Enhanced event structures: towards a true concurrency semantics for E-LOTOS

M. Kapus-Kolar [1]

Jožef Stefan Institute, Jamova 39, SI-1111 Ljubljana, Slovenia

Received 15 September 2005

Abstract

E-LOTOS is a standard process-algebraic language for formal specification of real-time concurrent and reactive systems. Its originally defined semantics is based on interleaving of events. In the present paper, we propose an enhanced kind of event structures and show how to employ them to give E-LOTOS processes a branching-time true concurrency semantics. The proposed event structures can model real-time processes with data handling and excel in concise representation of event renaming and synchronization.

Key words: Event structures, Process algebra, E-LOTOS, Formal semantics, True concurrency

1. Introduction

E-LOTOS [4,15], an enhanced successor of LOTOS [3,1], is one of the standard languages for formal specification of real-time concurrent and reactive systems. According to the operational semantics given in [4], an E-LOTOS specification characterizes a process by its readiness to engage into various kinds of atomic instantaneous events, where all internal process events are by definition anonymous and all concurrent events are represented as interleaved. That reflects the fact that LOTOS was originally intended for specification of "temporal ordering of observational behaviour" [3], where the possibility of simultaneous events was neglected.

Such characterization of a process often fails to provide sufficient information for its further refinement. For example, when refining an event into a process, one must know the relations of causality and conflict in which the event is engaged, because at least some of the events constituting its refinement are supposed to inherit them [6]. One might also want to identify events which are truly concurrent, so that their execution may be delegated to different concurrent components of a system. Hence, it is convenient to model a process by its events and their relationships, i.e. by its *event structure*.

With their detailed representation of process behaviour, event-structure models are ideal not only for refinement of events, but also for refinement of their relationships, necessary, for example, when designing a distributed implementation of a process, i.e. refining the relationships into a coordina-

[1] Tel.: +386 14773531; fax: +386 14262102.
E-mail address: monika.kapus-kolar@ijs.si.

tion protocol [14]. On the other hand, event structures refrain from modelling process architecture, following the idea that grouping of events into subprocesses is, like their assignment to gates, just a matter of interpretation [10]. Hence, an event structure is indeed just a collection of events and relationships, i.e. a set of orthogonal process properties.

The syntactic and semantic simplicity makes event structures easy to use and ideal for incremental design. Elements of an event structure may be added and removed at will, although it is advisable to take care that the structure stays within a class which can be easily manipulated with the available tools (e.g. to avoid causal ambiguity [16]). As increasingly more powerful manipulation tools are available, we in this paper limit our attention to the expressiveness of event structures.

A process-algebraic specification describes a set of elementary processes and their hierarchical composition. Elementary processes often correspond to individual events and composition operators provide information on their relationships. The problem is that when the operands of a composition operator are themselves compound processes, the relationships between their constituent events are described only implicitly. To overcome the problem, process-algebraic languages are being furnished with event-structure semantics.

For LOTOS without data, an event-structure semantics was proposed in [11]. The semantics was extended to timed processes [9], and subsequently employed [2] (still without data) for ET-LOTOS [12], a predecessor of E-LOTOS. In the present paper, we propose an event-structure semantics for E-LOTOS.

The proposed true concurrency semantics for E-LOTOS is not the only contribution of the paper. Perhaps even more important is the newly developed kind of event structures, which we name *enhanced event structures*, because in the name E-LOTOS, "E" stands for "enhanced". Enhanced event structures can model real-time processes with data handling and excel in concise representation of event renaming and synchronization.

The paper is organized as follows: In Section 2, we study the intuitive semantics of E-LOTOS processes and gradually develop a kind of event structures sufficiently expressive for their elegant modelling. Section 3 contains a detailed discussion of event-structure semantics of elementary E-LOTOS processes and of individual process composition operators. Section 4 concludes the paper.

2. Enhanced event structures

2.1. *Events*

The main objects in an event structure $\mathcal{E}$ are its events e, collected in an E. An e represents atomic, instantaneous execution of some tasks. Hence, it can be seen also as a boolean variable jumping from false (e has not yet occurred) to true (e has already occurred).

The elementary events of an E-LOTOS process B, i.e. the potential events of its elementary subprocesses, with no doubt correspond to the event concept defined above, and as such qualify for inclusion into E of the event structure modelling B. The past practice has been to also include into E the compound events of B [11,9,2], i.e. interactions of its subprocesses. We find the approach contraintuitive, because in the original E-LOTOS semantics, the behaviour of a B in no way depends on how its already executed elementary events have synchronized. In other words, while a B remembers the occurrence of its elementary events, it does not remember the occurrence of its compound events. Therefore, we propose that the members of E are exactly the potential elementary events of the modelled B. Thereby, the size of E is kept proportional to the number of the elementary subprocesses of B, while otherwise it could grow exponentially with the number of the parallel compositions specified for the processes.

2.2. *Preconditions and their triggers*

An e can occur only if it is currently logically enabled, i.e. if it has not been disabled or if all its disablings have been cancelled. In E-LOTOS, an e in a B might be disabled from the beginning or upon the occurrence of a disabling e', while cancellation of a disabling, i.e. enabling or re-enabling of e, re-

quires appropriate values of the input parameters of B and/or execution of a set of events whose occurrence and the data they have generated justify the cancellation.

In LOTOS or ET-LOTOS, an already enabled e might be disabled, but is never re-enabled. Hence, enabling and disabling of events, i.e. the relations of causality and conflict, are two separate issues. In E-LOTOS, its suspend/resume operator introduces resolvable conflicts. Examples of event structures capable to model resolvable conflicts are [5,13,7], but none of them addresses data and time.

Trying to represent enabling and disabling of events in an integrated manner, we observe that for every E-LOTOS event e, there exists a (possibly empty) set of preconditions, constraints expected to be satisfied just before the occurrence of e. In an $\mathcal{E}$, let preconditions ξ be objects of a special kind, collected in a Ξ. For an e, let $\Xi(e)$ list the associated ξ. For a ξ, let $E(\xi)$ list the e with ξ in $\Xi(e)$.

For every ξ and e in $E(\xi)$, we introduce a boolean trigger $\tau(e,\xi)$ and define that e is logically enabled if for every ξ in $\Xi(e)$, $\tau(e,\xi)$ is false or ξ is true. Besides, if a $\tau(e,\xi)$ is false just before the occurence of e, it must be false also just after it. Hence, a $\tau(e,\xi)$ indicates a disabling of e for which ξ is the cancelling condition. The approach is a generalization of that in [5].

2.3. *Postconditions and their triggers*

In E-LOTOS, the exact manner in which a gate event will occur (what its execution time and other generated data will be) is not known in advance, but there is in principle a constraint expected to be satisfied just after the occurrence, i.e. a postcondition constraint. In an $\mathcal{E}$, let postconditions λ be objects of a special kind, collected in a Λ. For an e, let $\Lambda(e)$ list the associated λ. For a λ, let $E(\lambda)$ list the e with λ in $\Lambda(e)$.

For every λ and e in $E(\lambda)$, we introduce a boolean trigger $\tau(e,\lambda)$ and define that just after the occurrence of e, every λ in $\Lambda(e)$ with $\tau(e,\lambda)$ true just before e must be true. Selectively triggered postconditions facilitate full control of event simultaneity, like the event structures proposed in [13]. One might want to use them to selectively restrict simultaneity of otherwise independent events.

Example 1 *To specify that an e_1 must occur simultaneously with an e_2 or an e_3, but not with both, one could introduce into $\Lambda(e_1)$ postconditions "$e_2 \wedge e_3$", "$e_2 \oplus e_3$" and "*false*", with triggers "$e_2 \oplus e_3$", "$\neg e_2 \wedge \neg e_3$" and "$e_2 \wedge e_3$", respectively.*

2.4. *Aging and urgency*

In the following, the term "time" denotes the internal time of an evolving $\mathcal{E}$ or the modelled process, i.e. the value of its clock, that starts from 0. For simplicity, we assume that the time domain is discrete. During the dynamic evolution of an $\mathcal{E}$, the flow of time is reflected in the aging of its events. Whenever the clock increases, so does the age $a(e)$ of every e currently in the state of idling. An e is in the state of idling when it has not yet occurred, but is logically enabled. When an e finally occurs, its aging stops, i.e. $a(e)$ reaches its final value, the relative execution time (RET) of e.

