An action refinement operator for E-LOTOS with true concurrency

M. Kapus-Kolar [1]

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

Received 31 January 2007

Abstract

Action refinement is an important operation in the hierarchical synthesis of concurrent systems. We propose an action refinement operator for E-LOTOS, a standard process-algebraic language for formal specification of real-time, concurrent and reactive systems. As the first step towards giving E-LOTOS a multitude of refinement operators supporting a variety of strategies for event relationship inheritance, we propose an operator which seems both useful and simple to implement. When the operator is applied to an E-LOTOS process, it modifies its enhanced event structure, i.e. its recently defined true concurrency model. The operator allows multiple alternative implementations even for urgent events and properly reflects the fact that gate actions of E-LOTOS processes are in the general case abstractions of distributed data generation procedures.

Key words: Process algebra, Action refinement, True concurrency, E-LOTOS

1. Introduction

One of the typical steps in the synthesis of a system is refinement of an action a into a multi-action process $\rho(a)$, where the fine-grain actions selectively inherit the attributes of a, including its relationships with the other actions of the system. If a is explicitly specified and the inheritance relation is implicitly known, the action refinement can be *syntactic*, i.e., in the formal system specification, one simply replaces a with $\rho(a)$ enclosed into brackets indicating that the process is a refinement of a coarse-grain action. It is then a matter of the formal semantics of the specification language to properly relate the constituent actions of $\rho(a)$ to the other actions. Such a refined system specification is very readable, as one can clearly see the hierarchy of actions.

It is, however, quite possible that a is not specified explicitly. For example, if it is an interaction between two processes of a system, the readiness of the processes to execute a is explicitly specified, while a itself is not. In such a case, only *semantic* refinement of a is possible, i.e., one takes the coarse-grained specification of the system and declares that whenever an action of kind a is to occur, it must be executed as $\rho(a)$. To facilitate such declarations, specification languages are increasingly being furnished with *action refinement operators*.

Ideally, a language would offer a rich choice of action refinement operators (or, equivalently, a single, but richly parameterized one), to support a wide range of different attribute inheritance strategies. At present, however, it is not unusual for a specification language to have no action refinement operator at all. The reason is probably that in the past, languages were not routinely furnished with a semantic model explicitly representing action relationships. Besides, even if a language does possess such a model, implementation of action relationship inheritance is not simple.

For a given specification language, it is, hence, reasonable to start with the modest goal of defining a single action refinement operator, implementing just a single, very specific inheritance strategy, preferably one well reflecting the underlying philosophy of the language. In this paper, we propose such an action refinement operator for E-LOTOS [4,14] (an enhanced successor of LOTOS [3,1]), one of the standard languages for formal specification of real-time, concurrent and reactive systems.

There have been earlier attempts to define action refinement for LOTOS and similar languages. Let us mention just [2,13], as two of the earliest, and then [11], whose method seems to be the most accomplished one among the currently available for timed LOTOS, for it supports even refinement of visible urgent actions. However, none of the existing methods is directly applicable to E-LOTOS, which is a much richer language and

[1] Tel.: +386 14773531; fax: +386 14773111.
E-mail address: monika.kapus-kolar@ijs.si.

therefore calls for an action refinement strategy of its own. Besides, our work should be interesting also as the first attempt to benefit from the recently proposed true concurrency semantics of the language [10].

The semantic model proposed for E-LOTOS processes in [10] is an instance of *enhanced event structures* (EESs) [10], where events are the atomic and instantaneous elementary constituents of process behaviour, i.e. the entities whose refinement we are trying to define. Like in [10], we focus entirely on the semantic aspects of the problem, neglecting the question on how to efficiently handle large or even infinite and/or recursively specified EESs. Such approach is not naive, as one can always furnish a specification language with additional restrictions banning specifications which are not sufficiently tractable.

The paper is organized as follows: Section 2 describes the external observers' perspective on E-LOTOS processes, i.e. the perspective from which action refinement is supposed to be specified. Section 3 describes the internal perspective on E-LOTOS processes, i.e., represents the processes as machines implementing the external behaviour which they exhibit. Let us note that we describe the E-LOTOS syntax and semantics just to the detail necessary for understanding the issues of action refinement. For more detailed information, please, refer to [4,10]. In Section 4, we conceptually and syntactically conceive an action refinement operator for E-LOTOS. In Section 5, we give its precise semantics, by defining how it effects the EES of a process. Section 6 comprises a discussion and conclusions.

Before we proceed, let us for motivation state those features of the operator proposed below which we find most convenient: It allows multiple alternative implementations even for urgent events and properly reflects the fact that gate actions of E-LOTOS processes are in the general case abstractions of distributed data generation procedures.

