PREDICATES FOR STATE CHANGES VS. PROCESSES FOR EVENT PATTERNS

TOMMASO BOLOGNESI

Abstract. The two informal 'mental landscapes' that provide the intuitive substratum for state-oriented and event-oriented formal specifications are discussed, and abstractly characterised as networks of constraints. The structuring facilities offered by the two approaches are contrasted. A technique is introduced for expanding an event-oriented specification consisting of a fixed pattern of interacting processes into a state-oriented specification formed by a complex 'action predicate' manipulating a set of state variables. Although by this transformation the event and process concepts can be absorbed into the state-based conceptual framework, we discuss some good reasons for regarding these concepts as primitive expressive tools, and for structuring specifications around them.

Keywords. D.2.1 Requirements/Specifications, D.2.2 Design Tools and Techniques — Formal methods, Process algebra, ASM, B, CSP, High-level Petri nets, LOTOS, TLA.

1. Introduction

Formal approaches to the behavioural specification of complex, concurrent, reactive, distributed systems are often partitioned into state-based and event-based. In a state-based approach (e.g. Abstract State Machines [1, 2], B [3], TLA [4, 5], or Z [6]) the emphasis is on the structure of the global system state. One first defines an abstract model for that structure, in terms of constants, variables, sets, relations, functions and, possibly, other mathematical objects, and then defines its discrete perturbations, called *steps*, *operations*, *actions*, or *events*. Each event is described individually, and independently from other events, by identifying the conditions that enable its occurrence (pre-conditions) and the effects on the state (post-conditions): distinct chunks of formal specification are used for describing distinct types of state change.

In an event-based approach (e.g. CSP [7], CCS [8], or LOTOS [9, 10]) the emphasis is shifted to structures of events in time. One first identifies the set of possible events, usually starting by those that are observed at the system boundary, and then specifies the relations among them (e.g. sequence, causality, independence, mutual exclusion). Each chunk of formal specification describes a pattern involving, typically, several events.

The state-based approach is perhaps more popular, and more familiar to many system and software engineers, although event-based modelling techniques are being increasingly used, especially in the early phases of software development: examples are UML use case and interaction diagrams, and Message Sequence Charts.

In [11] Abadi and Lamport write:

> The popular approaches to specification are based on either states or actions. In a state-based approach, an execution of a system is viewed as a sequence of states, where a state is an assignment of values to some set of components. An action-based approach views an execution as a sequence of actions. These different approaches are, in some sense, equivalent. An action can be modeled as a state change, and a state can be modeled as an equivalence class of sequences of actions. However, the two approaches have traditionally taken very different formal directions. State-based approaches are often rooted in logic, a specification being a formula in some logical system. Action-based approaches have tended to use algebra, a specification being an object that is manipulated algebraically. Milner's CCS is the classic example of an algebraic formalism.

The fact that the two approaches are in some sense equivalent does not mean that the choice of one or the other, for practical system development, is devoid of any consequence. Conversely, the fact that the two approaches have historically taken different formal directions does not mean that they are based on two radically different ways of intuitively conceiving system behaviours. Unfortunately, the difference between logical and algebraic foundations tends to obscure the possible bridges between state-based and event-based intuitive thinking, and has favoured the spreading of somewhat rigid, if not dogmatic attitudes. In our opinion, the area of formal specification of software and hardware systems suffers from this circumstance, since engineers and developers are not provided with the most transparent advice for making effective choices among formal languages.

In this paper we are *not* concerned with comparing the two specification paradigms with respect to:

- theoretical expressive power (of course, any reasonable formal language in either family can simulate Turing machines);
- formal semantic foundations;
- support to formal verification techniques.

On the contrary, we are interested in investigating and comparing them relative to:

- the type of informal thinking, or brainstorming, that precedes the formalisation phase;
- the structuring facilities and expressive flexibility that they offer during the formalisation phase.

For our purposes, we say that a formal specification language has high *expressive flexibility* when it allows one to reduce the gap between informal and formal descriptions: it should be possible to structure the formal specification so that it closely reflects the informal landscape in the specifier's mind, which may involve pictorial elements such as box diagrams, and different levels of granularity. A highly flexible formal language shall offer constructs that directly correspond to the most typical behavioural scenarios found in complex, concurrent, reactive, distributed systems. For example, distributed systems are conceived in terms of complex components that interact with one another and with their users; correspondingly, a flexible formal specification language shall offer constructs for individually specifying these components, and for specifying their interaction patterns. Communication protocols are often conceived in terms of complex behavioural phases organised in sequence, e.g. *Connection Setup*, *Data Transfer*, *Connection Release*; again, a flexible language shall offer formal constructs for individually describing those phases, and for readily expressing their sequencing.

There exists of course a trade-off between expressive *flexibility* and expressive *generality*. Highly specialised constructs support flexible specification, but for a relatively narrow class of systems, while more fundamental and generic constructs support less flexible specification, but enjoy wider applicability.

It is our opinion that research in formal specification languages has often paid more attention to analytical power than to expressive flexibility, probably because these languages have been mainly introduced for overcoming the limitations of the existing informal or semi-formal specification languages, which were flexible and intuitive enough, but not suited to automated analysis. In this paper we disregard this trend, and concentrate purely on *specification* flexibility, not on *verification*.

In principle, we would like to refer to the expressive features of the state-oriented and event-oriented specification *paradigms*, without referring to any actual formal language. However, for fixing ideas, we shall borrow a few fundamental constructs and notations from existing languages, in particular from the state-oriented, logic-based language TLA (or TLA+) [4, 5], and from the event-oriented, process algebraic language LOTOS [9, 10].

In Section 2 we consider the generic expressive tools and structuring facilities that come with state-based thinking, as embodied by a number of widely known formal languages based on logics, and we introduce a general diagram for pictorially representing the underlying ‘mental landscape’. For substantiating our discussion we use, as an example, a TLA+ specification of a variant of a multi-process ring buffer discussed in [12].

In Section 3 we consider the generic expressive tools and structuring facilities that come with event-oriented thinking, as embodied by process algebras and process-oriented, concurrent languages. Again, we introduce a general diagram for representing event-based specifications, and use the same, ring buffer example. The idea of presenting both state-oriented and event-oriented specifications as networks of constraints is presented already in [13], which also introduces a unifying framework for classifying formal specification languages based on the types of constraint that they support.

In Section 4 we show that the gap between the two specification paradigms is not too large: a three-step expansion technique is introduced, that transforms the event-oriented, LOTOS specification of the ring buffer, into the state-oriented, TLA+ specification. This transformation from processes to predicates exploits a

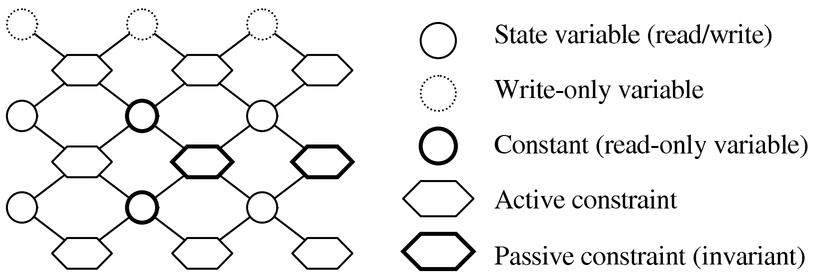
State variable (read/write)
Write-only variable
Constant (read-only variable)
Active constraint
Passive constraint (invariant)

Figure 2.1: The ‘state tapestry’

representation of parallel process compositions in Sum Of Products (SOP) form, that was first introduced in [14]. In light of the introduced transformation, we discuss the possible advantages of structuring systems in terms of processes rather than action predicates.

In Section 5 we summarize our results, provide some concluding remarks, and identify some topics for further work.

2. State-oriented formal specification

In this section we first summarize the fundamental elements of state-oriented thinking, by means of a generic diagram that we call ‘state tapestry’. Then we list some existing, widely known state-oriented formal languages based on logics, and briefly relate them to the elements of the diagram. Finally we introduce the state-oriented specification of the ring buffer for illustrating the typical structuring facilities offered by this paradigm.

2.1. The state tapestry

Figure 2.1 is an attempt to abstractly illustrate the fundamental elements of the mental picture one is likely to have in mind when starting a specification using a state-oriented formalism. In this early system conception phase, one is concerned with identifying:

- The set of useful *constants* (the fat circles).

