Composing Event Constraints in State-based Specification

Tommaso Bolognesi[1]

CNR - ISTI,
t.bolognesi@isti.cnr.it,
WWW home page: http://fmt.isti.cnr.it/

Abstract. Event-based process algebraic specification languages support an elegant specification technique by which system behaviours are described as compositions of constraints on event occurrences and event parameters. This paper investigates the possibility to export this specification paradigm to a state-based formalism, and discusses some deriving advantages in terms of verification.

1 Introduction

Process-algebraic specification languages have received much attention in the eighties and nineties, with emphasis, in the first decade, on theoretical foundations and semantics, and with various experiments and projects, in the second decade, for their application to the development of large-scale software systems.

Many people involved in software production, and several researchers from academia as well, agree today in observing that most of the early, ambitious goals of process algebra have not been met. The diffusion of process-algebraic languages within software companies, as a routine tool for design and verification activities, is quite limited.

Process-algebras and related methods represent just a portion of the articulated area of Formal Methods (FM). It is fair to admit that, so far, FM's in general have gained only limited acceptance in industry. Typically, they are applied in the development of safety-critical (sub-)systems, for which formal verification becomes crucial; but their popularity does not compare with that reached by more 'practical' approaches such as the object-oriented UML.

The reasons for these difficulties have been already discussed quite extensively in the past; we only mention, below, those that have more directly motivated the work presented in this paper.

– The offer of FM's appears still too wide and fragmented. The WWW Virtual Library on Formal Methods, as of today, provides a list of 92 individual notations and associated tools (http://vl.fmnet.info/#notations). While some techniques are fairly stable, others keep evolving, and new ones appear, although less frequently than in the past; making choices in such a wide and dynamic set is hard, also considering the high training costs associated with the adoption of a FM.

- 'Political' concerns and parochial attitudes have often obstructed the recognition of similarities among formal languages and the convergence towards fewer stable proposals. FM's would greatly benefit from some unification effort, as it happened with the definition of the Unified Modelling Language. Some steps in this direction have been already taken, as indicated, for example, by the recently established Integrated Formal Methods (IFM) series of international conferences, by the work on the Unified Theories of Programming [1], and by proposals such as *Circus* [2], but much remains to be done, also in the sense discussed in the next bullet.
- Perhaps more specifically in the area of process algebraic FM's, there has often been a mismatch between the offers from FM developers, and the real needs of perspective FM users, with the former emphasizing on the meta-level of formal semantic definition, on axiomatisations, on countless semantic relations, and the latter emphasizing on ease of use, expressive flexibility, pragmatic guidelines, tool support. The well known work on Design Patterns is an important factor for the wide acceptance of Object Oriented technology (note that patterns are language-independent!): this is, again, a lesson from O-O technology that the FM community might want to take into serious consideration.

Formal approaches to the behavioural specification are often partitioned into state-based (e.g. Abstract State Machines [3, 4], B [5], TLA [6, 7], or Z [8]), usually rooted in logics, and event-based (e.g. CSP [9], CCS [10], or LOTOS [11, 12]), with algebraic roots. Today, software engineers seem to look more favourably at state-based FM's: they better relate with other traditional and well understood engineering methods, they have proven to be quite effective in hardware design, and they lend themselves to increasingly effective analytical techniques. And yet, event-oriented thinking keeps playing an important role, at least in the early phases of system development; for example, use case diagrams, scenarios, message sequence charts are widely used in requirements analysis.

This paper is an attempt to promote some cross-fertilisation and convergence in the area of FM and, in particular, between the event-based and state-based specification paradigms: in line with the remarks above, we wish to address aspects of language pragmatics, namely specification style, structuring principles, and tool-supported verification, rather than semantic foundations.

Fifteen years have passed since the publication of the LOTOS specification of Al's Node by Quemada and Azcorra, [13] which has represented a tiny but very effective example of the so called 'Constraint-Oriented' specification style. We believe that, while several results of theoretical interest from process algebra have failed to scale up to realistic-size system development, some expressive tools from this area of FM's have shown to be very effective. We wish to list three of them, in order of increasingly complexity:

- The basic notion of 'event'. This is an abstraction that process-oriented specification languages regard as equally important as the notion of 'state variable', but conveniently distinguished from it.

- Parallel composition operators. These are the most typical among the ad-hoc, high level behavioural operators of process algebras. In particular, *selective synchrony* (i.e., the general LOTOS parallel operator '|[S]|', where S is a list of synchronization gates), fundamentally based on the gate/event concept, is crucial for compactly specifying interaction patterns simultaneously involving interleaving and synchronized events from different system components.
- The constraint-oriented specification style mentioned above. This way of structuring specifications is in turn based on the use of the selective synchrony parallel operator.

In this paper we investigate the extent to which the three expressive tools above, typical of process-algebraic specification, can be imported into state-based specification. For fixing ideas, we select one representative language from each paradigm: we choose LOTOS and TLA+. For space reasons, we have to assume some familiarity with both of them, but the uninitiated reader should still be able to follow our discussion on specification structuring principles, without being distracted by the details of the two concrete languages.

The key question addressed by the paper can be summarized as follows.

- Assume you want to adopt a state-based formalism such as TLA+ for specifying systems, being attracted by:
  - the simplicity and universality of its constructs, which are based on first order logic, and on few temporal logic operators – no need to learn ad-hoc, process-algebraic behavioural operators;
  - the conceptual simplicity of the verification technique it supports (implementation is pure logical implication);
  - the free availability of tools (most notably, the model checker TLC).
- Assume also you have an inclination towards structuring some behavioural specifications as collections of constraints insisting on partially overlapped sets of events (deriving from a possibly unconfessed past experience in the community of 'LOTOS-eaters').