All idling events by definition age at the speed of the clock, which an urgent event might freeze to zero, trying to enforce occurrence before the particular time instant is over. For an e, let $\phi(e)$ indicate whether it is currently absolutely urgent, while $\varphi(e)$ regulates its conditional urgency. An idling e is willing to age with a non-zero speed when $\phi(e)$ is false and e cannot immediately occur in a manner satisfying $\varphi(e)$ as a postcondition. A $\varphi(e)$ typically requires that e occurs under a name implying its urgency.

2.5. *Contexts*

If an e belongs to a B, it also belongs to all its superprocesses. Each such process represents a different context c, though the concept of a context is more general than the concept of a process.

For an e, let $C(e)$ list the contexts in which it is embedded. A $C(e)$ must be non-empty, so that it contains at least c_e, the home context of e. For a c, let $E(c)$ list its constituent e. We introduce no a priori restriction on how $E(c)$ and $E(c')$ may be related in the case of $(c \neq c')$.

2.6. *Event names*

When an e occurs, it occurs in every c in $C(e)$, under a specific name $n(e,c)$, with $n(e,c_e)$ the basic name of e. Event names n are objects of a special kind, collected in an N. For an e, let $N(e)$ list its associated n. For every $e' \neq e$, $N(e)$ and $N(e')$ must be disjoint.

Among the names defined for an e in an $\mathcal{E}$ modelling a process B, let $n(e)$ denote the one by which e presents itself to the environment of B, i.e. the external name of e. An only member of an $N(e)$ is $n(e)$ by definition.

Names $n(e,c)$ of an e can be seen also as values finalized upon the occurrence of e. Referring to the value of an $n(e,c)$ without first checking or securing that e has already occurred is not safe.

In E-LOTOS, an $n(e,c_e)$ often consists of several parts, many of them interpreted as data variables set by e and influencing execution of the remaining events, while the other $n(e,c)$ represent various renamings of e. As data generation is in E-LOTOS confined to events, we define that the only data containers of an $\mathcal{E}$, besides its input parameters collected in an I, are its e (boolean variables jumping from false to true), its $n(e,c)$ and its $a(e)$.

2.7. *Meeting points*

When an e occurs in a c, it occurs at one of the meeting points of c, abstract places at which events in $E(c)$ can, if necessary, synchronize with respect to c. While E-LOTOS process gates are an architectural concept, meeting points are just an auxiliary concept simplifying specification of event synchronization, which is itself just specifically interpreted event simultaneity and can as well be specified by triggered postconditions.

Meeting points m are objects of a special kind, collected in an M. For a c, let $M(c)$ list its constituent m. For every $c' \neq c$, $M(c)$ and $M(c')$ must be disjoint.

2.8. *Roles*

When some events synchronize in an m, each of them appears in a particular role. For example, in a synchronization of some processes B_i, each participating e from a B_i takes the role "a participant from B_i".

Roles r are objects of a special kind, collected in an R. For an m, let $R(m)$ list the r available to events e in synchronizations at m. For every $m' \neq m$, $R(m)$ and $R(m')$ must be disjoint. For an r in $R(m)$ of an m in an $M(c)$, let $E(r)$ list the e in $E(c)$ allowed to take the role, and $N(r)$ the names $n(e,c)$ of e in $E(r)$.

Now we are ready to degrade contexts from a primary concept into a derived structural property, so that they completely loose the flavour of processes. For an n, m or r, let $c(n)$, $c(m)$ or $c(r)$, respectively, denote the context to which it belongs. We define that two elements of $(N \cup M \cup R)$ belong to the same context exactly if this can be inferred by merely interpreting every $(n \in N(r))$ as $(c(n) = c(r))$ and every $(r \in R(m))$ as $(c(r) = c(m))$. An e is in an $E(c)$ exactly if an n in $N(e)$ belongs to c. The resulting partitioning must satisfy all the rules above, plus the implicit assumption that no two members of an $N(e)$ belong to the same context, i.e. that for every e and c' in $C(e)$, $n(e,c')$ is the only n in $N(e)$ with $(c(n) = c')$.

2.9. *Event occurrences*

An $\mathcal{E}$ evolves by aging and by event occurrences o. An o represents simultaneous occurrence of events in a non-empty set $E(o)$, where every e in $E(o)$ satisfies all its currently active pre- and postconditions. Two different o and o' of the same run are by definition not simultaneous, but the fact that they occur in succession does not imply that they do not occur upon the same value of the clock.

An o consists of synchronizations s in a non-empty set $S(o)$. An s represents synchronized occurrence of events in a non-empty set $E(s)$. An $E(o)$ is a disjoint union of $E(s)$ with s in $S(o)$.

For an s, let $C(s)$ list the c in $C(e)$ with e in $E(s)$. For every c in $C(s)$, s activates an $m(c,s)$ in $M(c)$, the m in which s occurs with respect to c. For every e in $E(s)$ and c in $C(e)$, s activates an $r(e,c,s)$ in $R(m(c,s))$ with e in $E(r(e,c,s))$, the r played by e in s with respect to c, where every r in $R(m(c,s))$ corresponds to at least one $r(e,c,s)$.

In E-LOTOS, compound events are hierarchical compositions of elementary events, with event composition a synonym for playing complementary roles. Hence, let every s secure that for any two different e and e' in $E(s)$, there is a c in $(C(e) \cap C(e'))$ with $(r(e,c,s) \neq r(e',c,s))$.

In E-LOTOS, every synchronization of events involves unification of their corresponding names, that must include the name of the gate. Hence, let every s secure that for every c in $C(s)$, all $n(e,c)$ with e in $E(s)$ and c in $C(e)$ are set to the same value in a domain statically defined by a $\mu(m(c,s))$, the naming constraint of $m(c,s)$.

In E-LOTOS, a compound event of a process B is often synchronized also with the environment of B, in an implicitly present synchronization context. Hence, let every s secure that all $n(e)$ with e in $E(s)$ are set to the same value representing the external name $n(s)$ of s. With this restriction, we are able to define that the external name $n(o)$ of an o is the set of all $n(s)$ with s in $S(o)$. As two or more members of an $n(o)$ might be finalized to the same value, an $n(o)$ might be finalized to a multiset.

3. True concurrency semantics of E-LOTOS processes

3.1. *Preliminaries*

Event structures and their attributes will be, wherever necessary to avoid ambiguities, decorated with the name of the E-LOTOS process for which they have been defined.

For an E-LOTOS e, let B_e denote the smallest B comprising it. Whenever an e is introduced into $\mathcal{E}_{B_e}$ during its construction, let $n(e,c_e)$ and an m_e whose $R(m_e)$ is an $\{r_e\}$ whose $N(r_e)$ is $\{n(e,c_e)\}$ be introduced by default.

In an $\mathcal{E}_B$, let every $\Lambda(e)$ comprise at least $\lambda(e)$, the naming and timing constraint of e. The defaults for a $\mu(m)$, a $\phi(e)$, a $\varphi(e)$, a $\tau(e,\xi)$ and a $\tau(e,\lambda)$ are "true", "false", "false", "true" and "true", respectively. E, N, M, R, Ξ and Λ, respectively, consist exactly of the e, n, m, r, ξ or λ introduced explicitly or present by default. I by default just covers the needs of $\mathcal{E}_B$ for input data.

3.2. *E-LOTOS event names*

An E-LOTOS $n(e,c)$ is a vector of one or more fields. Depending on whether the first field is finalized to $\mathbf{i}$, to δ, to a G from a universe $\mathcal{G}$, or to an X from a universe $\mathcal{X}$, $n(e,c)$ presents e as an anonymous unobservable action, as a successful termination, as an action on gate G, or as an exception signal X. The other fields of $n(e,c)$ represent the data carried by e in the context c. The data fields and their subfields may themselves be vectors of fields.

To facilitate access to individual fields of an n, each field F is associated with a key K uniquely identifying it among the components of the vector to which it belongs. The default key of the k-th component of a vector is "$\$k$". By writing "$K \Rightarrow F$" instead of "$F$", one explicitly shows that K is a key to F.

Example 2 *For an n finalized to a "$G(1,(\text{false},2))$", complete revealing of the default keys of its fields gives "$\$1 \Rightarrow G, \$2 \Rightarrow (\$1 \Rightarrow 1, \$2 \Rightarrow (\$1 \Rightarrow \text{false}, \$2 \Rightarrow 2))$". The default unique global identifier of the value "*false*" is "$n.\$2.\$2.\$1$".*

For an E-LOTOS process B, E_B is partitioned into a Δ_B, a Σ_B and an A_B, respectively listing the potential successful termination events, exception events, and true or dummy actions, i.e. the e for which $\lambda(e)_{B_e}$ sets $n(e,c_e)$ to a $\delta(\ldots)$, to an exception name, or to none of the two.