- The *state variables* (the normal and dotted circles) that collectively model the global system state: a global state is an assignment of values to these variables. Their types are of varying complexity: one circle may represent an integer, another a whole data base. The state variables are thought of as preserving their values in time, until some event happens that (instantaneously) changes them: state variables evolve in discrete steps. The reasons for singling out write-only variables (the dotted circles) will be clear later, when relating state-oriented and event-oriented models.
- The '*logics*' that governs the evolution of the global state. This is structured as a set of *constraints* (the hexagons), each insisting on some subset of state variables. These subsets need not be disjoint, and indeed shared variables are profitably exploited by this approach. We use the generic term 'constraint' for neutrality with respect to any actual formal specification language. It is quite natural to distinguish between passive and active constraints:
  - a *passive constraint*, called 'invariant' in most formal languages, is used to express a relation among some of the variables that must hold in any reachable global state;
  - an *active constraint* is meant to describe a step in the system behaviour. A step is a pair of successive global states.

When we conceive a state tapestry, what we really intend to describe is the possibly infinite set of possibly infinite sequences of steps that the tapestry can go through, starting from some initial state. In drawing these diagrams, we omit the representation of the initial states (they must satisfy all invariants); in the sequel, for clarity, we shall also omit the constants.

In synthesis, the state-oriented mental picture is one of a network of constraints that establishes both some static properties of the system state, and some rules for its step-by-step evolution.

2.2. Formal languages

The concepts of invariant and active constraint are offered, under different names, and with some variants, by several formal specification languages and models. Our purpose here is not to give a detailed account and comparison of these formalisms, but only to help the reader in relating the abstract concepts of the state tapestry with some concrete, familiar language constructs.

In Predicate/Transition nets [15, 16] the state is represented by the distribution of value-carrying tokens over a set of *places*, and is modified by the firing of *transitions*. Transitions are equipped with predicates, and arcs connecting places and transitions are labelled by multi-sets of terms. A transition can be fired when it is possible to find an assignment for the free variables in its predicate and in the neighbouring terms such that the predicate is satisfied and the terms on the input arcs denote values of tokens actually available in the corresponding input places. If the transition is fired, these tokens are removed from their places, and new tokens are added to the output places, whose values are defined by the terms on the corresponding output arcs, evaluated under that same assignment. A transition is an active constraint that insists on the state variables represented by its neighbouring places. Transition are fired one at a time, thus reflecting an interleaving constraint policy.

ASM (abstract state machines) [1, 2] offer *rules* of form 'if *cond* then *updates*', where *cond* is a boolean condition on the state, and *updates* is a set of parallel assignments. The state is described by functions, and an update is the assignment of a new value to a function at one point. An ASM rule can be seen as an active constraint that insists on the state components accessed in *cond* and *updates*. Synchronous parallelism is the default composition operator in ASM: when writing a list of rules, one implies that *all* enabled rules are executed simultaneously. An enabled rule is one whose *cond* is satisfied in the current state. Thus, ASM adopts a synchronous constraint policy.

B [3] offers specific clauses for introducing state variables, for initialising them, and for defining invariants and operations. *Operations* are active constraints, enriched by the explicit identification of possible input and output parameters; they involve a pre-condition and a post-condition (the operation 'body'), which is a set of parallel assignments. Operations may occur one at a time, according to an interleaving policy.

Z [6], from which B has borrowed several ideas, describes state changes by means of *operation schemata*. Assignments are expressed in Z by using the so called *primed decoration*: *x'* denotes the new value assigned to variable *x* in a step. Z has special decorations for input and output variables: variables *x*? and *y*! occurring in an operation schema are not part of the system state, but are only used, respectively, for accepting input from, and offering output to the user of the operation. The special symbol 'Ξ' is used for expressing the fact that some variables should preserve their value through a step.

In TLA [4, 5] active constraints are represented by *actions*: these are logical formulae that must be satisfied by every step. As in Z, these formulae involve primed and unprimed variables, and one has to explicitly identify the components of the state that should not vary through a step. Complex actions can be defined by composing other actions by logical operators. Most typically, in TLA+ [5] one defines a global action, called *Next*, as the *disjunction* of other actions, following an interleaving policy.

Of course we have only mentioned the constructs that form a common set of fundamental, state-oriented expressive tools; every formal language offers further, specific constructs, that we are not interested in discussing here.

In order to introduce state-oriented specification examples, we need some notation for expressing active constraints. We borrow some syntax and terminology from TLA+, and model an active constraint by an *action* predicate, involving primed and unprimed variables (still keeping in mind that this is a state-based, not an action-based formalism). The whole, state-oriented specification shall be centred on an action predicate, which must be satisfied by every step in a system run (TLA uses the 'box' temporal operator to this purpose, as in formula '$\Box Next$'; we shall implicitly assume that the topmost action predicate in a specification is prefixed by this operator).

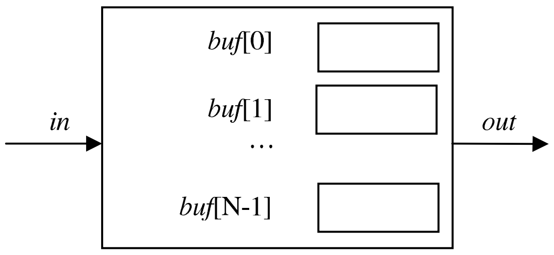

Figure 2.2: A ring buffer

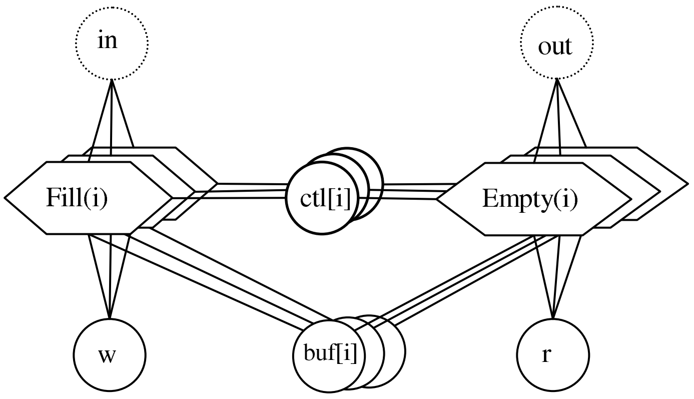

Figure 2.3: State tapestry for the ring buffer

2.3. State-oriented ring buffer

A FIFO ring buffer of capacity *N* is pictured in Figure 2.2, which we borrow from [12], together with the informal description of the system behaviour:

> The *i*th input value received on channel *in* is stored in *buf*[*i*-1 mod N], until it is sent on channel *out*. Input and output may occur concurrently, but input is enabled only when the buffer is not full, and output is enabled only when the buffer is not empty.

A state tapestry for the ring buffer is shown in Figure 2.3. The tapestry is formed by $2N$ constraints cumulatively insisting on $2N + 4$ variables. More precisely:

- variables *in* and *out* represent the input and output channels;
- bounded integer variables *w* and *r*, ranging in Z_N (the set $\{0, \ldots, N - 1\}$) indicate the next buffer location to be, respectively, written and read;

- each location *buf*[*i*] $(i = 1, \ldots, N-1)$ is handled by a *Fill*(*i*) and an *Empty*(*i*) constraint, both of which make use of a state variable *ctl*[*i*] indicating whether that location is *empty*, that is, to be filled (written), or *full*, that is, to be emptied (read).

In [L94b] Lamport compares a two-process and an N-process TLA specification of the ring buffer. The latter is similar to the version we have just described, and that we use throughout the paper, except that bit arrays *pp* and *gg*, and a special predicate *IsNext*, are used in place of *w* and *r*; we found the solution based on these integer variables a bit more convenient with respect to the comparison with event-oriented specifications.