Should you give up your way to conceive specifications? In this paper we try to justify a negative answer to this question.

In Section 2 we recall Al's Node, we introduce two abstract, monolithic formal specifications of it, in LOTOS and in TLA+, and show one way to model 'events' in TLA+.

In Section 3 we address constraint-oriented specification. We show that the behaviour of Al's Node can be conveniently expressed in terms of three constraints, and that their interplay can be compactly expressed in LOTOS. We then illustrate two ways in which we can approximate that structuring in TLA+.

In Section 4 we take advantage of the translations into TLA+ by conducting some verification activity. We prove, by the TLC model checker, that the two TLA+ constraint-oriented specifications of Al's Node are equivalent, and that they implement the initial, monolithic specification.

In Section 5 we present our conclusions and identify items for further work.

2 Event-based, monolithic specification

In this section we introduce Al's Node by providing an informal, monolithic (one process) description and a formalisation in LOTOS. We then illustrate a very similar formalization in TLA+, based on a simple idea for preserving the notion of observable event. This TLA+ specification provides the reference against which the subsequent, structured, constraint-oriented TLA+ specifications are formally verified.

2.1 Informal description of Al's Node

Al's Node is a switching device for controlling message traffic in a network. The node keeps the following data structures:

– a bag of messages,
– a set of active ports,
– a set of (*route*, *port*) pairs, defining the active routings.

Messages can be accepted by the system at any active port, and stored in the node. An incoming messages consist of a (*data*, *route*) pair: the *data* item has to be re-directed to the associated *route*. Stored messages can be lost, or output, not necessarily in FIFO order, at some active port that is paired, according to the current active routings, to the route indicated in the message.

For keeping specifications concise, and in accordance with the original example, we allow elements to be only added to, not removed from the set of active ports and the set of (*route*, *port*) pairs. We omit the treatment of timeouts.

2.2 Monolithic specification in LOTOS

The LOTOS specification makes use of three predefined sets: *Data*, *Ports*, *Routes*. For representing data structures we depart from the LOTOS standard, and use plain mathematical structures; besides being more convenient, they can be reused in TLA+. The specification consists in a single process, called *AlsNodeMonol*. This process insists on gates DATA_IN, DATA_OUT, DATA_LOSS, CTL, and is parameterized by the three variables:

msgBag: a bag of (*data*, *route*) pairs, of size at most N,
activePorts: a set of *Ports*,
activeRoutes: a set of (*route*, *port*) pairs.

Process *AlsNodeMonol* is structured as a set of alternatives, each consisting of an event, controlled by a guard (a 'selection predicate', in LOTOS terminology), and followed by a recursive process instantiation with updated parameters. In the specification we omit gate lists in recursive process instantiations: they are the same that appear in the enclosing process definition. We use symbol '(+)' for bag union and '(-)' for bag subtraction.

```
PROCESS AlsNodeMonol
   [DATA_IN, DATA_OUT, DATA_LOSS, CTL]
   (msgBag: BagOf(Data x Routes),
   activePorts: SubsetOf(Ports),
   activeRoutes: SubsetOf(Routes x Ports))
:=
   DATA_IN ?d: Data ?r: Routes ?p: Ports
      [p in activePorts, Size(msgBag) < N];
      LET msgBag' = msgBag (+) {<d, r>} IN
      AlsNodeMonol [---] (msgBag', activePorts, activeRoutes)
   []
   DATA_OUT ?d: Data ?r: Routes ?p: Ports
      [p in activePorts, <d, r> in msgBag, <r, p> in activeRoutes)];
      LET msgBag' = msgBag (-) {<d, r>} IN
      AlsNodeMonol [---] (msgBag', activePorts, activeRoutes)
   []
   DATA_LOSS ?d: Data ?r: Routes  [<d, r> in msgBag];
      LET msgBag' = msgBag (-) {<d, r>} IN
      AlsNodeMonol [---] (msgBag', activePorts, activeRoutes)
   []
   CTL ?p: Ports   [p notin activePorts];
      LET activePorts'= activePorts union {p} IN
      AlsNodeMonol [---] (msgBag, activePorts', activeRoutes)
   []
   CTL ?r: Routes  ?p: Ports   [<r, p> notin activeRoutes];
      LET activeRoutes'= activeRoutes union {<r, p>} IN
      AlsNodeMonol [---] (msgBag, activePorts, activeRoutes')
ENDPROC (* AlsNodeMonol *)
```

The interpretation of the events at the process gates is as follows. An event at gate DATA_IN, with parameters $(d,\ r,\ p)$ represent the input of message $(d,\ r)$ at port p. Events at gate DATA_OUT have a similar interpretation, while those at DATA_LOSS do not need a *port* parameter. Gate CTL is used only for adding elements to *activePorts* and *activeRoutes*. The complete set of possible events is explicitly defined in TLA+ module *AlsNodeInterface* in the next subsection.

2.3 Monolithic specification in TLA+ and event representation

A monolithic specification of Al's Node in TLA+ with the same structure of the above LOTOS specification can be provided quite easily, since the latter matches a *state-oriented* style, and makes use of a restricted number of operators, namely *action prefix* with *guards*, *choice* and *process instantiation*. We only have to decide about the representation of events. TLA+ does not offer a primitive notion of event. It offers *actions*. A TLA+ action is logical formula that includes primed and unprimed variables, and may or may not be satisfied by a *step*. A step is a pair of successive *states*. A state is an assignment of values to all state variables.