3.3. *Data expressions*

In many cases, the E-LOTOS dynamic semantics requires evaluation of a data expression. In some cases, it is foreseen that the evaluation might terminate by an exception, i.e. unsuccessfully. For all the cases, it is appropriate to foresee that the evaluation might as well block the system without previously signalling an exception, e.g. by engaging into an infinite computation. For all the other cases, we implicitly assume that the evaluation successfully produces a value of the expected type. In particular, successful evaluation is assumed for every boolean expression acting as a constraint. If a constraint is not explicitely specified, it is by default "true".

3.4. *Terminations*

The E-LOTOS dynamic semantics extensively refers to the readiness of a B to immediately successfully terminate or signal an exception [4]. Therefore, we introduce for every $\mathcal{E}_B$ a dynamic attribute Trm defined as follows: When the evolving $\mathcal{E}_B$ is ready to immediately exhibit successful termination or an exception of B, the value of Trm is the pending termination name, i.e. a $\delta(\ldots)$ or an $X(\ldots)$, otherwise the value is "**none**".

Whenever the value of Trm is a $\delta(Data)$, $Data$ is of the form "$V_1 \Rightarrow Val_1, \ldots, V_k \Rightarrow Val_k$", where V_1 to V_k are the members of Bnd_B, the set of the variables that B might bind on its way towards successful termination, and Val_1 to Val_k are their respective values upon the successful termination of B. For a B, let Ok_B indicate whether it might successfully terminate. If it is false, Bnd_B is empty. For every $\mathcal{E}_B$, Trm will be explicitly constructed as a function of its input parameters, its event occurrences, the data generated by the past events, and the current age of the pending events.

3.5. *Inaction*

"**stop**" denotes inaction, i.e. a B that neither exhibits any o nor blocks the clock [4]. It is appropriate that in $\mathcal{E}_B$, E is empty.

Ok is false. Trm is "**none**".

3.6. *Time block*

"**block**" denotes time block, i.e. a B exhibiting no o, but blocking the clock [4]. An appropriate $\mathcal{E}_B$ is as follows:

There is a dummy e with $\phi(e)$ "true" responsible for blocking the clock. As e is not supposed to occur, $\lambda(e)$ is "false", implying that e is in A_B.

Ok is false. Trm is "**none**".

3.7. *Successful termination*

An "**exit**$(Bnds)$" denotes a B whose only event is a conditionally urgent successful termination $\delta(Data)$ leading to a time block, where $Data$ denotes the variable bindings specified by $Bnds$ [4]. An appropriate $\mathcal{E}_B$ is as follows:

There are an e with $\varphi(e)$ "true", the successful termination event, and a dummy e' with $\phi(e')$ "true" responsible for blocking the clock after the occurrence of e.

$\lambda(e)$ requires that $n(e, c_e)$ is finalized to $\delta(Data)$. $\lambda(e')$ is "false".

As e' is supposed to be guarded by e, $\Xi(e')$ contains a ξ "e".

Ok is true. Bnd lists the variables bound by $Bnds$. Trm is "**if** $\neg e$ **then** $\delta(Bnds)$ **else none endif**".

Example 3 *"***exit**$(V_1 \Rightarrow 1, V_2 \Rightarrow V_3)$*" specifies a B executing a $\delta(Data)$ in which $Data$ consists of the constant 1 and the value of the input parameter V_3 being assigned to variables V_1 and V_2, respectively. In $\mathcal{E}_B$, $\lambda(e)$ requires that $n(e, c_e)$ is an "$F_1(V_1 \Rightarrow F_2, V_2 \Rightarrow F_3)$" with F_1, F_2 and F_3 finalized to δ, 1 and the value of V_3, respectively.*

3.8. *Assignment*

A "$Ptr := Expr$" basically denotes a B whose only event is a conditionally urgent successful termination $\delta(Data)$ leading to a time block, with $Data$ representing the variable bindings produced when the expression $Expr$ is successfully evaluated and its value is matched to the pattern Ptr [4]. It might, however, happen that the evaluation does not terminate or terminates by raising of an exception. In such a case, B is equivalent to "**block**" or to the exception raising, respectively. An appropriate $\mathcal{E}_B$ is as follows:

There are an e with $\varphi(e)$ "true", the successful termination event, an e' with $\varphi(e')$ "true", the exception event, and a dummy e'' with $\phi(e'')$ "true" responsible for blocking the clock after the occurrence of e or e' or when the evaluation of $Expr$ is non-terminating.

$\lambda(e)$ requires that $n(e, c_e)$ is finalized to $\delta(Data)$ with $Data$ representing the bindings produced when Ptr is matched to the value of $Expr$, if successfully computed. $\lambda(e')$ requires that $n(e', c_{e'})$ is finalized to the exception raised by $Expr$, if any. $\lambda(e'')$ is "false".

$\Xi(e)$ contains a ξ true exactly when the evalu-

ation of $Expr$ successfully terminates. $\Xi(e')$ contains a ξ' true exactly when the evaluation of $Expr$ raises an exception. $\Xi(e'')$ contains a ξ'' true exactly when $(e \vee e')$ or when the evaluation of $Expr$ is non-terminating.

Ok is true exactly if $Expr$ might successfully terminate. In that case, Bnd lists the variables bound by Ptr.

Trm is equivalent to "$\delta(Data)$" when $(\xi \wedge \neg e)$, to the pending exception name when $(\xi' \wedge \neg e')$, and to "**none**" otherwise.

Example 4 *"$(?V_1, ?V_2) :=$ **if** V_4 **then** $(1, V_3)$ **else raise** $X(V_4)$ **endif** " specifies a B that, depending on V_4, sets variable V_1 to 1 and variable V_2 to the value of the input parameter V_3, or raises exception $X(V_4)$. In $\mathcal{E}_B$, Trm is equivalent to "**if** $e \vee e'$ **then none elseif** V_4 **then** $\delta(V_1 \Rightarrow 1, V_2 \Rightarrow V_3)$ **else** $X(V_4)$ **endif** ".*

3.9. *Delay*

A "**wait**$(Expr)$" basically denotes a B whose only event is a conditionally urgent successful termination "$\delta()$" leading to a time block and not executable before the age determined by the value of the expression $Expr$ [4]. It might, however, happen that the evaluation of $Expr$ does not terminate or terminates by raising of an exception. In such a case, B is equivalent to "**block**" or to the exception raising, respectively.

Without loss of generality, we assume that whenever B is enabled, the input parameters of $Expr$ secure its successful evaluation. If this is not the case, rewrite B into "**any** : **any** $:=$ $Expr; B$", where ";" is the operator of sequential composition described in Section 3.16. The dummy assignment "**any** : **any** $:=$ $Expr$" will implement the "**block**" or the exception raising of $Expr$, if any, and enable B only in the case of successful evaluation of $Expr$.

With the assumption, an appropriate $\mathcal{E}_B$ is the $\mathcal{E}$ of "**exit**()" (see Section 3.7) modified as follows: $\lambda(e)$ additionally requires that $a(e)$ is finalized to a value not less than the value of $Expr$.

Ok is true. Bnd is empty. Trm is "**if** $\neg e \wedge (a(e) \geq Expr)$ **then** $\delta()$ **else none endif**".

3.10. *Internal action*

"**i**" denotes a B executing an immediate **i**() followed by "**exit**()" [4]. An appropriate $\mathcal{E}_B$ is the $\mathcal{E}$ of "**exit**()" (see Section 3.7) enhanced as follows:

There is an e'' with $\varphi(e'')$ "true", the internal action. $\lambda(e'')$ requires that $n(e'', c_{e''})$ is finalized to **i**().

As e is supposed to be guarded by e'', $\Xi(e)$ contains a ξ' "e''".

Ok is true. Bnd is empty. Trm is "**if** $e'' \wedge \neg e$ **then** $\delta()$ **else none endif**".

3.11. *Exception signalling*

A "**signal** $X(Expr)$" basically denotes a B immediately issuing an exception signal $X(Data)$ followed by "**exit**()", where $Data$ is the value of the expression $Expr$ [4]. It might, however, happen that the evaluation of $Expr$ does not terminate or terminates by raising of an exception. In such a case, B is equivalent to "**block**" or to the exception raising, respectively.

Without loss of generality, we assume that whenever B is enabled, the input parameters of $Expr$ secure its successful evaluation. If this is not the case, rewrite B into "**any** : **any** $:=$ $Expr; B$".