Let us now specify the active constraints by action predicates.

$$
\begin{aligned}
Fill(i) \;\triangleq\;\; & \wedge\, i = w \\
& \wedge\, ctl[i] = \text{“}empty\text{”} \\
& \wedge\, ctl'[i] = \text{“}full\text{”} \\
& \wedge\, in' \in Data \\
& \wedge\, buf'[i] = in' \\
& \wedge\, w' = (w+1) \bmod N \\
& \wedge\, \text{UNCHANGED}(ctl[x] \text{ and } buf[x] \text{ for } x \neq i,\, r,\, out)
\end{aligned}
$$

$$
\begin{aligned}
Empty(i) \;\triangleq\;\; & \wedge\, i = r \\
& \wedge\, ctl[i] = \text{“}full\text{”} \\
& \wedge\, ctl'[i] = \text{“}empty\text{”} \\
& \wedge\, out' = buf[i] \\
& \wedge\, r' = (r+1) \bmod N \\
& \wedge\, \text{UNCHANGED}(ctl[x] \text{ for } x \neq i,\, buf,\, w,\, in)
\end{aligned}
$$

The global action for the ring buffer is:

$$
\begin{aligned}
Next \triangleq\;\; & \vee\, \exists i \in Z_N : Fill(i) \\
& \vee\, \exists i \in Z_N : Empty(i)
\end{aligned}
$$

A *Fill(i)* step writing the input value (*in’* $\in$ *Data*) into the i^{th} buffer location (*buf’*[*i*] = *in*’) is possible if the location is the one where writing is expected (*i* =

w), and is empty (*ctl*[*i*] = “empty”); as a consequence, *w* is updated (*w*’ = (*w*+1) mod *N*) and the location becomes full (*ctl’*[*i*] = “full”). The *Empty(i)* step is analogously defined.

The *in* and *out* channels are conceived here as write-only variables; correspondingly, in the definitions above they appear only in primed form. This fact is not surprising for variable out, but may appear strange when referred to variable in. In writing *buf’*[*i*] = *in*’ we are equating the new values of two state components, without saying how this new value is determined; we are adopting a modelling abstraction by which the value offered to the buffer via the in channel is ’created’ at the same time at which it is written into the buffer location, not earlier. In this way, we are viewing write-only variables as the channels, or gates found in process algebra: they are locations for hand-shake, rendez-vous interaction and communication.

UNCHANGED is a TLA+ predicate identifying the state components that should preserve their values through the step. For example, UNCHANGED(*buf, w, in*) is equivalent to:

$$\begin{aligned}&\wedge\, buf' = buf\\&\wedge\, w' = w\\&\wedge\, in' = in.\end{aligned}$$

In the two occurrences of this predicate we have slightly departed from the TLA+ notation when expressing that all components of an array except one are unaffected. Note that the constraint-variable links implied by the UNCHANGED clauses are not explicitly shown in the state tapestry, since they represent, for the specifier, a sort of secondary state management concern.

Due to the instances of the UNCHANGED predicate and to the preconditions involving variables *w* and *r*, it turns out that any *Next*-step of the system may only satisfy one instance of the *Fill* or *Empty* predicate; in particular, writing and reading the buffer are two mutually exclusive events, that is, they are never simultaneous.

Note that, as an alternative, we could have defined a two-constraint state tapestry consisting of un-parameterized *Fill* and *Empty* predicates, in light of the fact that every instance of *Fill*(*i*) (resp. *Empty*(*i*)) accesses *w* (resp. *r*), and requires *i* = *w* (resp. *i* = *r)*:

$$
\begin{aligned}
Fill \;\triangleq\;\; & \land\ ctl[w] = \text{“}empty\text{”} \\
& \land\ ctl'[w] = \text{“}full\text{”} \\
& \land\ in' \in Data \\
& \land\ buf'[w] = in' \\
& \land\ w' = (w+1) \bmod N \\
& \land\ \text{UNCHANGED}(ctl[x] \text{ and } buf[x] \text{ for } x \neq w, r, out) \\
Empty \;\triangleq\;\; & \land\ ctl[r] = \text{“}full\text{”} \\
& \land\ ctl'[r] = \text{“}empty\text{”} \\
& \land\ out' = buf[r] \\
& \land\ r' = (r+1) \bmod N \\
& \land\ \text{UNCHANGED}(ctl[x] \text{ for } x \neq i, buf, w, in) \\
Next1 \;\triangleq\;\; & Fill \lor Empty
\end{aligned}
$$

However, associating a specific instance *Fill(i)* and *Empty(i)* with each buffer location provides a better basis for comparison with the subsequent event-oriented specification, and also offers an example of application of existential quantification.

In conclusion, when thinking of system behaviours in terms of the state tapestry, and adopting a formal specification language such as TLA+, the most fundamental expressive tools and structuring facilities that we are offered are basically those found in first order logic, with conjunction typically used for specifying simultaneous updates of state variables, disjunction used for listing alternative action possibilities at the global level, and existential quantification used for a more general expression of nondeterministic behaviour.

3. Event-oriented formal specification

This section deals with event-oriented specification, and is structured as the previous one: we introduce the ‘event tapestry’, we relate it with some existing,

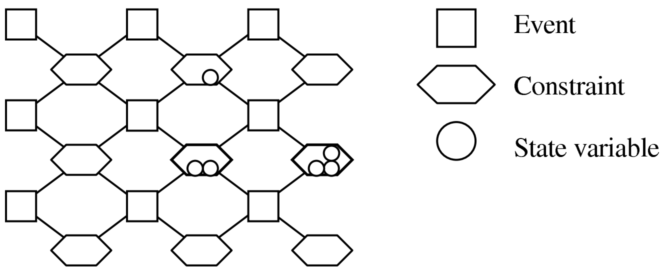

Figure 3.4: The 'event tapestry'

event-oriented formal languages based on process algebra, and we provide an event-oriented specification of the ring buffer for illustrating the typical structuring facilities offered by this paradigm.

3.1. The event tapestry

Analogous to Figure 2.1, Figure 3.4 identifies the elements of the mental landscape at the basis of event-oriented formal specification. In this case, in the early system conception phase one is concerned with identifying:

- The set of *events* (the boxes) that can be observed from the system environment. They are thought of as instantaneous manifestations of simple or structured values, possibly occurring at some identified location.
- The *constraints* (the hexagons) that relate event occurrences with one another, and determine possible system behaviours. Each constraint insists on some subset of the events: the subsets need not be disjoint, and indeed shared events are an essential feature of this approach. Constraints express relations such as sequential composition, causality, independence, mutual exclusion, synchrony, but also relations among the values assumed by the interconnected events.

The bipartite graph of Figure 3.4 is indeed so abstract that it allows for two alternative interpretations for the event-boxes.

- Box as event instance - A box represents a single, instantaneous event instance (that is, occurrence), so that infinite behaviours, with infinite event instances, require infinite diagrams. Events may or may not include, among their parameters, the absolute time of occurrence. Assume they do: then one could specify a network of constraints, each insisting on a subset of the events and relating their parameters – in particular, their timings (note that this is formally equivalent to a state tapestry with only passive constraints). Each solution to the network of constraints would represent a particular set of timed events, that is, a system behaviour. A simple example of a such a specification style, involving four timed events associated with a physical experiment, namely the relativistic measuring of a running train, is illustrated in [13]. Conversely, in the family of models called Labelled Event Structures [17, 18] events only possess an unstructured label, and the constraints are typically limited to causality, mutual exclusion, and independence.

- Box as event location - A box represents a conceptual or physical place for multiple, possibly infinite, event occurrences, so that finite diagrams may also describe infinite behaviours. Even in this case the constraints may or may not be concerned with time values; for example, one may specify that *any* two occurrences of events at locations *a* and *b*, within the same system run, must be separated by a minimum time delay of t_0.

In practice, the first approach cannot be used directly for specifying non trivial systems, but may be useful for representing the underlying semantics of models following the second approach. We shall therefore adopt the latter, and, for simplicity, shall not consider models handling explicit time information. We shall sloppily use the term *event* for referring both to an occurrence and to a place (also called *gate*).

An event tapestry identifies the constraints that may participate in the occurrence of an event, but is ambiguous about two aspects:

- Is it allowed for two events to occur simultaneously at different gates? For modelling a variety of concurrent, reactive, distributed systems, the ability to specify synchronicity of distinct events is not important; in other circumstances, for example in hardware design, synchronicity is a useful specification abstraction. Event-oriented formal languages have been designed that follow either approach, but the ones that follow the former appear as more 'fundamental', both in historic and in technical sense; thus we stick here to the basic assumption of event asynchrony.

- Does an event occurring at a given gate always require the synchronous participation of all constraints insisting on that gate? This would appear too strong a limitation. We do need some flexibility for expressing different synchronisation or interleaving policies within the same specification, depending on the considered gates and constraints. Thus, we expect formal specifications to allow us to resolve this graphical ambiguity, on a case by case basis.

In the same way as the concept of event is not completely excluded from state-oriented thinking – a step is an event – the concept of state plays some role in event-oriented thinking too. It is indeed convenient, and common, to think of constraints as provided with local state information, as depicted in Figure 3.4. State variables encapsulated by constraints may be of two types:

- control state variables – these are used exclusively for event ordering purposes;
- data variables – these are primarily used for modelling data structures, or concrete objects, and may or may not concur in controlling event ordering.