LOTOS events occur at *gates*. A gate is a location at which processes atomically synchronize and agree on tuples of values. A gate cannot be assimilated to a normal state variable because it is intrinsically unable to retain any value. Processes may be thought of as writing values at gates, but these values are immediately lost. They can only be retained by using local process variables (as in `'?d: Data'`).

Thus, we model events in TLA+ by a write-only state variable that we conventionally call e, and that shall only appear in primed form in action predicates, so that it never contributes to pre-conditions. Event e shall be a tuple (or a record), in which the first component is a sort of event type, and is the equivalent of a LOTOS gate identifier, while the remaining components represent the event parameters. The advantages of introducing the special event variable become more apparent with constraint oriented specification, as discussed in the next section.

A TLA+ specification is formed by a set of modules. Our first module, presented below, is called *AlsNodeInterface*. It introduces the basic components of the specification, that shall be imported by the subsequent TLA+ specifications. The interface introduces the node capacity N, the predefined sets *Data*, *Ports* and *Routes*, the special event variable e, and the set where it is supposed to range. These are also the events of the LOTOS specification.

MODULE *AlsNodeInterface*

VARIABLE e

CONSTANTS N, $Data$, $Ports$, $Routes$

$$\begin{aligned}
&EventSet \triangleq \\
&\qquad \{\langle \text{“DATA\_IN”}, d, r, p\rangle : d \in Data, r \in Routes, p \in Ports\} \\
&\qquad \cup \{\langle \text{“DATA\_OUT”}, d, r, p\rangle : d \in Data, r \in Routes, p \in Ports\} \\
&\qquad \cup \{\langle \text{“DATA\_LOSS”}, d, r\rangle : d \in Data, r \in Routes\} \\
&\qquad \cup \{\langle \text{“CTL”}, p\rangle : p \in Ports\} \\
&\qquad \cup \{\langle \text{“CTL”}, r, p\rangle : r \in Routes, p \in Ports\}
\end{aligned}$$

$$EventTypeInvariant \triangleq e \in EventSet \cup \{\langle\rangle\}$$

The monolithic TLA+ specification of Al's Node is provided in two steps, by modules *InternalAlsNodeMonol* and *AlsNodeMonol* below. The attribute 'internal' refers to the fact that the specification in that module makes use of variables *msgBag*, *activePorts*, *activeRoutes*, that should not be regarded as contributing to the reference, observable behaviour of the system. Adopting the process-algebraic view, the only observable behaviour should be that expressed by events, that is, by variable e (which is contributed by module *AlsNodeInterface*). Thus, in module *AlsNodeMonol* all variables except e are hidden, by means of temporal existential quantification.

MODULE *InternalAlsNodeMonol*

EXTENDS *AlsNodeInterface*, *Naturals*, *Bags*

VARIABLES to be internalized

msgBag,	bag of messages in transit
activePorts,	set of active ports
activeRoutes	the dynamic association route-port

$TypeInvariant \triangleq$
- $\land\ EventTypeInvariant$
- $\land\ IsABag(msgBag)$
- $\land\ BagToSet(msgBag) \in \text{SUBSET}\ (Data \times Routes)$
- $\land\ activePorts \in \text{SUBSET}\ Ports$
- $\land\ activeRoutes \in \text{SUBSET}\ (Routes \times Ports)$

$DataIn(d, route, port) \triangleq$
- $\land\ e' = \langle \text{“DATA\_IN”}, d, route, port \rangle$
- $\land\ port \in activePorts$
- $\land\ msgBag' = msgBag \oplus SetToBag(\{\langle d, route \rangle\})$
- $\land\ \text{UNCHANGED}\ \langle activePorts, activeRoutes \rangle$

$DataOut(d, route, port) \triangleq$
- $\land\ e' = \langle \text{“DATA\_OUT”}, d, route, port \rangle$
- $\land\ port \in activePorts$
- $\land\ \langle d, route \rangle \in BagToSet(msgBag)$
- $\land\ \langle route, port \rangle \in activeRoutes$
- $\land\ msgBag' = msgBag \ominus SetToBag(\{\langle d, route \rangle\})$
- $\land\ \text{UNCHANGED}\ \langle activePorts, activeRoutes \rangle$

$DataLoss(d, route) \triangleq$
- $\land\ \exists\, x \in BagToSet(msgBag) : x = \langle d, route \rangle$
- $\land\ e' = \langle \text{“DATA\_LOSS”}, d, route \rangle$
- $\land\ msgBag' = msgBag \ominus SetToBag(\{\langle d, route \rangle\})$
- $\land\ \text{UNCHANGED}\ \langle activePorts, activeRoutes \rangle$

$AddPort(port) \triangleq$
- $\land\ e' = \langle \text{“CTL”}, port \rangle$
- $\land\ port \notin activePorts$
- $\land\ activePorts' = activePorts \cup \{port\}$
- $\land\ \text{UNCHANGED}\ \langle msgBag, activeRoutes \rangle$