2. The external perspective on E-LOTOS processes

In the following, let B denote the E-LOTOS process whose actions are being refined, and B' the process resulting from the refinement. According to the philosophy which the language has inherited from LOTOS, the only relevant property of B is its behaviour, defined as its readiness to engage into various kinds of events. More precisely, it is only the external perspective on the behaviour which really matters. From the perspective, B engages into four kinds of events:

- It might execute a gate action, i.e., interact with its environment on a gate. The environment perceives the event as a $G(Data)$, where G is the gate and $Data$ is the data carried in the interaction. $G(Data)$ is, hence, the external name of the event. Gate actions are in E-LOTOS the only events which are non-urgent.
- It might execute an unobservable action. From the external perspective, such actions are anonymous and, hence, all given the same special dummy external name "**i**()".
- It might indicate successful termination. The external name of such an event is a $\delta(Data)$, where δ denotes successful termination and $Data$ for each variable which B might bind on its way towards successful termination, i.e. for each variable in Bnd_B, provides information on its value upon the termination.
- It might signal an exception. The external name of such an event is an $X(Data)$, with X denoting the general kind of the exception and $Data$ providing additional details.

From the external perspective, all events of B occur in the time defined by a global clock starting upon the start of B. It is important to be aware that two or more events might occur simultaneously. We adopt the default version of the true concurrency semantics from [10], according to which simultaneity is in principle allowed even for events with the same external name.

For every process B, [10] defines a dynamic attribute Trm_B. When B is ready for an immediate $\delta(Data)$ or $X(Data)$, the value of Trm_B is the event name. At any other time, the value of Trm_B is "**none**".

3. The internal perspective on E-LOTOS processes

While it is appropriate that action refinement is specified from the external perspective, its implementation requires changes in the disposition of B to act. To be able to define the changes, we need an internal perspective on B. This perspective is defined by its EES $\mathcal{E}$ [10].

From the internal perspective, B is basically a static collection of elementary events e and their relationships. Its dynamics arises from the fact that each of its events might at some point in time jump from its initial state "has not yet occurred" to its state "has already occurred".

An e can occur only when it is logically enabled, i.e. not suspended. This is when each of the preconditions ξ of e whose trigger $\tau(e,\xi)$ is currently true is satisfied. If a $\tau(e,\xi)$ is false just before the occurrence of e, it must be false also just after it. Besides, just after the occurrence of e, each of the postconditions λ of e whose trigger $\tau(e,\lambda)$ was true just before e must be satisfied. In the default E-LOTOS semantics proposed in [10], we give every e exactly one postcondition: its permanently active naming and timing constraint $\lambda(e)$.

The passing of time is reflected in B as aging of its events. An age is a value of type "time", which in all our examples corresponds to non-negative rationals. Initially, the age $a(e)$ of an e is 0, but increases whenever e idles, i.e. postpones its occurrence in spite of being logically enabled. When an e eventually occurs, $a(e)$ stops changing and indicates the relative execution time (RET) of e.

If a logically enabled e is currently urgent, it in principle occurs immediately, i.e. before the global time makes further progress, unless it is immediately suspended by the occurrence of some other events. Nevertheless, there might still exist some constraints rendering immediate occurrence of e impossible. In such a case, if e is not immediately suspended, B is blocked for ever. The urgency of an e is regulated by a $\phi(e)$ and a $\varphi(e)$. An e is urgent when $\phi(e)$ is true or if it can immediate occur in a manner satisfying $\varphi(e)$ as a postcondition.

When an e occurs, a specific name $n(e,c)$ is generated for it for every context c in which it is embedded. For example, every subprocess of B to which e belongs, including the smallest such subprocess B_e, is typically declared as a specific context for it. In particular, every E-LOTOS renaming or hiding operator acting on e within B introduces an additional name for it. The most important context for an e is its home context c_e, i.e. the context of B_e. One of the names generated for an e is designated as its external name $n(e)$, while $n(e,c_e)$ is the basic name of e.

An $n(e,c)$ is in general a hierarchical list of fields. Within a list, the default key for its k-th field is "$\$k$". Take, for example, a name n of the form "$\delta(V_1 \Rightarrow Val_1, V_2 \Rightarrow Val_2)$". We interpret it as a list of two elements. The first element has value δ and default identifier $n.\$1$. The second element is a list with elements Val_1 and Val_2. The default identifier for Val_1 is $n.\$2.\$1$, but the element also has a special identifier $n.\$2.V_1$, thanks to the explicitly specified key V_1. For every event e, the information on $n(e,c_e).\$1$ available in $\lambda(e)$ qualifies e as an (observable or unobservable) action, as a successful termination or as an exception signalling. Accordingly, E, the event set of $\mathcal{E}$, is partitioned into sets A, Δ and Σ, respectively.