When we conceive an event tapestry, what we intend to describe is the possibly infinite set of possibly infinite sequences of events that may occur at the gates, while satisfying the constraints.

3.2. Formal languages

Process algebras such as CCS [8], CSP [7] and LOTOS [9] recognise the event as a first class citizen, and provide a notion of process, for modelling constraints, and a few behavioural operators – most notably parallel composition – for composing them. Events are interactions among the processes that form the system, or between these and the external environment. In CCS, parallel composition implies two-party synchronisation, while in CSP and LOTOS it supports multi-party synchronization.

In process algebra, behaviours are described by behaviour expressions, that are formed by behavioural operators. In ‘pure’ process algebra, state information is coded exclusively in the syntactic structure of the evolving behaviour expression, which plays the role of the control state variables mentioned above: it controls event orderings. In data-enhanced process algebra, state information can be represented also by variables, that are used for modelling the manipulated data structures.

In order to introduce event-oriented specification examples, we need some notation for expressing events and event constraints. We shall use a few operators, namely *action prefix*, *guard*, *choice*, *parallel composition*, borrowing them from LOTOS; of course, we shall need also *process definitions* and *process instantiations*. We shall depart from the standard LOTOS syntax in representing data and types, and in minor details that we do not even bother mentioning.

3.3. Event-oriented ring buffer

When adopting an event-oriented thinking mode, we may conceive the ring buffer as the event tapestry shown in Figure 3.5.

The global behaviour of the system is now conceived as a special composition of N+2 constraints insisting on two gates, and constraining the ordering of the events that occur there. Informally, the constraints are as follows.

LocalLoop For any given buffer location i, input and output events referring to that location must alternate; furthermore, the value offered by an output event must be the same value that was accepted by the preceding input event.

Inputs The input events must involve the N buffer locations cyclically.

Outputs The output events must involve the N buffer locations cyclically.

In constraint *LocalLoop*(i) variables *val* and *ctl* play the role of, respectively, variables $buf[i]$ and $ctl[i]$ in the state tapestry of Figure 2.3. We do not need the indices here because these variables are local to the constraint, which is already indexed. The diagram is ambiguous about the subset of constraints participating in each occurrence of an *in* or *out* event. The intended behaviour here is that every occurrence of an *in* (resp. *out*) event is a two-party synchronisation between the

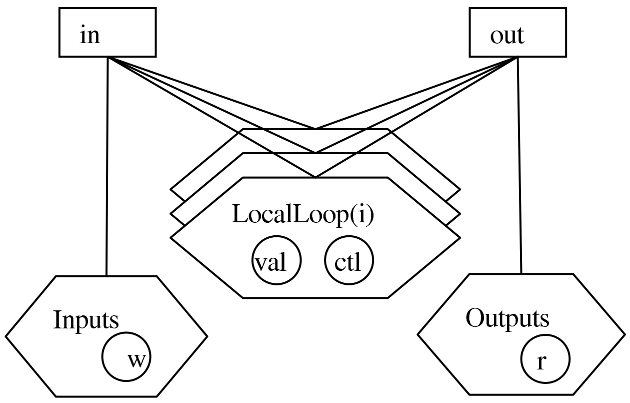

Figure 3.5: Event tapestry for the ring buffer

Inputs (resp. *Outputs*) constraint and exactly one instance (the right one!) of the *LocalLoop*(*i*) constraint. Constraint *LocalLoop*(*i*) owns only partial state information for deciding whether, say, an *in* event is possible, and this is provided by variable *ctl*; the other necessary information is provided by variable *w*, encapsulated in constraint *Inputs*, which identifies the next buffer location to be written. The two constraints can only share this information via a synchronisation event. Therefore, we conceive events as triples involving a gate name ('*gate*'), the index of the affected buffer location ('*slot*'), and the value to be written into, or read from it ('*value*'). Thus, the event space is:

$$\{gate.(slot, value) \mid gate : \{in, out\}, slot : Z_N, value : Data\}$$

Regard at *gate*, *slot* and *value* as field names. The dot separating the gate from the remaining, parenthesized event parameters is introduced for consistency with process algebraic notation.

For example, event $in.(0, 7)$ represents the manifestation at gate *in* of value 7 for buffer slot 0. The presence of the *slot* field is meant to allow processes *Inputs* and *Outputs* to enable at any time, by synchronisation, the proper instance of process *LocalLoop*.

The LOTOS specification of the complete system of constraints is provided below, by the definition of process *RingBuffer*.

```
RingBuffer[in, out] :=
         ||| LocalLoop[in, out](0)
         ||| ...
         ||| LocalLoop[in, out](N-1)
   |[in, out]|
         Inputs[in](0) ||| Outputs[out](0)

where

LocalLoop[in, out](i: ZN) :=
   in !i ?val:Data;
   out !i !val;
   LocalLoop[in, out](i)

Inputs[in](w: ZN) :=
   in !w ?_:Data;
   Inputs[in](w+1 mod N)

Outputs[out] (r: ZN) :=
   out !r ?_: Data;
   Outputs[out](r+1 mod N)
```

There is full correspondence between the event tapestry in Figure 3.5 and the three-line top behaviour expression of the specification above. Each of the *N*+2 constraints in that figure corresponds to a process instantiation in the multiple parallel composition expression. Every process has gate parameters, listed in square brackets, corresponding to the events directly connected to the constraint in the graph. This expressions exhibits a pattern that is found very frequently in LOTOS specifications, namely a combination of two forms of parallel composition: interleaving (‘|||’) and selective synchrony (‘|[*in*, *out*]|’). The former specifies that the events of the two composed processes must interleave their actions without any synchronisation; the latter specifies that events from the two components occurring at the *synchronisation gate list* identified within the operator itself (gates *in* and *out*) must synchronise, while the other events are still interleaved. The interleaving operator is a special case of the selective synchrony operator in which the synchronisation gate list is empty. By combining several instances of these operators in the expression above, where indentation is used to eliminate parentheses, we achieve the desired synchronisation pattern, that was left unexpressed in the diagram.

Let us now look at the three process definitions following the `where` keyword.

The action denotations, found in the LOTOS text at every line terminating with a semicolon symbol, precisely denote elements of the event space defined above. An action denotation is formed by a gate name followed by zero or more fields, each preceded by the '?' or the '!' symbol. An exclamation mark followed by an expression denotes the value of the expression, while a question mark followed by a variable declaration denotes any value of the specified type, and binds the fresh variable to that value. For example, action denotation `'in !i ?val:Data'` introduces a new variable val, and binds it to some value in $Data$; reference to this value is possible, via the variable, from within the scope of the action prefix operator, as done in `'out !i !val'`. The processes that share a gate do not have to control all the fields of the synchronisation events occurring there: they may express *don't care* conditions on some of them, via the '?' symbol and a conventional dummy variable represented by the underscore symbol, as in `'in !w ?_:Data'`.

In summary, the state variables in the abstract diagram of Figure 3.5 have been modelled in the event-oriented LOTOS specification either

- as parameters of a process – variables *w* and *r* become parameters of processes *Inputs* and *Outputs*, respectively – or
- as internal variables of a process, introduced by the '?' symbol – variable *val* is an internal variable of process *LocalLoop* – or
- as an evolving behaviour expression – variable *ctl* in the diagram corresponds to the evolving behaviour expression in the body of process *LocalLoop*, which can take two forms, namely the full expression, and the intermediate form `'out!i!val; LocalLoop[in,out](i)'` (after the out event the expression turns into a process instantiation, which is equivalent to the full expression). These two forms correspond, respectively, to the values *"empty"* and *"full"* used in the state-oriented specification.

In conclusion, while in state-oriented specification we use action predicates and their logical compositions, in the event-oriented setting the fundamental structuring facilities that we use are processes and process algebraic operators; in particular, we have illustrated the use of the two forms of the fundamental, parallel composition operator: interleaving and selective synchrony. In a network of interacting processes, each process specifies partial requirements on the ordering of some events, and on the values of some of their fields; parallel operators then specify which requirements are to be composed at which gates. This particular usage of parallel composition is known to LOTOS users as *constraint-oriented specification style*.

4. Bridging the gap between state-oriented and event-oriented specification

What is the difference between conceiving a system as a network of constraints on state variables, as done in Section 2, and as a network of constraints on events, as done in Section 3? Are we confronted with two fundamentally distinct ways of thinking about system behaviours: action predicates and their compositions for describing state changes on one hand, and processes and their compositions for describing event patterns on the other? The purpose of this section is to shed some light on the size and nature of this gap.