$AddRoutePort(route, port) \triangleq$
- $\land\ e' = \langle \text{“CTL”}, route, port \rangle$
- $\land\ \langle route, port \rangle \notin activeRoutes$
- $\land\ activeRoutes' = activeRoutes \cup \{\langle route, port \rangle\}$
- $\land\ \text{UNCHANGED}\ \langle msgBag, activePorts \rangle$

$$\begin{aligned}
&\mathit{Init} \triangleq \\
&\qquad \land e = \langle\rangle \\
&\qquad \land \mathit{msgBag} = \mathit{EmptyBag} \\
&\qquad \land \mathit{activePorts} = \{\} \\
&\qquad \land \mathit{activeRoutes} = \{\} \\
&\mathit{Next} \triangleq \\
&\qquad \lor \exists\, d \in \mathit{Data},\ r \in \mathit{Routes},\ p \in \mathit{Ports} : \mathit{DataIn}(d,\ r,\ p) \\
&\qquad \lor \exists\, d \in \mathit{Data},\ r \in \mathit{Routes},\ p \in \mathit{Ports} : \mathit{DataOut}(d,\ r,\ p) \\
&\qquad \lor \exists\, d \in \mathit{Data},\ r \in \mathit{Routes} : \mathit{DataLoss}(d,\ r) \\
&\qquad \lor \exists\, p \in \mathit{Ports} : \mathit{AddPort}(p) \\
&\qquad \lor \exists\, r \in \mathit{Routes},\ p \in \mathit{Ports} : \mathit{AddRoutePort}(r,\ p)
\end{aligned}$$

$$\mathit{Spec} \triangleq \quad \mathit{Init} \land \Box[\mathit{Next}]_{\langle e,\ \mathit{msgBag},\ \mathit{activePorts},\ \mathit{activeRoutes}\rangle}$$

$\mathit{BagConstraint} \triangleq \mathit{BagCardinality}(\mathit{msgBag}) \leq N$

THEOREM *TypeInvariant*

MODULE *AlsNodeMonol*

EXTENDS *AlsNodeInterface*

$$\begin{aligned}
&\mathit{Inner}(\mathit{msgBag},\ \mathit{activePorts},\ \mathit{activeRoutes}) \triangleq \\
&\qquad \text{INSTANCE } \mathit{InternalAlsNodeMonol}
\end{aligned}$$

$$\begin{aligned}
\mathit{Spec} \triangleq \quad &\boldsymbol{\exists}\, \mathit{msgBag},\ \mathit{activePorts},\ \mathit{activeRoutes} : \\
&\mathit{Inner}(\mathit{msgBag},\ \mathit{activePorts},\ \mathit{activeRoutes})!\mathit{Spec}
\end{aligned}$$

The central part of module *InternalAlsNodeMonol* consists in the definition of five basic actions: *DataIn*, *DataOut*, *DataLoss*, *AddPort*, *AddRoutePort*. Each manipulates in a different way parts of the global state, and accounts at the same time for one event type which, in the first three cases, has the same name as the action predicate. These five actions appear as disjuncts in the global action *Next*. Finally, formula *Spec* defines all legal behaviours of Al's Node: these are infinite sequence of global states in which the first element satisfies the *Init* formula, and any pair of adjacent states satisfies the *Next* action, or leaves all variables unchanged (*stuttering step*). Note that, although in principle a step could simultaneously satisfy more than one basic action, this never happens because the actions end up being mutually exclusive, if not for the incompatibility of their pre-conditions, because of the disjoint event sets that they support (post-conditions).

The correspondence between the LOTOS and TLA+ specifications is clear. In particular: LOTOS choice corresponds to disjunction; selection predicates and the updated process parameters reflect pre- and post-conditions in TLA+

actions; the LOTOS symbol '?' corresponds to existential quantification, which generalizes disjunction.

3 Event-based, constraint-oriented specification

In this section we first provide an informal, constraint-oriented description of Al's Node and its LOTOS formalisation, essentially following [QA89]. Then we introduce two TLA+ specifications meant to approximate, in two different ways, the compact structure of the LOTOS specification.

3.1 Informal

The behaviour of Al's Node is captured by the three constraints illustrated in Figure 1. This diagram reflects the key ideas of event-based, constraint-oriented specification. The behaviour of a system is conceived as the composition of some

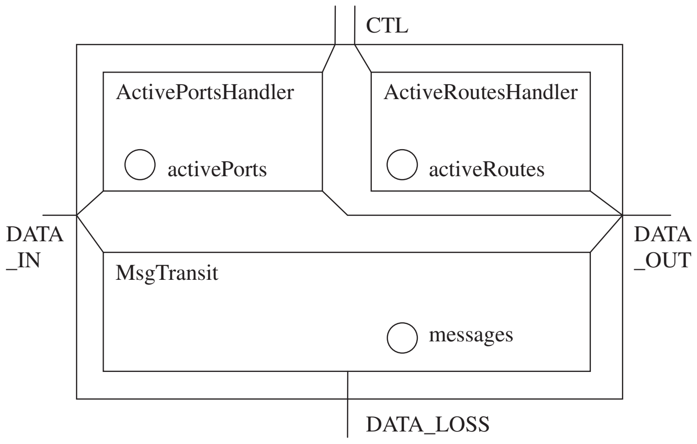

Fig. 1. Constraint-composition for Al's Node

constraints, each insisting on some subset of the externally visible events. Typically, these subsets are partially overlapped. A constraint C may encapsulate data structures; these are not directly accessible by the other constraints. Based on their values, C expresses pre-conditions that concur in determining the 'when' (ordering constraints) and the 'what' (constraints on parameters) of its controlled events. On other parameters of these events, as well as on the other events, C has no influence. An event occurrence has an impact on the data structures of all the constraints that support the event (post-conditions).

Informally, the constraints for Al's Node are as follows:

MsgTransit Messages consisting of a (*data*, *route*) pair are input by the system and stored. Each of the stored messages is eventually lost or output.

ActivePorts Messages can be input and output only at one of the currently active ports. The set of currently active ports can be updated, by adding one new port at a time.

ActiveRoutes A (*data*, *route*) stored message can be output only at a port which is associated, according to the current active routings, to the message's route. The current active routings can be updated, by adding one new (*route*, *port*) pair at a time.

The above description contains the same information as the previous, monolithic, informal description of the system, but organizes it into three partial behavioural views according to a principle of separation of concerns intended to favour both the writing and the reading of this type of documentation.

