Springer-Verlag.

[39] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Boston, U.S.A., 2000.

[40] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.

[41] J.M. Spivey. *The Z Notation: a Reference Manual*. Prentice-Hall, International, 1992.

[42] H. Treharne and S. Schneider. How to drive a B machine. In J.P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB2000: Formal Specification and Development in Z and B*, volume 1878 of *LNCS*, pages 188–208. Springer-Verlag, 2000.

[43] H. Treharne and S. Schneider. Using a process algebra to control B OPERATIONS. In *IFM'99 1st International Conference on Integrated Formal Methods*, pages 437–457, York, 1999. Springer-Verlag.

[44] J.C.P. Woodcock and A.L.C. Cavalcanti. The Semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, Grenoble, France, 2002. Springer-Verlag.