The comparison between the two specification para- digms is ultimately legitimised by the fact that both can be given semantic foundations in terms of transition systems. In the state-oriented setting, given the initial state σ_0 – an assignment of values to all state variables–, and the (fixed) global action predicate *Next* (possibly interpreted in an environment of other predicate definitions), transitions take the form:

$$\sigma_i \longrightarrow \sigma_{i+1}$$

where (σ_i, σ_{i+1}) is a *Next*-step, that is, a pair of states that satisfies predicate *Next*. In the event-oriented setting, given the initial behaviour expression bex_0 (possibly interpreted in an environment of process definitions), transitions take the form:

$$bex_i \xrightarrow{e_{i+1}} bex_{i+1}$$

where bex_i is the current behaviour expression, whose interpretation under the fixed Structural Operational Semantics (SOS) yields event e_{i+1} and the new expression bex_{i+1}.

However, for assessing the expressive flexibility of the two specification paradigms, as experienced by the specifier, we should not really look at the fine-grained level of individual transitions, but at the coarse grained syntactic level, where one manipulates action predicates or processes. One way to measure the 'distance' between these two ways of structuring system behaviours is to investigate the cost of converting one into the other.

In the sequel we illustrate a technique for transforming event-oriented specifications into equivalent state-oriented specifications. The technique can be applied to a relatively large and interesting class of specifications, involving a constant number of parallel interacting processes. Key factors in this transformation are:

- the creation of control state variables for representing evolving behaviour expressions,
- the conversion of gates into write-only state variables, and
- the representation of multiple parallel process compositions by gate-indexed sums of products (SOP's).

We illustrate our technique by converting the LOTOS specification of the ring buffer presented in Section 3.3 into the TLA+ specification of Section 2.3.

4.1. Step 1: gate-indexed predicates for the component processes

LOTOS process *RingBuffer* is defined in Section 3.3 as the parallel composition of instances of processes *LocalLoop*, *Inputs* and *Outputs*. Our first transformation step consists in manipulating those three processes individually.

Let us start with process *Inputs*. We wish to describe the dynamics of this process by means of an action predicate to be iteratively applied to a suitable set of state variables. The process body is defined in terms of two operators, namely action prefix and process instantiation. By applying the SOS rules of these operators we derive the following transition scheme for process *Inputs* (or, more precisely, for its generic instantiation), which fully captures its dynamics:

$$Inputs[in](w) \xrightarrow{in.(w,d)} Inputs[in]((w+1) \bmod N)$$

where *w*: Z_N, *d*: *Data*. We say that process *Inputs* is *stable*, because each transition in its behaviour relates two expressions exhibiting the same structure (a process instantiation *is* a behaviour expression). Which set of state variables is adequate for describing the universe before and after the execution of a transition by process *Inputs*, and which action predicate relates these two states? In light of the stability of the process, we do not need control state variables for representing evolving behaviour expression information. What changes in the expression before and after the transition is simply the value of the process parameter w; we shall therefore create a state variable for recording these evolutions. Then, based on the idea of using write-only state variables for modelling gates, we introduce state variable in: any event occurring at that gate is modelled by equating in' with the tuple of parameters of the event. In conclusion, state information is captured by the pair (w, in), and the action predicate that describes the evolutions of this state structure, as expressed by the transition scheme above, is:

$$\begin{aligned} Inputs_{in} \;\triangleq\;& \exists dummy \in Data : \\ & \quad \wedge\; in' = (w, dummy) \\ & \quad \wedge\; w' = (w+1) \bmod N \end{aligned}$$

Analogously, for process *Outputs* we derive action predicate:

$$\begin{aligned} Outputs_{out} \;\triangleq\;& \exists dummy \in Data : \\ & \quad \wedge\; out' = (r, dummy) \\ & \quad \wedge\; r' = (r+1) \bmod N \end{aligned}$$

that operates on the pair of state variables (r, out). We have introduced the name ‘dummy’ for the unnamed variable represented by the underscore symbol in the original process definitions. Existential quantification corresponds to the question marks in the original action denotations, and precisely captures the fact that those (dummy) variables are introduced, but not yet bound; they will, but only at synchronisation time.

A (LOTOS) process can be active at one gate at a time. Thus, for process X and gate g we shall denote by X_g the action predicate expressing the action capabilities of that process at that gate: these are the *gate-indexed action predicates* of the process. A predicate X_g defined as *false* indicates that the SOS does not support the derivation of transitions labelled by an event at gate g for process *X*. This is the case of predicates $Inputs_{out}$ and $Outputs_{in}$. The global action capabilities of a process shall be basically expressed by the disjunction of its gate-indexed predicates.

Consider now process *LocalLoop* (see Section 3.3). This process is not stable: after the first *in* event it reaches an intermediate state, and only after another *out* event does it assume again its initial shape, namely a self-instantiation. In light of the advantages of deriving gate-indexed action predicates from processes that are stable, we start by turning *LocalLoop* into one such process, called *LL*:

```
LL[in, out] (i: ZN, ctl: EmptyOrFull, val: Data) :=
   [] [ctl = "empty"] ->
         in !i ?newVal:Data;
         LL[in, out] (i, "full", newVal)
   [] [ctl = "full"] ->
         out !i !val;
         LL[in, out] (i, "empty", val)
```

This LOTOS-to-LOTOS transformation reflects the remarks at the end of Section 3.3 about the different ways to represent state information available in LOTOS. In process *LL* we have modeled the two-step cyclic evolution of the running behaviour expression of process *LocalLoop* by a control state variable *ctl*; this has required us to introduce the new variable *val* for recording the value accepted at gate *in*. The five elements that appear associated with each instance of constraint *LocalLoop* in Figure 3.5, namely (in, out, i, ctl, val), are those that precisely appear now in the header of process *LL*.

We look now for the state-oriented, TLA+ representation of this process, that is, for a tuple of state variables and for the gate-indexed action predicates that manipulate it. As we did for processes *Inputs* and *Outputs*, we directly derive the tuple of state variables from the parameters of the stable process, and use the same identifiers: thus, our state variables are (in, out, i, ctl, val).

Turning to the gate-indexed action predicates, we proceed, as before, by considering the SOS-supported derivation of transitions for the behaviour expression – let us call it E – in the body of *LL*. Expression E makes use only of the *guard*, *action prefix*, *choice* and *process instantiation* constructs (thereby conforming, not surprisingly, to what is known as *state-oriented* LOTOS specification style). By applying the SOS rules of these operators to the expression we derive the transition scheme:

$$LL[in, out](i, ctl, val) \xrightarrow{event} LL[in, out](i', ctl', val')$$

The topmost operator of E is binary choice. According to the SOS rules for this operator, a transition

$$B1[]B2 \xrightarrow{event} B'$$

is possible in two cases, namely if either

$$B1 \xrightarrow{event} B' \text{ or } B2 \xrightarrow{event} B',$$

where $B1$, $B2$ and B' are behaviour expressions. Correspondingly, referring to the choice expression E, we have two cases:

- Case 1. A transition from the first argument of the choice was used in the derivation. Then:
  - the guard $[ctl = \text{“}empty\text{”}]$ must have been true (by the SOS rule for the guard operator),
  - the $event$ must have occurred at gate in, with two data fields represented by the bound variable i and the yet unbound variable $newVal$ (by the SOS rule of action prefix),
  - in the new process instantiation it will be $i' = i$, $ctl' = \text{“}full\text{”}$, and $val' = newVal$ (by the SOS rule of process instantiation);

- Case 2. A transition from the second argument of the choice was used in the derivation. Then:
  - the guard $[ctl = \text{“}full\text{”}]$ must have been true,
  - the $event$ must have occurred at gate out, with two data fields represented by the bound variables i and val,
  - in the new process instantiation it will be $i' = i$, $ctl' = \text{“}empty\text{”}$ and $val' = val$.

Notice that, although in defining process *LL* we have attributed to *i*, *ctl* and *val* identical formal status, namely that of process parameters, variable *i* can be distinguished from the other two variables in that it is never affected by the transitions: this variable (the only one to appear in the original process *LocalLoop*) is nothing but a constant index used for distinguishing the N process instances, each handling a different buffer location. Thus, if $LL[in, out](i, ctl, val)$ denotes the i^{th} instance of process *LL*, we shall let $LL_{in}(i)$ and $LL_{out}(i)$ denote its associated gate-indexed action predicates, cumulatively insisting on state variables (in, out, ctl, val). In the body of these predicates, variable i shall never appear in primed form, while it shall appear as an index of ctl and val, which are proper state variables, for avoiding name collisions when all predicates are composed into the global, next-state action predicate.