3.2 Constraint-oriented specification in LOTOS

The three constraints of Al's Node behaviour are directly expressed, in LOTOS, by the following three processes. We use the same constants, predefined sets and events of the monolithic specification, and a constant *NoMsg*, that is required to be different from any message. Note that process *MsgTransit* is in turn defined in terms of an auxiliary process *OneMsg*. Ultimately, *MsgTransit* is formed by N interleaved instances of process *OneMsg*, each temporarily holding one message, or no message. We write *msg*[1] and *msg*[2] for denoting the two components of a message.

```
PROCESS MsgTransit[DATA_IN, DATA_OUT, DATA_LOSS] (k: Nat)
:=
   [] [k > 1] -> OneMsg[DATA_IN, DATA_OUT, DATA_LOSS](NoMsg, Empty)
                 |||
                 MsgTransit[DATA_IN, DATA_OUT, DATA_LOSS](k-1)
   [] [k = 1] -> OneMsg[DATA_IN, DATA_OUT, DATA_LOSS](NoMsg, Empty)

WHERE

PROCESS OneMsg[DATA_IN, DATA_OUT, DATA_LOSS]
   (msg: Data x Routes, status: {Empty, Full})
:=
   [] [status = Empty] -> DATA_IN ?d: Data ?r: Routes ?p: Ports;
      LET msg' = <d, r>, status' = Full IN  OneMsg[---] (msg', status')
   [] [status = Full] -> DATA_OUT !msg[1] !msg[2] ?p: Ports;
      LET msg' = NoMsg, status' = Empty IN  OneMsg[---] (msg', status')
   [] [status = Full] -> DATA_LOSS !msg[1] !msg[2];
      LET msg' = NoMsg, status' = Empty IN  OneMsg[---] (msg', status')
ENDPROC (* OneMsg *)

ENDPROC (* MsgTransit *)
```

```
PROCESS ActivePortsHandler[DATA_IN, DATA_OUT, CTL]
   (ActivePorts: SUBSET of Ports)
:=
   [] DATA_IN ?d: Data ?r: Routes ?p: Ports [port in activePorts]
         ActivePortsHandler[---](---)
   [] DATA_OUT ?d: Data ?r: Routes ?p: Ports [port in activePorts]
         ActivePortsHandler[---](---)
   [] CTL ?p: Ports [port in activePorts]
         LET activePorts' = activePorts \union {p} IN
         ActivePortsHandler[---](activePorts')
ENDPROC (* ActivePortsHandler *)

PROCESS ActiveRoutesHandler[DATA_IN, DATA_OUT, CTL]
   (ActiveRoutes: SUBSET of (Routes X Ports))
:=
   [] DATA_OUT ?d: Data ?r: Routes ?p: Ports [<d, r> in ActiveRoutes];
         ActiveRoutesHandler[---](---)
   [] CTL ?r: Routes ?p: Ports [<r, p> notIn in ActiveRoutes];
         LET activeRoutes' = activeRoutes \union {<r, p>} IN
         ActiveRoutesHandler[---](activePorts')
ENDPROC (* ActivePortsHandler *)
```

Once the three constraints are defined as processes, their composition is described by a parallel expression, which directly reflects the cooperation pattern of Figure 1. This if found in the body of process *AlsNodeCO* ('CO' stands for constraint-oriented).

```
PROCESS AlsNodeCO[DATA_IN, DATA_OUT, DATA_LOSS, CTL]
      (n: Nat)
:=
            MsgTransit[DATA_IN, DATA_OUT, DATA_LOSS](n)
      |[DATA_IN, DATA_OUT]|
            (ActivePortsHandler[DATA_IN, DATA_OUT, CTL](EmptySet)
            |[DATA_OUT]|
             ActiveRoutesHandler[DATA_OUT, CTL](EmptySet)
            )
ENDPROC (* AlsNodeCO *)
```

3.3 First constraint oriented specification in TLA+

The challenge we pose now, in moving to state-based specification, and to TLA+ in particular, is to try and preserve the three-fold structure of the constraint-oriented specification, first by independently describing the three constraints, and then by trying to combine them in a compact expression.

Even more ambitiously, we try to do that by using only what we might call 'basic TLA+'. In the introduction of [7] Lamport indicates that the basic concepts introduced in Part I of his book should enable the reader to handle most of the specification problems one is likely to encounter in ordinary engineering practice. Thus, we stick to those basic concepts and investigate the extent to which constraint-oriented specification fits into this picture. An additional, pragmatic reason for this restriction is that we want to verify our specifications by the TLC tool, which currently does not handle the composite specifications discussed in Part II of that book ('More advanced topics').

In basic TLA+ one models a complex component by writing a complex action. The action consists of a disjunction of more elementary actions, each describing some event possibility. For writing our constraint-oriented specification we use the conventional, write-only, event variable e, and set two requirements:

- Each constraint must be described by a separate action, handling a disjoint portion of the global state, namely the bag of messages, the set of active ports, and the set of active routes; these actions must share only the generic event variable, and must cooperate in defining its value, according to constraint groupings that depend on the type of event, as illustrated in Figure 1.
- The global system (action *Next*) must only refer to those three actions, not to their sub-components.

Composition can only be logical conjuntion or disjunction; disjunction is readily excluded – it is too weak – and we are left with the tentative definition:

$$\mathit{Next} \triangleq \mathit{MsgTransit} \land \mathit{ActivePortsHandler} \land \mathit{ActiveRoutesHandler}$$

The problem now is one of defining the three component actions in such a way that the global behaviour is as expected. Since each component is going to be a disjunction of sub-components, and some sub-components contain only partial descriptions of events, to be conjoined with complementary, partial descriptions of the same event in other constraints, we need to make sure that the global conjunction induce all the desired pairings among the disjuncts, but *only* those pairings.