When an e occurs, the occurrence is a part of a synchronized occurrence s of elementary events in a (possibly singleton) set $E(s)$. Such an s denotes occurrence of the compound event which we shall call $\langle E(s)\rangle$. In an s, all e in $E(s)$ occur under the same external name, the external name $n(s)$ of s, which thereby becomes the external name $n(\langle E(s)\rangle)$ of $\langle E(s)\rangle$. Synchronizations s are what we call events in the external perspective, where $n(s)$ is the name under which they are perceived, if visible, by the environment of B, and referred to by operators on B.

When an s occurs, each e in $E(s)$ is a contribution of one of the elementary subprocesses of B executing s as a common act. In each of the concerned contexts c, s activates one of the local meeting points, in which the members of $E(s)$ belonging to c synchronize with respect to c. For a meeting point m, $R(m)$ denotes the set of the associated roles. For a role r, $E(r)$ denotes the set of the elementary events allowed to play the role. In each activated meeting point m, the synchronizing events play roles which are to the required degree complementary and adapt their external name to the local naming constraint $\mu(m)$. How exactly will elementary events group into synchronizations is in general not known in advance, because E-LOTOS, like LOTOS, supports flexible multiway synchronization. This is the main reason why syntactic action refinement is so very difficult to implement for LOTOS and its successors [13].

Alternatively, $\mathcal{E}$ can be interpreted as a collection of data objects and rules for their concurrent updating. The data objects are

- individual input data parameters of B, with value fixed before the start of B,
- individual elementary events e, interpreted as Boolean variables always indicating whether e has already occurred,
- individual event names $n(e,c)$, with value officially set upon the occurrence of e, though often known already before it, and
- individual event ages $a(e)$, with value always representing the amount of time which e has spent in the state of idling.

We define that a data object is an input parameter of an e if $\lambda(e)$ refers to it, but it is not e, $a(e)$ or an $n(e,c)$. The E-LOTOS semantics of [10] secures that when an e is logically enabled for the first time, each of its input parameters already has an official and stable value.

The defaults for a $\mu(m)$, a $\phi(e)$, a $\varphi(e)$, a $\tau(e,\xi)$ or a $\tau(e,\lambda)$ are "true", "false", "false", "true" and "true", respectively.

4. Conception of an action refinement operator

4.1. *Assessing the needs*

The needs which an E-LOTOS action refinement operator should cover will be discussed from the external perspective. We are definitely interested in the refinement of gate actions of B. We are not interested in the refinement of its unobservable actions, since they are anonymous. For the successful termination of B (if any), E-LOTOS already provides a refinement operator, namely, the sequential composition operator [4], which "traps" the event and starts a process acting as its handler instead. E-LOTOS also facilitates trapping of exception signals, but as such trapping permanently reverts control from B, the operator specifying it qualifies as a useful refinement operator only for exceptions denoting unsuccessful termination of B. We, hence, need an operator facilitating refinement of gate actions and exception signallings.

4.2. *The internal perspective on action refinement*

By an action refinement operator, one specifies a collection of rules on how B' should implement individual s from B. By such a rule, one declares that whenever B would execute an s with $n(s)$ of a specific kind, B' should not execute s as it is, while executing it as a specific B'' is acceptable. From the internal perspective, the decla-

ration requires refinement of every $\langle E' \rangle$ with E' able to act as $E(s)$ of an s with $n(s)$ of the specified kind.

4.3. *Alternative refinements*

For an $\langle E' \rangle$ in B, it might be impossible to tell in advance what $n(\langle E' \rangle)$ will be. Hence, even if one strictly refrains from associating a name n with multiple alternative refinements for s with $n(s) = n$, it is in general not possible to tell in advance precisely how an $\langle E' \rangle$ from B will be executed by B'. Obviously, we will have to support multiple alternative executions per an $\langle E' \rangle$. Once we master this problem, it should not be too difficult to also master multiple alternative refinements per an s. If $E' \subseteq \Sigma$, handling of alternative executions of $\langle E' \rangle$ is simplified by the fact that E' is always an $\{e\}$ with e never logically enabled without $n(e)$, and thereby $n(\langle E' \rangle)$, being already known precisely.

4.4. *Abstracting processes into events*

When B' executes an $\langle E' \rangle$ with $E' \subseteq A$ from B as a B'', there is often more than one acceptable way of abstracting B'' into $n(\langle E' \rangle)$ and more than one acceptable mapping of $n(\langle E' \rangle)$ into the basic names $n(e, c_e)$ of e in E'. While $n(\langle E' \rangle)$ plays in B' only an auxiliary role, the associated $n(e, c_e)$ are extremely important, because they define how the output data of B'' affect further execution of B'. Hence, B' must in general immediately upon successful termination of B'', i.e. in the same time instant, perform an auxiliary internal action by which it makes a consistent choice of the names. The action will be called $end(B'')$.