The two transition cases examined above originate from the two inference rules for the choice operator, but it turns out that they also characterise, separately, the action capabilities of process *LL* at gates in and out, respectively. Thus, by reformulating in logical form the facts established in the two cases, we readily identify the two gate-indexed action predicates for the i^{th} instance of the process:

$$
\begin{array}{lcl}
LL_{in}(i) & = & \wedge\ ctl[i] = \text{“}empty\text{”} \\
 & & \wedge\ ctl'[i] = \text{“}full\text{”} \\
 & & \wedge\ \exists\ newVal \in Data : \\
 & & \quad \wedge\ in' = (i, newVal) \\
 & & \quad \wedge\ val'[i] = newVal
\end{array}
$$

$$\begin{aligned} LL_{out}(i) \quad = \quad & \wedge\, ctl[i] = \text{“}full\text{”} \\ & \wedge\, ctl'[i] = \text{“}empty\text{”} \\ & \wedge\, out' = (i, val[i]) \end{aligned}$$

Collectively, the action predicates derived for processes *Inputs*, *Outputs* and *LL* manipulate variables $(in, out, ctl, val, w, r)$. Each one of the above definitions should be complemented by an UNCHANGED clause expressing conservative assignments for the variables it does not explicitly update: we have omitted them for conciseness.

4.2. Step 2: gate-indexed algebraic expressions from parallel behaviour expressions

Let us now consider the top parallel behaviour expression of process *RingBuffer*, where we have replaced the instances of *LocalLoop* by those of *LL*, initialized with parameter *ctl* set to “empty”, and parameter *value* set to *undef*.

```
      ||| LL[in, out](0, "empty", undef)
      ||| ...
      ||| LL[in, out](N-1, "empty", undef)
|[in, out]|
      Inputs[in](0) ||| Outputs[out](0)
```

The stability of the $N + 2$ composed processes and the SOS rules of the parallel composition operator imply the stability of the whole expression: any transition yields a new behaviour expression which is identical to the original one, except for the actual parameters in process instantiations. Then, the global state of the state-oriented specification shall be simply the union of the state variables manipulated by the action predicates derived for each process, namely the already identified tuple $(in, out, ctl, val, w, r)$, where *ctl* and *val* are arrays of size N. We are interested in finding the logical expression that describes the evolution of this state structure.

In [14] a technique is introduced for deriving, gate by gate, algebraic expressions abstractly describing the action capabilities of a multiple parallel behaviour expression, and for turning this family of expressions into a convenient graphical form called *process interaction net* (PIN). In summary, letting

$\Pi = \{P_i[G_i] \mid 1 = 1, \ldots, n\}$ be a set of process instantiations, where G_i is the list of gates at which process P_i is potentially active,

$G = \cup_{i=1,\ldots,n} G_i$ be the universe of gates (viewing gate lists as sets),

E be a multiple parallel behaviour expression over Π,

that technique allows one to derive from *E*, and for each gate *g* in *G*, an algebraic expression E_g formed by the process instantiations (or, more concisely, by the bare process identifiers), by the sum and product operators (with product '*' often replaced by plain juxtaposition), by zero's and parentheses, as follows:

- process instantiation $P_i[G_i]$ becomes
  - $P_i[G_i]$ (or just P_i), if $g \in G_i$
  - 0 otherwise;
- parallel operator $|S|$, where *S* is a set of synchronisation gates, becomes
  - '*' if $g \in S$
  - '+' if $g \notin S$.

Thus, if *E* is the top parallel expression of process *RingBuffer* we derive two gate-indexed expressions:

$$\begin{aligned} E_{in} &= (LL(0) + \cdots + LL(N-1)) * (Inputs + 0) \\ E_{out} &= (LL(0) + \cdots + LL(N-1)) * (0 + Outputs). \end{aligned}$$

Recall that the parameter associated to *LL* is not a proper state variable, but an index, which can be understood as part of the process identifier. We then turn every E_g into Sum Of Products form, denoted ${E_g}^{SOP}$. For our example we obtain:

$$E_{in}^{SOP} = LL(0) * Inputs + \cdots + LL(N-1) * Inputs$$
$$E_{out}^{SOP} = LL(0) * Outputs + \cdots + LL(N-1) * Outputs$$

A PIN is a bipartite graph formed by process-nodes and gate-nodes; process-nodes are labelled by process instantiations, while gate-nodes are labelled by gate identifiers; label duplication is admitted only for gate-nodes. A process-node labelled by process instantiation $P_i[G_i]$ can only be connected to gate-nodes labelled by

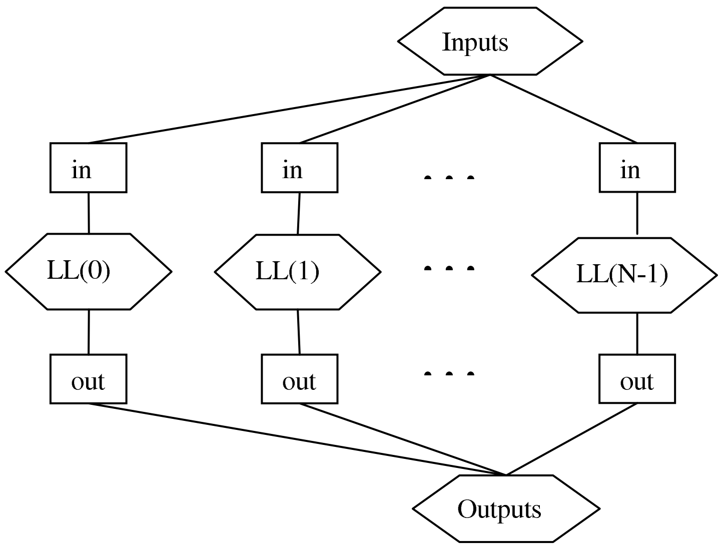

Figure 4.6: A process interaction net for process *RingBuffer*

gates that appear in G_i. A gate-node and all the process nodes adjacent to it form what we call a $(g$,*PP*$)$*-multiarc* when g is the label of the gate-node and *PP* is the set of labels of the process-nodes.

Based on the SOP expressions E_{in}^{SOP} and E_{out}^{SOP}, we can immediately obtain the PIN that graphically represents the interaction possibilities for the original parallel expression (see Figure 4.6): we create one process node for every process instantiation in the parallel expression *E*, and one $(g$,*PP*$)$-multiarc for every product term in $E_g{}^{SOP}$ formed by the set *PP* of process instantiations (or identifiers). A PIN can indeed be adopted as a multi-argument, graphical operator for expressing relatively complex process interaction patterns; [14] provides the (rather obvious) formal semantics for such an operator, and proves the behavioural equivalence between the expression and the graph.

Note that the equation P*0 = 0 precisely characterizes the fact that a process unable to operate at some gate blocks another process willing to synchronise with it on that gate.

Let us now see how to use gate-indexed SOP's for completing our transformation.

4.3. Step 3: expanding gate-indexed algebraic expressions by gate-indexed predicates

Once the product and sum operators are interpreted as logical conjunction and disjunction, E_g^{SOP} can be interpreted as a logical expression describing the different ways in which events at gate g may occur. These are indeed abstract conditions identifying only *which* possible process groupings may yield an interaction at some gate, without saying *how* each process supports those events. If we now expand the SOP's of a parallel expression *E* by replacing, in a gate-wise manner, process instantiations by the gate-indexed action predicates previously derived, we obtain a complete logical formulation of the action possibilities of *E*, that is, the complete state-oriented specification.

Consider a generic parallel expression *E* involving the composition of two processes, *P1* and *P2*, both insisting on gates *a* and *b*. By the conversion rules just described, the parallel expression yields the two expressions E_a and E_b, which are turned into SOP forms ${E_a}^{SOP}$ and ${E_b}^{SOP}$. Based on the definitions of process *P1* (resp. *P2*) we obtain the action predicates $P1_a$ and $P1_b$ (resp. $P2_a$ and $P2_b$). Then, the *a*-indexed predicates $P1_a$ and $P2_a$ are substituted for the corresponding process identifiers appearing in ${E_a}^{SOP}$. Similarly, $P1_b$ and $P2_b$ replace the occurrences of *P1* and *P2* in ${E_b}^{SOP}$. The two obtained expressions are composed by disjunction, thus yielding the final action predicate, which describes the action capabilities of the initial behaviour expression *E*.