With the assumption, an appropriate $\mathcal{E}_B$ is the $\mathcal{E}$ of "**i**" (see Section 3.10) modified as follows: $\lambda(e'')$ requires that $n(e'', c_{e''})$ is finalized to $X(Data)$ with $Data$ the value of $Expr$.

Ok is true. Bnd is empty. Trm is "**if** $\neg e''$ **then** $X(Expr)$ **elseif** $\neg e$ **then** $\delta()$ **else none endif**".

3.12. *Gate action*

A "$G\ Ptr_1@Ptr_2[Cnst]$" denotes a B willing to execute an action named $G(Data)$, on the gate G, with the data $Data$ matching the pattern Ptr_1, with a RET $Time$ matching the pattern Ptr_2, with $Data$ and $Time$ satisfying $Cnst$ [4]. The action is not urgent, but if it does occur, it is followed by an "**exit**$(Bnds)$" with $Bnds$ specifying that the data carried by the δ event represents the variable bindings produced when $Data$ and $Time$ are matched to Ptr_1 and Ptr_2, respectively.

An appropriate $\mathcal{E}_B$ is the $\mathcal{E}$ of "**exit**$(Bnds)$" (see Section 3.7) enhanced as follows:

There is an e'', the gate action. $\lambda(e'')$ requires that $n(e'', c_{e''})$ is finalized to a $G(Data)$ with $Data$ matching Ptr_1, that $a(e'')$ is finalized to a $Time$ matching Ptr_2, and that $Data$ and $Time$ satisfy $Cnst$.

As e is supposed to be guarded by e'', $\Xi(e)$ contains a ξ' "e''".

$n(e, c_e)$ has a named field for every variable bound by e''. For each of the fields, $\lambda(e)$ requires that it is finalized to the same value as the corresponding field of $n(e'', c_{e''})$ (if the variable is bound by Ptr_1) or as $a(e'')$ (if the variable is bound by Ptr_2).

Ok is true exactly if $Cnst$ is satisfiable. In that case, Bnd lists the variables bound by Ptr_1 or Ptr_2. Trm is "**if** $e'' \wedge \neg e$ **then** $\delta(Bnds)$ **else none endif**".

Example 5 *"$G(?V_1 : \text{int})@?V_2[(V_1 = 4) \wedge (V_2 < V_0)]$" specifies a B with a gate action $G(4)$ with a RET less than V_0, the only input parameter of B. The action binds V_1 to 4 and V_2 to the RET.*

In $\mathcal{E}_B$, $\lambda(e'')$ requires that $n(e'', c_{e''})$ is an "$F_1(F_2)$" with F_1, F_2 and $a(e'')$ finalized to G, 4 and a time value less than V_0, respectively.

$\lambda(e)$ requires that $n(e, c_e)$ is an "$F_1(V_1 \Rightarrow F_2, V_2 \Rightarrow F_3)$" with F_1, F_2 and F_3 finalized to δ, to the same value as "$n(e'', c_{e''}).\$2.\$1$", and to the same value as $a(e'')$, respectively.

*Trm is "***if** $e'' \wedge \neg e$ **then** $\delta(V_1 \Rightarrow n(e'', c_{e''}).\$2.\$1, V_2 \Rightarrow a(e''))$ **else none endif**".*

3.13. *Event renaming*

A "**rename** Ren **in** B_1 **endren**" denotes a B behaving as B_1 with its events renamed as specified by the renaming Ren, that is basically intended for renaming gate actions into gate actions and exceptions into exceptions [4]. We assume that the semantics of the renaming operator is as proposed in [8], i.e. that Ren is a relation between the old and the new names possibly defining for a gate action multiple alternative presentations. For exceptions, successful terminations, internal actions and gate actions which it does not effect, Ren is by definition a function.

An appropriate $\mathcal{E}_B$ is $\mathcal{E}_{B_1}$ enhanced as follows:

For every e in $(A \cup \Sigma)$, one adds an $n(e)_B$ and an m_e whose $R(m_e)$ is an $\{r_e\}$ whose $N(r_e)$ is $\{n(e)_B\}$. Besides, $\lambda(e)$ additionally requires that $n(e)_{B_1}$ and $n(e)_B$ are related as specified by Ren.

Ok_B is Ok_{B_1}. Bnd_B is Bnd_{B_1}. Trm_B returns the result of Trm_{B_1} renamed as specified by Ren.

Example 6 *"**rename action** $G(?V_1 : \text{nat})$ **is** $G'(?V_2 : \text{nat})[V_1 \leq V_2 < V_3]$ **in** B_1 **endren**" specifies a B behaving as B_1 with every action $G(V_1)$ with a natural V_1 less than V_3, an input parameter of B, allowed to present itself with any new name $G'(V_2)$ with a natural V_2 between V_1, inclusively, and V_3.*

In $\mathcal{E}_B$, $\lambda(e)$ for e in $(A \cup \Sigma)$ additionally requires that in the case of $n(e)_{B_1}$ finalized to a $G(V_1)$ with a natural V_1 less than V_3, $n(e)_B$ is finalized to one of the expected $G'(V_2)$, while otherwise, it is finalized to the same value as $n(e)_{B_1}$.

Suppose that B_1 executes a compound event s consisting of elementary events e_1 and e_2 with $n(e_1)_{B_1}$ and $n(e_2)_{B_1}$ finalized to $G(1)$, while V_3 is 3. For each e_i, the candidate values for $n(e_i)_B$ are $G'(1)$ and $G'(2)$. The semantics of $\mathcal{E}_B$ secures that $n(e_1)_B$ and $n(e_2)_B$ are finalized to the same value (see Section 2.9), the $n(s)_B$ related to $n(s)_{B_1}$ exactly as specified by the renaming.

3.14. *Action hiding*

A "**hide** Hid **in** B_1 **endhide**" denotes a B behaving as B_1 with its gate actions selectively converted into internal actions, where the selection is specified by Hid [4]. The hiding operator renames the actions into $\mathbf{i}()$ and makes them urgent. If only the renaming is considered, B is equivalent to a "**ren** Ren **in** B_1 **endren**".

An appropriate $\mathcal{E}_B$ is the $\mathcal{E}$ of "**ren** Ren **in** B_1 **endren**" (see Section 3.13) modified as follows: For every e for which the renaming introduces a new name $n(e, c)$, $\varphi(e)$ is extended with "$\vee(n(e, c) = \mathbf{i}())$".

Ok_B is Ok_{B_1}. Bnd_B is Bnd_{B_1}. Trm_B is Trm_{B_1}.

3.15. *Nondeterministic assignment*

A "Ptr := **any** $Type[Cnst]$" basically denotes a B executing an immediate $\mathbf{i}()$ which matches

the pattern Ptr to a nondeterministically selected value of the type $Type$ satisfying $Cnst$ and is followed by an "**exit**($Bnds$)" with $Bnds$ specifying that the data carried by the δ event represents the variable bindings produced by the matching [4]. It might, however, happen that no such value exists, so that B is equivalent to "**block**".

An appropriate $\mathcal{E}_B$ is the $\mathcal{E}$ of a "**hide** G **in** $GPtr_1$@!0[$Cnst$] **endhide**" with G a dummy gate and Ptr_1 "$Ptr : Type$" (see Sections 3.12 and 3.14) modified as follows: For the e'' representing the hidden action on gate G, $\phi(e'')$ is "true".

3.16. *Sequential composition*

A "$B_1; B_2$" denotes a B executing B_1 and B_2 in a sequence, with B_2 assuming control when B_1 becomes ready for successful termination [4]. An appropriate $\mathcal{E}_B$ is constructed as follows:

Step 1: To secure that δ events of B provide a value for every V in Bnd_B, δ events of B_2 are enhanced with information on bindings in B_1. In $\mathcal{E}_{B_2}$, one adds for every e in Δ an $n(e)_B$ and an m_e whose $R(m_e)$ is an $\{r_e\}$ whose $N(r_e)$ is $\{n(e)_B\}$. Besides, $\lambda(e)$ additionally requires that whenever $n(e)_{B_2}$ is finalized to a $\delta(Data)$, $n(e)_B$ is finalized to a $\delta(Data')$ with $Data'$ providing a value for every V in Bnd_B, where the value is "V" for V in $(Bnd_B \setminus Bnd_{B_2})$ and as in $Data$ for V in Bnd_{B_2}.

Step 2: To make the variable bindings produced by B_1 available to B_2, every reference to a V in Bnd_{B_1} is in $\mathcal{E}_{B_2}$ and in Trm_{B_2} replaced by "$Trm_{B_1}.\$2.V$".