To this purpose, we design our first TLA+ specification according to the following three criteria:

- We explicitly identify, for each constraint, the subset of events of primary interest (called *KeyEvents*): these are events that involve the reading and/or writing of the state variables the constraint is in charge of.
- Each constraint C is described as the disjunction of two cases: if the occurring global event is one of the key events, then C contributes to it by a disjunction of sub-components, say C_1, ..., C_n, each one describing a different type or subtype of event; if the global event is not a key event, then C contributes by 'neutrally' allowing the event to occur ('don't care') and by simply making sure that its own state variables are unaffected.
- As required, the sub-components C_1, ..., C_n of constraint C shall only manipulate the set, say C_{vars}, of state variables controlled by the constraint. However, if some C_i updates only *some* of these variables, it will also make sure that the other variables of C_{vars} are unaffected.

The need to preserve the value of some variables across a step is a consequence of the fact that in TLA+ the variables not explicitly controlled by an action can assume arbitrary values.

TLA+ module *AlsNodeCO1* below directly reflects the requirements and criteria above.

```
──────────────────── MODULE AlsNodeCO1 ────────────────────
EXTENDS AlsNodeInterface, Naturals, FiniteSets, TLC

VARIABLES
        msg,            array of N messages in transit
        ctl,            array of N control states
        activePorts,    set of active ports
        activeRoutes    the dynamic association route-port
───────────────────────────────────────────────────────────
TypeInvariant ≜
   ∧ EventTypeInvariant
   ∧ msg ∈ [1 .. N → ((Data × Routes) ∪ {⟨⟩})]
   ∧ ctl   ∈ [1 .. N → {"empty", "full"}]
   ∧ activePorts ∈ SUBSET  Ports
   ∧ activeRoutes ∈ SUBSET (Routes × Ports)
───────────────────────────────────────────────────────────
EventSubset(key) ≜ {ee ∈ EventSet : ee ≠ ⟨⟩ ∧ ee[1] = key}
───────────────────────────────────────────────────────────
MsgTransit_In ≜
        ∃ i ∈ 1 .. N, d ∈ Data, route ∈ Routes, port ∈ Ports :
              ∧ ctl[i] = "empty"
              ∧ ctl′ = [ctl EXCEPT ![i] = "full"]
              ∧ e′ = ⟨"DATA_IN", d, route, port⟩
              ∧ msg′ = [msg EXCEPT ![i] = ⟨d, route⟩]

MsgTransit_Out ≜
        ∃ i ∈ 1 .. N, d ∈ Data, route ∈ Routes, port ∈ Ports :
              ∧ ctl[i]   = "full"
              ∧ msg[i] = ⟨d, route⟩
              ∧ ctl′ = [ctl EXCEPT ![i] = "empty"]
              ∧ e′ = ⟨"DATA_OUT", msg[i][1], msg[i][2], port⟩
              ∧ msg′ = [msg EXCEPT ![i] = ⟨⟩]

MsgTransit_Loss ≜
        ∃ i ∈ 1 .. N, d ∈ Data, route ∈ Routes :
              ∧ ctl[i]   = "full"
              ∧ msg[i] = ⟨d, route⟩
              ∧ ctl′ = [ctl EXCEPT ![i] = "empty"]
              ∧ e′ = ⟨"DATA_LOSS", msg[i][1], msg[i][2]⟩
              ∧ msg′ = [msg EXCEPT ![i] = ⟨⟩]
```

```
MsgTransit ≜
        LET KeyEvents ≜
                EventSubset("DATA_IN")
                ∪ EventSubset("DATA_OUT")
                ∪ EventSubset("DATA_LOSS")
        IN
                ∨    ∧    e′ ∈ KeyEvents
                     ∧    ∨ MsgTransit_In
                          ∨ MsgTransit_Out
                          ∨ MsgTransit_Loss
                ∨    ∧    e′ ∈ EventSet \ KeyEvents
                     ∧    UNCHANGED ⟨msg, ctl⟩
```

```
ActivePortsHandler_ControlIn ≜ ...
ActivePortsHandler_DataIn ≜ ...
ActivePortsHandler_DataOut ≜ ...
ActivePortsHandler ≜ ...
```

```
ActiveRoutesHandler_ControlIn ≜ ...
ActiveRoutesHandler_DataOut ≜ ...
ActiveRoutesHandler ≜ ...
```

```
Init ≜
        ∧ e = ⟨⟩
        ∧ msg = [i ∈ (1 .. N) ↦ ⟨⟩]
        ∧ ctl  = [i ∈ (1 .. N) ↦  "empty"]
        ∧ activePorts = {}
        ∧ activeRoutes = {}

Next ≜
        ∧ MsgTransit
        ∧ ActivePortsHandler
        ∧ ActiveRoutesHandler

Spec ≜ Init ∧ □[Next]_⟨e, msg, ctl, activePorts, activeRoutes⟩

 AlsNodeCO2Instance ≜ INSTANCE AlsNodeCO2
 AlsNodeCO2Spec ≜ AlsNodeCO2Instance!Spec
```

```
 THEOREM Spec ⇒ □TypeInvariant
 THEOREM Spec ⇒ AlsNodeMonol!Spec
 THEOREM Spec ⇒ AlsNodeCO2Spec
```

3.4 Second constraint oriented specification in TLA+

A second approach to the constraint-oriented specification of Al's Node in TLA+ is suggested by considering the two-step procedure introduced in [14] for transforming a LOTOS multiple parallel expression into a set of algebraic expressions describing explicitly the process groupings that support the events at the different gates.