Let us apply the procedure to our example. Define two substitutions:

Φ_{in} = [LL_{in}(i) / *LL*(i), i = 1, ..., *N* -1, *Inputs*$_{in}$/*Inputs*, *Outputs*$_{in}$/*Outputs*]
Φ_{out} = [LL_{out}(i) / *LL*(i), i = 1, ..., *N* -1, *Inputs*$_{out}$/*Inputs*, *Outputs*$_{out}$/*Outputs*]

Then we have:

$$\begin{aligned}
F_{in} &= E_{in}^{SOP}[\Phi_{in}] \\
&= LL_{in}(0) * Inputs_{in} + \cdots + LL_{in}(N-1) * Inputs_{in} \\
F_{out} &= E_{out}^{SOP}[\Phi_{out}] \\
&= LL_{out}(0) * Outputs_{out} + \cdots + LL_{out}(N-1) * Outputs_{out}
\end{aligned}$$

By switching to logic notation:

$$\begin{array}{lcl} F_{in} & \triangleq & (LL_{in}(0) \wedge Inputs_{in}) \vee \cdots \vee (LL_{in}(N-1) \wedge Inputs_{in}) \\ & = & \exists i \in Z_N : LL_{in}(i) \wedge Inputs_{in} \\ F_{out} & \triangleq & (LL_{out}(0) \wedge Outputs_{out}) \vee \cdots \vee (LL_{out}(N-1) \wedge Outputs_{out}) \\ & = & \exists i \in Z_N : LL_{out}(i) \wedge Outputs_{out} \end{array}$$

The final action predicate is defined as follows:

$$\begin{array}{lcl} F & \triangleq & F_{in} \vee F_{out} \\ & = & \vee\ \exists i \in Z_N : LL_{in}(i) \wedge Inputs_{in} \\ & & \vee\ \exists i \in Z_N : LL_{out}(i) \wedge Outputs_{out} \end{array}$$

Compare now this definition with the state-oriented specification of the ring buffer introduced in Section 2:

$$\begin{array}{lcl} Next & \triangleq & \vee\ \exists i \in Z_N : Fill(i) \\ & & \vee\ \exists i \in Z_N : Empty(i) \end{array}$$

We need to compare predicate *Fill(i)* and the conjunction $LL_{in}(i) \wedge Inputs_{in}$. By expansion:

$$
\begin{array}{rcl}
LL_{in}(i) \wedge \ Inputs_{in} & = & \wedge\ ctl[i] = \text{``}empty\text{''} \\
& & \wedge\ ctl'[i] = \text{``}full\text{''} \\
& & \wedge\ \exists\ newVal \in Data : \\
& & \quad \wedge\ \ in' = (i, newVal) \\
& & \quad \wedge\ \ val'[i] = newVal \\
& & \wedge\ \exists\ dummy \in Data : \\
& & \quad \wedge\ \ in' = (w, dummy) \\
& & \quad \wedge\ \ w' = (w+1) \bmod N \\
& = & \wedge\ i = w \\
& & \wedge\ ctl[i] = \text{```}empty\text{''} \\
& & \wedge\ ctl'[i] = \text{``}full\text{''} \\
& & \wedge\ \exists\ newVal \in Data : \\
& & \quad \wedge\ \ in' = (i, newVal) \\
& & \quad \wedge\ \ val'[i] = newVal \\
& & \quad \wedge\ \ w' = (w+1) \bmod N
\end{array}
$$

On the other hand, by rewriting

$$in' \in Data \longrightarrow \exists\ newVal \in Data : in' = newVal$$

and omitting the UNCHANGED clause, the body of *Fill*(*i*) becomes:

$$
\begin{array}{l}
\wedge\ i = w \\
\wedge\ ctl[i] = \text{``}empty\text{''} \\
\wedge\ ctl'[i] = full \\
\wedge\ \exists newVal \in Data : in' = newVal \\
\wedge\ buf'[i] = in' \\
\wedge\ w' = (w+1) \bmod N
\end{array}
$$

By equating arrays *val* and *buf*, the two formulae become equivalent, except for the structure of variable *in*, which, in the second case, records also the index of the buffer location affected by the input operation.

Analogously, we compare $Empty(i)$ and $LL_{out}(i) \wedge Outputs_{out}$. By expansion:

$$
\begin{array}{rcl}
LL_{out}(i) \wedge\ Outputs_{out} & = & \wedge\ ctl[i] = \text{“}full\text{”} \\
& & \wedge\ ctl'[i] = \text{“}empty\text{”} \\
& & \wedge\ out' = (1, val[i]) \\
& & \wedge\ \exists\ dummy \in Data : \\
& & \quad \wedge\ \ out' = (r, dummy) \\
& & \quad \wedge\ \ r' = (r+1) \bmod N \\
& = & \wedge\ i = r \\
& & \wedge\ ctl[i] = \text{“}full\text{”} \\
& & \wedge\ ctl'[i] = \text{“}empty\text{”} \\
& & \wedge\ out' = (i, val[i]) \\
& & \wedge\ r' = (r+1) \bmod N
\end{array}
$$

On the other hand, by omitting the UNCHANGED clause, the body of *Empty*(*i*) is:

$$
\begin{array}{l}
\wedge\ i = r \\
\wedge\ ctl[i] = \text{“}full\text{”} \\
\wedge\ ctl'[i] = \text{“}empty\text{”} \\
\wedge\ out' = buf[i] \\
\wedge\ r' = (r+1) \bmod N
\end{array}
$$

Again, the two formulae are equivalent, except for the names of the two arrays and the structure of variable *out*.

In conclusion, modulo the refinement to the gate variable, we have converted the process-oriented specification into the state-oriented one.

4.4. Remarks

Which conclusions can we draw about the differences between *thinking* at system behaviours in terms of states or events, in light of the above transformation?

We have started with a parallel composition of processes, possibly representing physical system components, and have ended up with a disjunction of action predicates in which processes and components have basically disappeared. The largest part of the gap is bridged by the crucial Step 2, which defines an *expansion* of a single parallel behaviour expression into a gate-indexed family of algebraic expressions, that are in turn expanded in SOP form. On one side we think in terms of interacting components, giving an implicit description of their individual interactions by means of the parallel operators; on the other side we leave the identification of system components implicit, while describing explicitly their interactions, case by case. We have used $N + 2$ process instances in the event-oriented specification, and $2N$ action predicate instances in the state-oriented specification of the ring buffer.

In general, where a typical process- and event-oriented formalisation would reflect the structure of a (distributed) system, a typical formalisation based on action predicates would basically disregard the system components, while directly providing a more lengthy, *explicit* enumeration of all their interaction possibilities, in form of logical disjuncts. In fact, the *explicit* description of the system structure is possible also in the state-based, logical setting: the composition of two subsystems is specified by the logical conjunction of their specifications, as explained in Chapter 10 of [5]. However, when using the low level logical operators of conjunction and disjunction for composing action predicates in a bottom-up fashion and specifying complex interaction patterns, one has to be very careful in filtering out all the undesired pairings – but only them! Further effort has to be spent in providing individual predicates with appropriate instances of the UNCHANGED clause, which, in general, depend on the context where the predicates are to be used, and on the desired global behaviour. This difficulty, well known as the *frame problem*, and the considerable, additional specification effort it involves, can be fully avoided by using higher level operators, that were explicitly designed for expressing interaction patterns.

5. Conclusions

We have investigated the expressive flexibility and structuring facilities offered by two fundamental specification paradigms, state-oriented and event-oriented, concentrating on very basic expressive features. In state-oriented specification, one is primarily concerned with modelling the system state by an adequate set of state variables, and with specifying the atomic operations that modify the state. In our state-oriented examples we have adopted a well established logical approach, by which the pre-conditions and post-conditions of operations are described by 'action predicates'; state variables have global scope, and can be accessed by any predicate. Complex specifications are structured by composing action predicates via the operators of predicate logic; advanced behavioural properties, such as liveness, could be specified by using temporal logic operators, but we have confined our discussion to simple specifications that deal only with safety properties.

In event-oriented specification, one is primarily concerned with modelling the space of events, and with specifying event patterns. In our event-oriented examples we have adopted the process-algebraic approach, in which these patterns are described by processes. We have conceived a process as an entity that may encapsulate state information (control and data structures) and is able to interact with other processes in its environment by rendez-vous. Simple process behaviours are expressed in terms of guards, action prefix, choice, recursive process instantiation. Complex specifications are structured by composing processes via parallel composition, in its pure interleaving or selective synchrony forms; of course, some other operators would be available, such as sequential process composition and disruption, but we have confined our discussion to parallel composition, since this operator is the key for structuring specifications, and because it naturally compares with logical conjunction.