Step 3: To prevent premature enabling of B_2, every $\Xi_{B_2}(e)$ is extended with a ξ true exactly when Trm_{B_1} returns a $\delta(\ldots)$.

Step 4: To prevent the occurrence of δ events of B_1 (they must be "trapped" and handled by B_2), $\lambda(e)$ is for every e in Δ_{B_1} changed into "false".

Step 5: The constituent objects of the obtained $\mathcal{E}_{B_1}$ and $\mathcal{E}_{B_2}$ are promoted into objects of $\mathcal{E}_B$.

Ok_B is $(Ok_{B_1} \wedge Ok_{B_2})$. If it is true, Bnd_B is $(Bnd_{B_1} \cup Bnd_{B_2})$.

When Trm_{B_1} returns a $\delta(\ldots)$, Trm_B returns the result of Trm_{B_2}. Otherwise, it returns the result of Trm_{B_1}.

Example 7 *Take a B of the form "$B_4; B_3$" where B_4 is "$B_1; B_2$", B_1 is "***wait**(2)*", B_2 is "$?V_1 := 0$" and B_3 is "$?V_2 := V_1 + 1$". Let e_1 to e_3 be the δ events of B_1 to B_3, respectively.*

*Trm_{B_1} is "***if** $\neg e_1 \wedge (a(e_1) \geq 2)$ **then** $\delta()$ **else none endif**". Trm_{B_2} is equivalent to "***if** $\neg e_2$ **then** $\delta(V_1 \Rightarrow 0)$ **else none endif**". Constructing $\mathcal{E}_{B_4}$, one adds precondition "$\neg e_1 \wedge (a(e_1) \geq 2)$" for every e in E_{B_2}. Trm_{B_4} is equivalent to "***if** $\neg e_1 \wedge (a(e_1) \geq 2) \wedge \neg e_2$ **then** $\delta(V_1 \Rightarrow 0)$ **else none endif**".*

$\lambda(e_3)_{B_3}$ requires "$n(e_3)_{B_3} = \delta(V_2 \Rightarrow (V_1 + 1))$". Constructing $\mathcal{E}_B$, one provides e_3 with a new external name $n(e_3)_B$ and enhances $\lambda(e_3)$ with "$n(e_3)_B = \delta(V_1 \Rightarrow V_1, V_2 \Rightarrow (V_1 + 1))$", that is subsequently enhanced into "$n(e_3)_B = \delta(V_1 \Rightarrow Trm_{B_4}.\$2.V_1, V_2 \Rightarrow (Trm_{B_4}.\$2.V_1 + 1))$".

*Ξ_{B_3} is extended with precondition "$\neg e_1 \wedge (a(e) \geq 2) \wedge \neg e_2$" for every e in E_{B_3}. Trm_{B_3} is equivalent to "***if** $\neg e_3$ **then** $\delta(V_2 \Rightarrow (V_1 + 1))$ **else none endif**" and enhanced into "***if** $\neg e_3$ **then** $\delta(V_2 \Rightarrow (Trm_{B_4}.\$2.V_1 + 1))$ **else none endif**". Trm_B is equivalent to "***if** $\neg e_1 \wedge (a(e_1) \geq 2) \wedge \neg e_2 \wedge \neg e_3$ **then** $\delta(V_1 \Rightarrow 0, V_2 \Rightarrow 1)$ **else none endif**".*

3.17. *Variable declaration*

A "**var** $V_1 : Type_1 := Expr_1, \ldots, V_k : Type_k := Expr_k$ **in** B_1 **endvar**", where all "$:= Expr_i$" with V_i not an input parameter of B_i are optional, denotes a B representing B_1 with the listed variables V_i, of type $Type_i$, internalized and initialized to the value of $Expr_i$ [4].

Without loss of generality, we make the following assumptions, which one should try to satisfy exactly in the given order:

1) No $Expr_i$ refers to V_1 to V_k. If an $Expr_i$ does refer to a V_j, i.e. to the input parameter V_j of B, change "$V_j := Expr_j$" into a "$V'_j := Expr_j$" with V'_j a distinct new name, and in B_1, every reference or assignment to V_j into a reference or assignment, respectively, to V'_j.

2) All $Expr_i$ are void. If an $Expr_i$ is not, change "$V_i : Type_i := Expr_i$" into "$V_i : Type_i$" and enhance B_1 into "$?V_i := Expr_i; B_1$".

With the assumptions, an appropriate $\mathcal{E}_B$ is $\mathcal{E}_{B_1}$ enhanced as follows:

For every e in Δ, one adds an $n(e)_B$ and an m_e whose $R(m_e)$ is an $\{r_e\}$ whose $N(r_e)$ is $\{n(e)_B\}$.

Besides, $\lambda(e)$ additionally requires that whenever $n(e)_{B_1}$ is finalized to a $\delta(Data)$, $n(e)_B$ is finalized to a $\delta(Data')$ with $Data'$ representing $Data$ with the bindings of variables V_1 to V_k removed.

Ok_B is Ok_{B_1}. Bnd_B is $(Bnd_{B_1} \setminus \{V_1, \ldots, V_k\})$.

Trm_B returns the result of Trm_{B_1}, except in the case of successful termination, when it returns the result with the bindings of variables V_1 to V_k removed.

3.18. *Exception handling*

A "**trap exception** $X_1(Inp_1)$ **is** B_1 **endexn** ... **exception** $X_k(Inp_k)$ **is** B_k **endexn exit** Inp_{k+1} **is** B_{k+1} **endexit in** B_{k+2} **endtrap**", where the "**exit** ... **endexit**" part is optional, denotes a B which basically executes B_{k+2}, but if B_{k+2} becomes ready for a trapped successful termination or exception, control is transferred to the corresponding handler B_i, with Inp_i specifying how the data carried by the termination of B_{k+2} is interpreted as input data of B_i [4]. When an exception in B_{k+2} is handled, all the bindings produced by B_{k+2} before the exception are ignored.

Without loss of generality, we make the following assumptions, which one should try to satisfy exactly in the given order:

1) The "**exit** ..." part is not void. If it is, enhance it into "**exit is exit**() **endexit**".

2) There is no Inp_i binding a V from a Bnd_{B_j}. If an Inp_i does bind such a V, of a type $Type$, let Inp_i bind a new V' instead and enhance B_i into "**var** $V : Type := V'$ **in** B_i **endvar**".

3) Every Bnd_{B_i} with $(i \leq (k+1))$ and Ok_{B_i} is Bnd_B. If a V is missing in a Bnd_{B_i}, enhance B_i into "$?V := V; B_i$".

With the assumptions, an appropriate $\mathcal{E}_B$ is constructed as follows:

Step 1: To implement the required transfer of data from B_{k+2} to the handlers of its terminations, every reference to a V bound by an Inp_i is in $\mathcal{E}_{B_i}$ and in Trm_{B_i} replaced by the value of V expressed as a function of $Trm_{B_{k+2}}$ as specified by Inp_i under the assumption that $Trm_{B_{k+2}}$ returns a termination activating B_i.

Step 2: To prevent premature enabling of the termination handlers, every $\Xi_{B_i}(e)$ with $(i \leq k+1)$ is extended with a ξ_i true exactly when $Trm_{B_{k+2}}$ returns a termination enabling B_i.

Step 3: To prevent the occurrence of the terminations of B_{k+2} trapped in B, $\lambda(e)$ is for every e in $(\Delta_{B_{k+2}} \cup \Sigma_{B_{k+2}})$ extended with the requirement that $n(e)_{B_{k+2}}$ is not finalized to a name of such a termination.

Step 4: The constituent objects of all the obtained $\mathcal{E}_{B_i}$ are promoted into objects of $\mathcal{E}_B$.

Ok_B is $((Ok_{B_{k+1}} \wedge Ok_{B_{k+2}}) \vee \exists i \leq k.Ok_{B_i})$. A V is in Bnd_B if it is in a Bnd_{B_i} with $(i \leq k)$. If $(Ok_{B_{k+1}} \wedge Ok_{B_{k+2}})$, a V is in Bnd_B also if it is in $Bnd_{B_{k+1}}$, or if it is in Bnd_{k+2} and its binding in B_{k+2} is not trapped by Inp_{k+1} [15].

When $Trm_{B_{k+2}}$ returns a termination handled by a B_i, Trm_B returns the result of Trm_{B_i}. Otherwise, it returns the result of $Trm_{B_{k+2}}$.