Let us apply the first step of that simple procedure to the top behaviour expression of LOTOS process *AlsNodeCO*. For every gate G we rewrite the parallel expression as follows: we replace a process instantiation by the plain process name, if G is in the gate list of that process, and by a zero otherwise; and we replace a parallel operator by a product operator, if G is in the list of synchronization gates, and by a sum operator, if it is not. We thus obtain the following set of gate-labelled algebraic expressions:

```
DATA_IN:     MsgTransit * (ActivePortsHandler + 0)
DATA_OUT:    MsgTransit * (ActivePortsHandler * ActiveRoutesHandler)
DATA_LOSS:   MsgTransit + (0 * 0)
CTL:         0 + (ActivePortsHandler + ActiveRoutesHandler)
```

By the second transformation step, these expressions are flattened into SOP (sum of products) form:

```
DATA_IN:     MsgTransit * ActivePortsHandler
DATA_OUT:    MsgTransit * ActivePortsHandler * ActiveRoutesHandler
DATA_LOSS:   MsgTransit
CTL:         ActivePortsHandler + ActiveRoutesHandler
```

Each gate-labelled SOP represents now the possible groupings of processes that support the events at that gate (see [14] for details).

Module *AlsNodeCO2* below represents an alternative, constraint-oriented specification of Al's Node in which the global *Next* action is formed by exactly the five disjuncts above (interpreting sum and product as disjunction and conjunction, respectively). Each disjunct refers to a specific event type, as identified by the different gates. Notice that we could make use of the same elementary actions used in module *AlsNodeCO1*, which is instantiated with name *CO1*.

MODULE *AlsNodeCO2*

EXTENDS *AlsNodeInterface*, *Naturals*, *FiniteSets*

VARIABLES

msg,	array of N messages in transit
ctl,	array of N control states
activePorts,	set of active ports
activeRoutes	the dynamic association route-port

$CO1 \triangleq$ INSTANCE *AlsNodeCO1*

```
$Init \triangleq CO1!Init$
$TypeInvariant \triangleq CO1!TypeInvariant$
$EventSubset(x) \triangleq CO1!EventSubset(x)$
```

```
$MsgTransit \triangleq$
                 $\lor CO1!MsgTransit\_In$
                 $\lor CO1!MsgTransit\_Out$
                 $\lor CO1!MsgTransit\_Loss$

$ActivePortsHandler \triangleq$
                 $\lor \quad CO1!ActivePortsHandler\_ControlIn$
                 $\lor \quad CO1!ActivePortsHandler\_DataIn$
                 $\lor \quad CO1!ActivePortsHandler\_DataOut$

$ActiveRoutesHandler \triangleq$
                 $\lor CO1!ActiveRoutesHandler\_ControlIn$
                 $\lor CO1!ActiveRoutesHandler\_DataOut$
```

```
$Next \triangleq$
        $\lor$      $\land e' \in EventSubset(\text{"DATA\_IN"})$
                    $\land MsgTransit$
                    $\land ActivePortsHandler$
                    $\land \text{UNCHANGED } activeRoutes$

        $\lor$      $\land e' \in EventSubset(\text{"DATA\_OUT"})$
                    $\land MsgTransit$
                    $\land ActivePortsHandler$
                    $\land ActiveRoutesHandler$

        $\lor$      $\land e' \in EventSubset(\text{"DATA\_LOSS"})$
                    $\land MsgTransit$
                    $\land \text{UNCHANGED } activePorts$
                    $\land \text{UNCHANGED } activeRoutes$

        $\lor$      $\land \exists p \in Ports : e' = \langle \text{"CTL"}, p \rangle$
                    $\land \text{UNCHANGED } \langle msg, ctl \rangle$
                    $\land ActivePortsHandler$
                    $\land \text{UNCHANGED } activeRoutes$

        $\lor$      $\land \exists r \in Routes, p \in Ports : e' = \langle \text{"CTL"}, r, p \rangle$
                    $\land \text{UNCHANGED } \langle msg, ctl \rangle$
                    $\land \text{UNCHANGED } activePorts$
                    $\land ActiveRoutesHandler$

$Spec \triangleq Init \land \Box[Next]_{\langle e, msg, ctl, activePorts, activeRoutes \rangle}$

$AlsNodeCO1Spec \triangleq CO1!Spec$
```

```
THEOREM  $Spec \Rightarrow \Box TypeInvariant$
THEOREM  $Spec \Rightarrow AlsNodeMonol!Spec$
THEOREM  $Spec \Rightarrow AlsNodeCO1Spec$
```

Care should be taken in preventing uncontrolled behaviour of state variables that are not explicitly handled by a product of elementary actions. Thus, the disjuncts in the body of *Next*, derived from the SOP's, are enriched by UNCHANGED clauses that handle those excluded variables: in this way, each disjunct covers the complete global state.