We have illustrated a technique for transforming a fixed pattern of stable (or easily stabilized) interacting processes into a composite logical expression manipulating a set of state variables, thus reducing the gap between the two paradigms. We have not addressed the transformation problem in its full generality. It might be interesting to define a transformation technique for the generic (LOTOS) behaviour expression in a compositional way, by providing action predicates for the individual behavioural operators, and by building the overall action predicate in a syntax-driven way.

In [12] Lamport illustrates the insubstantiality of the notion of process by formally verifying the equivalence of two *different versions of a system*, namely the *N*-process and the two-process ring buffer, specified in the *same language*, namely

TLA; the transformation is completely formal: it uses rules of logic to rewrite formulae. The work in [12] has been an important source of inspiration for our paper, which has however followed a different path. The transformation technique we have illustrated relates two specifications of the *same version of the system* (the multi-process ring buffer), written in two *different formal languages*, one being LOTOS, the other being basically (a subset of) TLA+; and we have done it by explicitly taking into account the formal SOS rules of LOTOS. Another, secondary difference between the two exercises is that our multi-process version of the ring buffer is different from Lamport's, and has been preferred because it only exploits the rendez-vous process interaction mechanism, that fits into pure event-oriented thinking much better than shared variables.

Both exercises lead to the conclusion that 'processes are in the eye of the beholder'. However, while in [12] Lamport seems to push this consideration to the point of disqualifying process-oriented languages in almost any respect, we would rather take it as a mere indication that the process concept can be removed from specification languages without decreasing *theoretical* expressive power. We hope we have succeeded in providing some evidence that process-based languages such as LOTOS and CSP offer high expressive *flexibility*, at least limited to the discussed, common application scenarios, while preserving formal semantic foundations.

Dealing with specification issues, in this paper we have not taken up the verification challenge posed in [12], which questions the possibility to carry out an equivalence proof for two process-algebraic specifications of the ring buffer (the two-process and the multi-process version) purely based on process algebraic laws. While in this case we share the widely diffused opinion that process-algebraic axiomatic approaches to verification are not effectively scalable, we did work out a proof of equivalence between two LOTOS specifications of the ring buffer (the multi-process specification presented here, and Lamport's two-process version), based on the well known concept of bisimulation, and on SOS inference rules. This proof is not presented here for space reasons. It would be interesting to compare it with the one worked out by Lamport, and to check whether the granularity induced in our proof by the usage of SOS rules makes it more concise and readable.

Once the state-oriented and event-oriented specification paradigms are investigated, and their advantages/disadvantages assessed, a natural next step is to study the various ways in which they could be possibly integrated. Our first attempts to combine constraints on state variables and on events are described in [19]. These led to the definition of the 'co-notation' (constraint-oriented notation) [20, 21] an experimental language by which system behaviours can be specified as hierarchical compositions of simple and complex constraints both on events and on state variables. The co-notation offers invariants, action predicates, and a form of parallel composition similar to that of process algebra.

A lively research area has recently emerged that deals with integrated formal methods (IFM) [22]; research in this direction typically aims at designing formal languages that combine features from process algebra (CSP) and from state-based approaches (Z); examples are CSP-OZ [23], TCOZ [24], and Circus [25]. It would be interesting to investigate the type of informal thinking that precedes and supports formal specification activities based on these languages. An attempt to upgrade the state-oriented formalism of ASM by the inclusion of process algebraic operators is presented in [26].

In all the event-oriented formalisms that we have mentioned, events are mediated by state information, be it represented by state variables playing some role in the pre- and post-conditions of the event, or by the shape of an evolving behaviour expressions, corresponding to control state information. We like to conclude this paper by mentioning an interesting question raised by Lamport (in a private communication): are there *practical*, formal, pure event-oriented specification languages in which events are not mediated by an underlying state structure? The already mentioned labelled event structures [17, 18] are a genuinely stateless formalism, but they do not seem to offer the structuring facilities required for practical, large-scale applicability.

Acknowledgments

The idea of regarding event gates as state variables emerged after several lively discussions with Egon Boerger, whose purpose was to provide Abstract State Machines with some of the good expressive features of process algebra. I wish to express my gratitude to Hacène Fouchal for inviting me to OPODIS'2002, and for his patience in waiting for my contribution; this paper reflects most of the ideas that I have presented at that conference, and that I could discuss, in particular, with the other invited spaker, Leslie Lamport. Finally, many thanks to the latter, for drawing my attention to his paper 'Processes are in the Eye of the Beholder', and for his prompt and stimulating reactions to some over-optimistic views about process algebra and event-oriented thinking.

References

[1] Gurevich, Y.: Evolving Algebras 1993 - Lipari Guide. In Boerger, E., ed.: Specification and Validation Methods, Oxford Univ. Press (1995) 9–36

[2] Boerger, E., Staerk, R.: Abstract State Machines - A Method for High-Level System Design and Analysis. Springer (2003)

[3] Abrial, J.R.: The B-Book - Assigning Programs to Meanings. Cambridge Univ. Press (1996)

[4] Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems **16** (1994) 872–923

[5] Lamport, L.: Specifying Systems - The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2003)

[6] Spivey, J.M.: The Z Notation - A Reference manual. Prentice-Hall (1989)

[7] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)

[8] Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer-Verlag (1980)

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

[10] Brinksma, E.: ISO, Information Processing Systems, Open Systems Interconnection, LOTOS, a formal description technique based on the temporal ordering of observational behaviour - IS8807. Technical report, Geneva (1989)

[11] Abadi, M., Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems **15** (1993) 73–132

[12] Lamport, L.: Processes are in the eye of the beholder. Theoretical Computer Science **179** (1997) 333–351

[13] Bolognesi, T.: A conceptual framework for state-based and event-based formal behavioural specification languages. In: Proceedings of IEEE Int. Conf. on Complex Computer Systems (ICECCS 2004), IEEE Press (2004)

[14] Bolognesi, T.: Deriving graphical representations of process networks from algebraic expressions. Information Processing Letters **46** (1993) 289–294

[15] Reisig, W.: Petri Nets - An Introduction. Volume 4 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1985)

[16] Genrich, H.J.: Predicate/transition nets. Lecture Notes in Computer Science **254** (1987) 207–247

[17] Winskel, G.: Events in Computation. PhD thesis, Univ. of Edinburgh (1980)

[18] Winskel, G.: An introduction to event structures. Lecture Notes in Computer Science **354** (1989) 364–397

[19] Bolognesi, T., Ciaccio, G.: Cumulating constraints on the *when* and the *what*. In Tenney, R.L., Amer, P.D., Uyar, M.U., eds.: Formal Description Techniques VI - Proceedings of IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques (FORTE'93), North-Holland (1993) 433–450

[20] Bolognesi, T.: Expressive flexibility of a constraint-oriented notation. The Computer Journal **40** (1997) 259–277

[21] Bolognesi, T., Accordino, F.: A layer on top of PROLOG for composing behavioural constraints. Software Practice and Experience **28** (1998) 1415–1435

[22] Grieskamp, W., Santen, T., Stoddart, B., eds.: Proc. 2nd Int. Conference 'Integrated Formal Methods' (IFM 2000), Springer (2000)

[23] Fischer, C.: CSP-OZ: a combination of Object-Z and CSP. In Bowman, H., Derrick, J., eds.: Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS), Chapman and Hall, London (1997) 423–438

[24] Mahony, B.P., Dong, J.S.: Blending Object-Z and Timed CSP: An introduction to TCOZ. In Futatsugi, K., Kemmerer, R., Torii, K., eds.: The 20th International Conference on Software Engineering (ICSE'98), Kyoto, Japan, IEEE Computer Society Press (1998) 95–104

[25] Woodcock, J.C.P., Cavalcanti, A.L.C.: The semantics of Circus. In: ZB 2002: Formal Specification and Development in Z and B, Springer-Verlag (2002)

[26] Bolognesi, T., Boerger, E.: Abstract State Processes. In: Abstract State Machines - Advances in Theory and Applications (Proceedings of ASM2003 - LNCS 2589), Springer-Verlag (2003)

Authors addresses:

Tommaso Bolognesi

C.N.R. - ISTI

1, Via Moruzzi, 56124

Pisa, Italy

t.bolognesi@isti.cnr.it