3.19. *Iteration*

A "**loop** X **in** B' **endloop**" basically denotes a B executing an infinite sequence of instances of B' [4]. It might, however, happen that the current instance of B' becomes ready to signal an exception $X(\ldots)$ and B consequently successfully terminates by executing $\delta()$.

Constructing an appropriate $\mathcal{E}_B$, one rewrites B into a "**trap exception** $X()$ **is exit**() **endexn in** B'' **endtrap**" with B'' "$B_1; B_2; \ldots$" with B_i instances of B', constructs $\mathcal{E}_{B''}$ and enhances it into $\mathcal{E}_B$ as described in Section 3.18. $\mathcal{E}_{B''}$ is constructed as follows:

Step 1: Every reference to a V in $Bnd_{B'}$ in an $\mathcal{E}_{B_i}$ with $(i > 1)$ is replaced by "$Trm_{B_1;\ldots;B_{i-1}}.\$2.V$", to make the variable bindings produced by B_j with $(j < i)$ available to B_i.

Step 2: To prevent premature enabling of a B_i, every $\Xi_{B_i}(e)$ with $(i > 1)$ is extended with a ξ_i true exactly when $Trm_{B_1;\ldots;B_{i-1}}$ returns a $\delta(\ldots)$.

Step 3: To prevent the occurrence of δ events of a B_i, $\lambda(e)$ is for every e in a Δ_{B_i} changed into "false".

Step 4: The constituent objects of the obtained $\mathcal{E}_{B_i}$ are promoted into objects of $\mathcal{E}_{B''}$.

$Ok_{B''}$ is false. $Trm_{B''}$ returns the result of to the $Trm_{B_1;\ldots;B_i}$ not returning a $\delta(\ldots)$ while $Trm_{B_1;\ldots;B_{i-1}}$ does, if any, and "**none**" otherwise.

Let us note that originally, the most general

kind of a breakable loop is "**loop** $X : Type$ **in** B' **endloop**", equivalent to "**trap exception** $X(?V : Type)$ **is exit**(V) **endexn in** B'' **endtrap**" [4]. However, we find this construct incompatible with the rest of the standard, because the process returns a value, like a data expression would, while E-LOTOS processes are in principle supposed to return only variable bindings.

3.20. *Case*

A "**case** $Expr$ **is** $Ptr_1[Cnst_1] \rightarrow B_1 | \ldots | Ptr_k [Cnst_k] \rightarrow B_k$ **endcase**" basically denotes a B executing the first B_i among B_1 to B_k whose pattern Ptr_i matches the value of the expression $Expr$ in a manner satisfying $Cnst_i$ [4]. If such a B_i does not exist, B raises a special exception "Match()". It might, however, happen that the evaluation of $Expr$ does not terminate or terminates by raising an exception. In such a case, B is equivalent to "**block**" or to the exception raising, respectively.

Without loss of generality, we make the following assumptions, which one should try to satisfy exactly in the given order:

1) Evaluation of $Expr$ always successfully terminates. If this is not the case, rewrite B into "**any** : **any** := $Expr$; B".

2) Evaluation of $Expr$ always results in selection of a B_i. If this is not the case, add another alternative "**any** : **any** $\rightarrow$ (**signal** Match(); **block**)".

3) There is no Ptr_i binding a V from a Bnd_{B_j}. If a Ptr_i does bind such a V, of a type $Type$, let Ptr_i bind, and $Cnst_i$ restrict, a new V' instead, and enhance B_i into "**var** $V : Type := V'$ **in** B_i **endvar**".

4) Every Bnd_{B_i} with Ok_{B_i} is Bnd_B. If a V is missing in a Bnd_{B_i}, enhance B_i into "$?V := V$; B_i".

With the assumptions, an appropriate $\mathcal{E}_B$ is constructed as follows:

Step 1: To implement the required transfer of data from $Expr$ to a B_i, every reference to a V bound by a Ptr_i is in $\mathcal{E}_{B_i}$ and in Trm_{B_i} replaced by the value of V expressed as a function of $Expr$ as specified by Ptr_i under the assumption that $Expr$ returns a value activating B_i.

Step 2: Every $\Xi_{B_i}(e)$ is extended with a ξ_i true exactly when after successful evaluation of $Expr$, B_i is the selected alternative.

Step 3: The constituent objects of the obtained $\mathcal{E}_{B_i}$ are promoted into objects of $\mathcal{E}_B$.

Ok_B is true exactly if it is possible that ξ_i is true for an i with Ok_{B_i} true. A V is in Bnd_B if it is possible that ξ_i is true for an i with V in Bnd_{B_i}.

Trm_B returns the result of the Trm_{B_i} with ξ_i true.

3.21. *Choice*

A "**sel** $B_1 [] \ldots [] B_k$ **endsel**" denotes a B running processes B_i in parallel until an action selecting a B_j occurs and permanently disables the other alternatives [4]. The action may be the first event in B_j or an auxiliary $\mathbf{i}()$ executed when B_j is ready for a $\delta(\ldots)$ or an exception as its first event. The prefixing of potentially decisive termination events with an $\mathbf{i}()$ is necessary for prevention of the time nondeterminism which could otherwise result from their trapping. Unlike [4], we do not insist that processes B_i must be processes unable to successfully terminate without previously executing an action or signalling an exception.

Without loss of generality, we assume that every Bnd_{B_i} with Ok_{B_i} is Bnd_B. If a V is missing in a Bnd_{B_i}, enhance B_i into "$?V := V; B_i$". With the assumption, an appropriate $\mathcal{E}_B$ is constructed as follows:

Step 1: In every $\mathcal{E}_{B_i}$, E is extended with an e_i with $\varphi(e_i)$ "true", the auxiliary action for selecting B_i. $\lambda(e_i)$ requires that $n(e_i, c_{e_i})$ is finalized to $\mathbf{i}()$.

Step 2: To secure that an auxiliary $\mathbf{i}()$ occurs only when necessary, every $\Xi_{B_i}(e_i)$ is extended with a ξ_i true exactly when Trm_{B_i} returns a termination and no e in the original A_{B_i} has occurred so far.

Step 3: To secure that the first event in B is not a $\delta(\ldots)$ or an exception, every $\Xi_{B_i}(e)$ with e in $(\Delta_{B_i} \cup \Sigma_{B_i})$ is extended with a ξ_i' true exactly when an e' in A_{B_i} has already occurred.

Step 4: To secure that selection of a B_j permanently disables every B_i with $(i \neq j)$, every $\Xi_{B_i}(e)$ is for every $(j \neq i)$ extended with a ξ_i^j "false", where every $\tau(e, \xi_i^j)$ is "ξ_j'".

Step 5: The constituent objects of the obtained $\mathcal{E}_{B_i}$ are promoted into objects of $\mathcal{E}_B$.

Ok_B is true exactly if an Ok_{B_i} is. A V is in Bnd_B exactly if it is in a Bnd_{B_i}.

Trm_B is equivalent to the Trm_{B_i} with ξ'_i true, if any, and to "**none**" otherwise.

3.22. *Choice over values*

The event structure composition procedure from Section 3.21 imposes no restrictions on the non-empty set from which indexes i and j of the constituent alternatives are drawn. Hence, the procedure can be employed also as a concise description of the $\mathcal{E}_B$ belonging to a B specified as a "**choice** $Ptr[]B'$ **endch**", i.e. behaving as if choosing between alternatives B_{Val} of the form "**case** Val **is** $Ptr \rightarrow B'$ **endcase**" for Val ranging over the values matching the pattern Ptr, and yet another alternative "**stop**" [4].

3.23. *Suspend/resume*

A "$B_0[X>B'$", where it is assumed that B' is a process unable to signal an exception $X(\ldots)$ without previously executing an action or signalling another exception, denotes a B running "**sel** $B'_0[]B_i$ **endsel**" with B'_0 the current residuum of B_0 (the initial residuum of B_0 is B_0 itself) and B_i the current one among the consecutive instances $B_1, B_2, \ldots$ of B' run by B, until an action resolves the choice in favour of B_i and suspends B_0 until an $X(\ldots)$ is trapped in B_i and control is transferred to "$B'_0[X>B'$" with $B_{i+1}, B_{i+2}, \ldots$ the remaning instances of B', so that B_0 is resumed [4]. Bindings made in the earlier instances of B' are not visible in its current instance. Note that unlike in "**sel** $B_0[]B'$ **endsel**", every (even a non-initial) $\delta(\ldots)$ or exception signal of B_0 is prefixed with its own auxiliary $\mathbf{i}()$, where the $\mathbf{i}()$ introducing successful termination of B_0 permanently disables the current and the following instances of B'. Unlike [4], we do not insist that B' must be a process unable to successfully terminate without previously executing an action or signalling an exception.