4 Verification

Are the two constraint-oriented TLA+ specifications correct implementations of the more abstract, monolithic specification of Al's Node? In TLA, implementation is implication. Thus, let us start by verifying, using the TLC model checker, that the *Spec* in module *AlsNodeCO1* implies the *Spec* in module *AlsNodeMonol*. Recall that the latter is obtained from module *InternalAlsNodeMonol* by hiding all variables except the event variable *e*. The proof consists then in exhibiting a *refinement mapping* that relates the variables of *AlsNodeCO1* to those of *InternalAlsNodeMonol*. Module *MCAlsNodeCO1* below (MC stands for 'Model Checker') extends module *AlsNodeCO1* by adding the definition of the desired refinement mapping.

```
MODULE $MCAlsNodeCO1$
EXTENDS    $Bags,\ AlsNodeCO1$

$omsgBag \triangleq$        a state function of $\langle ctl,\ msg \rangle$
  LET $f[k \in (0 \,..\, N)] \triangleq$
    IF $k > 0$
      THEN
        IF $ctl[k] =$ "full"
            THEN $SetToBag(\{msg[k]\}) \oplus f[k-1]$
            ELSE $f[k-1]$
      ELSE $EmptyBag$
  IN  $f[N]$

$AN \triangleq$ INSTANCE $InternalAlsNodeMonol$  WITH  $msgBag \leftarrow omsgBag$

$ImplementationProperty \triangleq AN!Spec$

THEOREM   $Spec \Rightarrow ImplementationProperty$
```

The theorem at the end of the module states that if the tuple of variables (*e*, *msg*, *ctl*, *activePorts*, *activeRoutes*) behaves as specified by the *Spec* in module *AlsNodeCO1*, then the tuple (*e*, *omsgBag*, *activePorts*, *activeRoutes*) behaves as specified by the *Spec* in module *InternalAlsNode*, with *omsgBag* playing the role of *msgBag*. The refinement mapping provides 'witnesses' for the hidden variables *msgBag*, *activePorts*, *activeRoutes* of the *Spec* in module *AlsNodeMonol*; in particular, the implementation handles exactly the same data structures *activePorts* and *activeRoutes* of the internal, abstract specification, so that the refinement mapping for them is just the identity function (see [7] for further discussion on refinement mappings, and on the way TLC supports these proofs). The module required for proving that also our second TLA+ constraint-oriented specification implements the initial, monolithic specification, is identical to the module above, except that the EXTENDS clause has to refer to *AlsNodeCO2*.

Although the two constraint-oriented specifications of Al's Node have been developed by following different intuitions, a closer comparison of the definitions of *MsgTransit*, *ActiveRoutesHandler*, *ActivePortsHandler*, and *Next* in the two modules suggests that they might be logically equivalent. Rather than manually rewriting one into the other, we run again TLC, and prove their equivalence by showing that they imply each other (note that they manipulate exactly the same state variables). The two relevant theorems appear at the bottom of modules *AlsNodeCO1* and *AlsNodeCO2*. (For convenience of exposition, the two modules we show end up instantiating each other; of course TLC cannot handle this circularity, and before running the tool, one has to edit them so that they refer to each other in turns.)

5 Conclusions

We have investigated the possibility to export what we consider an elegant, event-based specification style, based on constraint composition, to a state-based formalism such as TLA+. For doing so, we have introduced in our TLA+ specifications a special purpose, write-only variable, conventionally called *e*, which plays the role of LOTOS gates and events. When composing TLA+ actions by logical conjunction, this variable plays a crucial role in selecting the desired pairings of disjuncts, while filtering out the undesired ones.

The main advantage of using a process-algebraic formalism for writing specifications in constraint-oriented style is one of conciseness: the ad-hoc specialized operator of selective synchrony allows one to capture constraint composition by compact parallel composition expressions. Not surprisingly, using the more generic, logical conjunction operator generally leads to longer specifications. In particular, one has to split the specification effort between two complementary concerns: describing the changes of some variables, and making sure that other variables preserve their values. This feature, known as the 'frame problem', is shared by other logic-based formalisms, e.g. Z, while is completely absent in event-based formalisms.

However, moving to a state-based, and logic-based setting, while retaining event-oriented thinking, may offer some benefits:

– The formalism is more basic, more general, and can be learnt more easily.
– Specifications can be manipulated and transformed by simple and universally recognized laws of logics, rather than by specialized algebraic laws for behavioural operators.
– When events are assimilated to state variables, they inherit all the manipulation techniques available for the latter. For example, one can express different levels of visibility, for a given specification, by simultaneously hiding some components of the global state and some components of the event.

This last circumstance enables interesting expressive and analytical possibilities that we have only started to explore.

Acknowledgments. I express my gratitude to Leslie Lamport and Egon Boerger, for various stimulating discussions on the state-based vs. event-based diatribe.

References

1. Hoare, T., He, J.: Unifying Theories of Programming. Prentice Hall (1998)
2. Woodcock, J.C.P., Cavalcanti, A.L.C.: The semantics of Circus. In Bert, D., Bowen, J., Henson, M.C., Robinson, K., eds.: ZB 2002: Formal Specification and Development in Z and B, Springer-Verlag (2002) 184–203 Lecture Notes in Computer Science, 2272.
3. Gurevich, Y.: Evolving Algebras 1993 - Lipari Guide. In Boerger, E., ed.: Specification and Validation Methods, Oxford Univ. Press (1995) 9–36
4. Boerger, E., Staerk, R.: Abstract State Machines - A Method for High-Level System Design and Analysis. Springer (2003)
5. Abrial, J.R.: The B-Book - Assigning Programs to Meanings. Cambridge Univ. Press (1996)
6. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems **16** (1994) 872–923
7. Lamport, L.: Specifying Systems - The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2003)
8. Spivey, J.M.: The Z Notation - A Reference manual. Prentice-Hall (1989)
9. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
10. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science. Springer-Verlag (1980)
11. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems **14** (1987) 25–59
12. 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)
13. Quemada, J., Azcorra, A.: A constraint oriented specification of al's node. In van Eijk, P.H.J., Vissers, C.A., Diaz, M., eds.: The Formal Description Technique LOTOS, North-Holland (1989) 83–88
14. Bolognesi, T.: Deriving graphical representations of process networks from algebraic expressions. Information Processing Letters **46** (1993) 289–294