Without loss of generality, we make the following assumptions:

1) If Ok_{B_0}, Bnd_{B_0} is Bnd_B. If a V is missing in Bnd_{B_0}, enhance B_0 into "$?V := V; B_0$".

2) If $Ok_{B'}$, $Bnd_{B'}$ is Bnd_B. If a V is missing in $Bnd_{B'}$, enhance B' into "$?V := V; B'$".

With the assumptions, an appropriate $\mathcal{E}_B$ is constructed as follows:

Step 1: In $\mathcal{E}_{B_0}$, E is for every e in Σ extended with an e' with $\varphi(e')$ "true", the auxiliary prefix of e. $\lambda(e')$ requires that $n(e', c_{e'})$ is finalized to $\mathbf{i}()$.

Step 2: To secure that the prefix of an exception signal in B_0 occurs only when necessary, one in $\mathcal{E}_{B_0}$ for every e in Σ, its prefix e' and ξ in $\Xi(e)$ replaces e in $E(\xi)$ with e' and accordingly renames $\tau(e, \xi)$ into $\tau(e', \xi)$, so that e' is logically enabled exactly when e would originally be. As e' may occur only if e is not trapped, $\Xi(e')$ is extended with a $\xi_{e'}$ true exactly when $\lambda(e)$ upon logical enabling of e provides a value for $n(e)_{B_0}$.

Step 3: To secure that an exception signal in B_0 occurs only after its prefix, every $\Xi_{B_0}(e)$ with e in Σ_{B_0} is extended with a ξ_e "e'" with e' the prefix of e.

Step 4: In $\mathcal{E}_{B_0}$, E is extended with an e_δ with $\varphi(e_\delta)$ "true", the auxiliary prefix of the successful termination of B_0. $\lambda(e_\delta)$ requires that $n(e_\delta, c_{e_\delta})$ is finalized to $\mathbf{i}()$.

Step 5: To secure that e_δ occurs when B_0 would originally successfully terminate, Ξ_{B_0} is extended with a ξ_δ true exactly when Trm_{B_0} returns a $\delta(\ldots)$.

Step 6: To secure that successful termination of B_0 occurs after e_δ, every $\Xi_{B_0}(e)$ with e in Δ_{B_0} is extended with a ξ'_δ "e_δ", while the other members of.

Step 7: For every instance B_i of B', $\mathcal{E}_{B_i}$ is modified as in Steps 1 to 3 in Section 3.21, so that an e_i representing an auxiliary prefix $\mathbf{i}()$ is introduced.

Step 8: To secure that B' properly suspends and re-enables B_0, every $\Xi_{B_0}(e)$ is for every instance B_i of B' extended with a ξ^i_0 true exactly when Trm_{B_i} returns an $X(\ldots)$, where $\tau(e, \xi^i_0)$ is true exactly if an e' in A_{B_i} has already occurred.

Step 9: To prevent premature enabling of individual instances of B', every $\Xi_{B_i}(e)$ with $(i > 1)$ is extended with a ξ''_i true exactly when $Trm_{B_{i-1}}$ returns an $X(\ldots)$.

Step 10: To prevent the occurrence of the trapped X, every $\lambda(e)$ with e in a Σ_{B_i} with $(i > 0)$ is extended with the requirement that $n(e)_{B_i}$ is not finalized to an $X(\ldots)$.

Step 11: To secure that exception prefixes of B_0

suspend instances of B' until the exception signal occurs, every $\Xi_{B_i}(e'')$ with $(i > 0)$ is for every e in Σ_{B_0} and its prefix e' extended with a ξ_i^e "e", where $\tau(e'', \xi_i^e)$ is "e'".

Step 12: To secure that e_δ permanently disables instances of B', every $\Xi_{B_i}(e)$ with $(i > 0)$ is extended with a ξ_i^δ "false", where $\tau(e, \xi_i^\delta)$ is "e_δ".

Step 13: The constituent objects of the obtained $\mathcal{E}_{B_i}$ are promoted into objects of $\mathcal{E}_B$.

Ok_B is $(Ok_{B_0} \vee Ok_{B'})$. Bnd_B is $(Bnd_{B_0} \cup Bnd_{B'})$.

When a Trm_{B_i} with $(i > 0)$ and an e in the enhanced A_{B_i} true returns a value other than an $X(\ldots)$, Trm_B returns the value. If there is no such i, Trm_B returns the result of Trm_{B_0} provided that the result is an exception and there are an e in Σ_{B_0} and its auxiliary prefix e' such that $(e' \wedge \neg e)$ is true, or that the result is a $\delta(\ldots)$ and e_δ is true. Otherwise, Trm_B returns "**none**".

3.24. *Parallel composition*

A "**par** Dgr **in** $Gts_1 \rightarrow B_1 \| \ldots \| Gts_k \rightarrow B_k$ **endpar**", where it is assumed that Bnd_{B_1} to Bnd_{B_k} are pairwise disjoint, denotes a B running B_1 to B_k in parallel, synchronized as specified by Dgr and Gts_1 to Gts_k [4]. For every B_i, Gts_i lists the gates on which B_i synchronizes with its peers. If the gate G on which a synchronization occurs has its synchronization degree D defined in Dgr, the event is a synchronization of exactly D processes B_i with G in Gts_i, otherwise it is a synchronization of all such processes. Every exception signal of a B_i is prefixed with an auxiliary $\mathbf{i}()$. B successfully terminates when all its constituents do.

An appropriate $\mathcal{E}_B$ is constructed as follows:

Step 1: For every e in a Σ_{B_i}, an auxiliary prefix e' is introduced in the same manner as Steps 1 to 3 in Section 3.23 do for e in Σ_{B_0}.

Step 2: To secure that every exception prefix in a B_i suspends every peer B_j until the exception signal occurs, every $\Xi_{B_j}(e'')$ is for every e in a Σ_{B_i} with $(i \neq j)$ and its prefix e' extended with a ξ_j^e "e", where $\tau(e'', \xi_j^e)$ is "e'".

Step 3: To secure that δ events of B provide a value for every V in Bnd_B, δ events of every B_i are enhanced with wild cards for the missing bindings. In $\mathcal{E}_{B_i}$, one adds for every e in Δ an n_e representing a new $n(e)_{B_i}$, and an m_e whose $R(m_e)$ is an $\{r_e\}$ whose $N(r_e)$ is $\{n_e\}$. Besides, $\lambda(e)$ additionally requires that whenever the original $n(e)_{B_i}$ is finalized to a $\delta(Data)$, n_e is finalized to a $\delta(Data')$ with $Data'$ providing a field for every V in Bnd_B, where for V in Bnd_{B_i}, the value of the field must be as in $Data$.

Step 4: The constituent objects of the obtained $\mathcal{E}_{B_i}$ are promoted into objects of $\mathcal{E}_B$.

Step 5: To secure the required synchronization of gate actions and successful terminations, one adds for every e in $(A \cup \Delta)$ an n'_e representing a new $n(e)_B$ and for every non-empty subset α of $\{1, \ldots, k\}$ an m_α whose $R(m_\alpha)$ for every i in α contains an r_α^i whose $N(r_\alpha^i)$ contains names n'_e for e in A_{B_i} and if α is $\{1, \ldots, k\}$, also for e in Δ_{B_i}. For every e in $(A \cup \Delta)$, $\lambda(e)$ additionally requires that n'_e is finalized to the same value as the old $n(e)_B$.

A $\mu(m_\alpha)$ requires that in every s with m_α an $m(c, s)$, the value to which the names $n(e, c)$ with e in $E(s)$ are finalized is 1) an $\mathbf{i}(\ldots)$ with α a singleton set, or 2) a $G(\ldots)$ with an i the only member of α and G not in Gts_i, or 3) a $G(\ldots)$ such that G is in Gts_i for every i in α and $|\alpha|$ is the explicitly or implicitly specified synchronization degree of G, or 4) a $\delta(\ldots)$ with α $\{1, \ldots, k\}$. If a $\mu(m_\alpha)$ is equivalent to "false", m_α may be omitted.

Ok_B is $\wedge_{i=1,\ldots,k} Ok_{B_i}$. Bnd_B is $\cup_{i=1,\ldots,k} Bnd_{B_i}$.

When a Trm_{B_i} returns an exception and there are an e in Σ_{B_i} and its auxiliary prefix e' such that $(e' \wedge \neg e)$, Trm_B returns the result of Trm_{B_i}. When Trm_{B_1} to Trm_{B_k} return some $\delta(Data_1)$ to $\delta(Data_k)$, respectively, Trm_B returns $\delta(Data_1, \ldots, Data_k)$. Otherwise, Trm_B returns "**none**".

Example 8 *In a B specified as "***par in** $[G] \rightarrow B_1 \| [G] \rightarrow B_2$ **endpar***", D_G is 2. Suppose that in $\mathcal{E}_{B_1}$, E is an $\{e_1\}$ with e_1 a permanently enabled event whose $n(e_1)$ may be finalized to $G()$ or to a $G'()$. Suppose that in $\mathcal{E}_{B_2}$, E is an $\{e_2, e'_2\}$ with e_2 and e'_2 two permanently enabled events that can occur only in synchronization, in an s whose $n(s)$ is $G()$.*

Constructing $\mathcal{E}_B$, we introduce new meeting points $m_{\{1\}}$, $m_{\{2\}}$ and $m_{\{1,2\}}$. For $m_{\{1\}}$, there is a role $r^1_{\{1\}}$ whose $E(r^1_{\{1\}})$ is $\{e_1\}$. For $m_{\{2\}}$, there is a role $r^2_{\{2\}}$ whose $E(r^2_{\{2\}})$ is $\{e_2, e'_2\}$. For $m_{\{1,2\}}$,

there are roles $r^1_{\{1,2\}}$ *and* $r^2_{\{1,2\}}$*, where* $E(r^1_{\{1,2\}})$ *is* $\{e_1\}$ *and* $E(r^2_{\{1,2\}})$ *is* $\{e_2, e'_2\}$*. If* $\mathcal{G}$ *is* $\{G, G'\}$*,* $\mu(m_{\{1\}})$ *and* $\mu(m_{\{2\}})$ *allow events* $\mathbf{i}()$ *and* $G'(\ldots)$*, while* $\mu(m_{\{1,2\}})$ *allows events* $G(\ldots)$ *and* $\delta(\ldots)$*.*

An s *in* $\mathcal{E}_B$ *can involve either* e_1 *in role* $r^1_{\{1\}}$*, or* e_1 *in role* $r^1_{\{1,2\}}$ *and* e_2 *and* e'_2 *in role* $r^2_{\{1,2\}}$*, where in the second case, the constraint that both* e_2 *and* e'_2 *must be involved is inherited from* $\mathcal{E}_{B_2}$*. The resulting* $n(s)$ *is* $G'()$ *or* $G()$*, respectively.*

3.25. *Parallel over values*

A "**par** Ptr **in** $Lst|||B'$" with Lst a list of values matching the pattern Ptr and $Bnd_{B'}$ empty denotes a B basically running a process "**case** Val **is** $Ptr \rightarrow B'$ **endcase**" for every value Val in Lst, but equivalent to "**exit**()" in the exceptional case of empty Lst [4]. Note that the length of Lst might not be known in advance.

An appropriate $\mathcal{E}_B$ is the $\mathcal{E}$ of "**par in** $[] \rightarrow B_1 || [] \rightarrow B_2 || \ldots$ **endpar**" with each B_i specified as "**case** length(Lst) $\geq i$ **is** !true $\rightarrow B'_i$|!false $\rightarrow$ **exit**() **endcase**" where B'_i is "**case** nth(Lst, i) **is** $Ptr \rightarrow B'$ **endcase**". After a parametrized description of the constituent $\mathcal{E}_i$ is obtained, the event structure composition procedure from Section 3.24 can be employed as a concise description of $\mathcal{E}_B$, since it poses no restrictions on k, the number of the constituent structures.

3.26. *Additional simultaneity restrictions*

There is a question whether two or more elementary events of an E-LOTOS process should be allowed to simultaneously occur on the same process gate without synchronizing into a compound event, i.e. whether gates are sharable resources. If the events also carry the same data, so that they are indistinguishable for the process environment, such simultaneity is even more questionable.

Where suppression of such simultaneity is absolutely necessary, it can be specified by additional triggered postconditions. We do not, however, think that the suppression should be incorporated into the E-LOTOS semantics, because many E-LOTOS specifications, particularly those extensively employing the event renaming operator, use gates just as an auxiliary concept with a vague architectural interpretation. As for externally indistinguishable events, one often intends to make them more distinct by a further refinement, implying that restricting their simultaneity in an early design phase would be premature.

4. Concluding remarks

We have proposed a new kind of event structures sufficiently expressive to model even the rich semantics of E-LOTOS processes without explosion of the event set. In the presence of timing constraints, data handling, multi-party synchronization, "m-among-n" synchronization, event renaming with action splitting [8], process suspension and resumption, and exception trapping, conceiving such a model has been quite an endeavour.

The resulting true concurrency semantics of E-LOTOS processes explicitly represents even those relationships between process events which have so far been described only informally, in various working documents and tutorials on the language. If the proposed formal semantics seems complicated, this is not because the employed modelling technique lacks expressiveness, but because the underlying intuitive semantics of the processes is complicated.

With their object-oriented representation of events and their relationships, and with their orthogonal representation of various concerns, the proposed event structures seem ideal for incremental (or even on-the-fly) process design, i.e. for addition, deletion and/or refinement of events and/or their relationships. This feature of the modelling technique is particularly welcome in the age when cooperative distributed design and execution of processes should be a routine.

References

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

[2] H. Bowman and J.-P. Katoen, A true concurrency semantics for ET-LOTOS, in: Proc. CSD'98 (IEEE Computer Society Press, 1998) 228–239.

[3] ISO, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, ISO 8807, ISO – Information Processing Systems – Open Systems Interconnection, 1989.

[4] ISO/IEC, Enhancements to LOTOS (E-LOTOS), ISO/IEC 15437, ISO – Information Technology, 2001.

[5] H. Fecher, Event structures for interrupt process algebras, in: Proc. EXPRESS'03, Electr. Notes Theor. Comput. Sci. 96 (2004) 113–127.

[6] H. Fecher, M. E. Majster-Cederbaum, and J. Wu, Refinement of actions in a real-time process algebra with a true concurrency model, in: Proc. REFINE'02, Electr. Notes Theor. Comput. Sci. 70(3) (2002).

[7] R. van Glabbeek and G. Plotkin, Event structures for resolvable conflict, in: Proc. MFCS'04, Lecture Notes in Computer Science, vol. 3153 (Springer, Berlin, 2004) 550-561.

[8] M. Kapus-Kolar, A generalization of the E-LOTOS renaming operator: a convenience for specification of new forms of process composition, Computer Standards & Interfaces 26(6) (2004) 549–563.

[9] J.-P. Katoen, D.Latella, R. Langerak, and E. Brinksma, On specifying real-time systems in a causality-based setting, in: Proc. FTRTFT'96, Lecture Notes in Computer Science, vol. 1135 (Springer, Berlin, 1996) 385–405.

[10] L. Lamport, Processes are in the eye of the beholder, Theoretical Computer Science 179(1–2) (1997) 333–351.

[11] R. Langerak, Bundle event structures: a non-interleaving semantics for LOTOS, in: Proc. FORTE'92 (North-Holland, Amsterdam, 1993) 331–346.

[12] L. Léonard and G. Leduc, An introduction to ET-LOTOS for the description of time-sensitive systems, Computer Networks & ISDN Systems 29(3) (1997) 271–292.

[13] D. Mizuguchi and K. Yamaguchi, Causal event structures with complete transformation rules, IPSJ Journal 44(1) (2003) 15–28.

[14] A. Nakata, T. Higashino, and K. Taniguchi, Protocol synthesis from context-free processes using event structures, in: Proc. RTCSA'98 (IEEE Computer Society Press, 1998) 173-180.

[15] A. Verdejo, E-LOTOS: Tutorial and semantics, M.S. Thesis, Universidad Complutense de Madrid, 1999.

[16] G. Winskel, An introduction to event structures, in: Proc. REX'89, Lecture Notes in Computer Science, vol. 354 (Springer, Berlin, 1989) 364–397.

Monika Kapus-Kolar received the B.S. degree in electrical engineering from the University of Maribor, Slovenia, and the M.S. and Ph.D. degrees in computer science from the University of Ljubljana, Slovenia. Since 1981 she has been with the Jožef Stefan Institute, Ljubljana, where she is currently a researcher at the Department of Communication Systems. Her current research interests include formal specification techniques and methods for the development of distributed systems and computer networks.