In general, the details of the s into which B'' abstracts might not be fully available until $end(B'')$ occurs. It is, hence, appropriate to define that $\langle E' \rangle$ abstractly occurs upon the action. The arrangement corresponds to the very common view that an abstract event is complete when its implementation is. The occurrence moment of $\langle E' \rangle$ defines the RET of individual e in E'.

If $E' \subseteq \Sigma$, E' is an $\{e\}$ with $n(e, c_e)$ not affecting further execution of B' and with the RET of e zero by definition, since exception signallings are urgent. Hence, the above reasons for giving B'' an $end(B'')$ no longer apply, but we still need it as an action representing the abstract occurrence of $\langle E' \rangle$ (see Sections 4.8 and 4.9). The action is usually interpreted as the $\delta(\ldots)$ event of B'' renamed into **i**() [6]. Such interpretation is very convenient, since it explains the role of the $\delta(\ldots)$ event within B'.

Example 1 *Let B be a "***rename action** $G(?V : \text{int})$ **is** $G'!\text{abs}(V)$ **in** $G?V_1 : \text{int} || G?V_2 : \text{int}$ **endren***". "$||$" is the operator of concurrency requiring synchronization on every gate action. Hence, the two concurrent processes "$G?V_1 : \text{int}$" and "$G?V_2 : \text{int}$" execute their only gate action in cooperation, as a $G(Val)$, where Val is the integer value thereby assigned to the variables V_1 and V_2. From the internal perspective, the two subprocesses execute an e_1 and an e_2, respectively, where the gate action corresponds to $\langle \{e_1, e_2\} \rangle$. The environment of B perceives the gate action as a $G'(|Val|)$, and the implicitly specified successful termination event of B as $\delta(V_1 \Rightarrow Val, V_2 \Rightarrow Val)$.*

Suppose that $\langle \{e_1, e_2\} \rangle$ is executed as a B'' instead and that the output data of B'' imply that the acceptable values for $n(\langle \{e_1, e_2\} \rangle)$, i.e. the acceptable abstractions of B'', are $G'(1)$ and $G'(2)$. Hence, it is upon the successful termination of B'' not evident whether $n(e_1, c_{e_1})$ and $n(e_2, c_{e_2})$ should be set to $G(1)$, to $G(-1)$, to $G(2)$ or to $G(-2)$. This is critical, because the two names, respectively, define the value indicated for V_1 and V_2 in the $\delta(\ldots)$ event of B, i.e. the $\delta(\ldots)$ event of B'. The two concurrent processes in B cannot secure the required $V_1 = V_2$ without the specified synchronization on gate G, but within B', the synchronization is not explicit. Hence, for consistently resolving the choice, we need an auxiliary event.

4.5. *Starting of event implementations*

When a B'' is started as a potential implementation of an $\langle E' \rangle$, its starting time and input data must secure that in the case that B'' successfully terminates, it will be possible to give its abstraction $\langle E' \rangle$ an external name consistent with its occurrence moment and the output data of B''. The choice between multiple acceptable starting times is nondeterministic.

The data available to a B'' upon its start are primarily the input parameters of B'. If a B'' is a potential implementation of an $\langle E' \rangle$ with $E' \subseteq \Sigma$ and is by definition or specification not allowed to start until the only e in E' is logically enabled, the data carried in the signal may also be among the input parameters of B'', for remember that $n(\langle E' \rangle)$ is already known upon the enabling of e.

Finding an appropriate starting time for an implementation B'' of an $\langle E' \rangle$ might be extremely difficult, because the successful termination time of B'' must satisfy the requirements of each individual e in E'. If a timed e in E' is ever suspended by another event over a finite non-zero time interval during which a timed e' in E' is not, the two members of E' virtually lack a common notion of time, so that satisfying them both might even be impossible. Therefore, any such $\langle E' \rangle$ will be considered unrefinable, i.e., B' will execute it as it is even if a refinement has accidentally been specified for it. A formal characterization of the events which we consider unrefinable is given in Section 5.1.

Example 2 *Let B be a "$B_1 | [G] | B_2$". "$|[\ldots]|$" is the operator of concurrency requiring synchronization on every action on the listed gates (in our case, on gate G). Let each process B_i be a "$B_{i,1}[X_i > B_{i,2}$". The suspend/resume operator "$[X_i >$" specifies that B_i basically executes process $B_{i,1}$, but repeatedly suspends it by consecutive instances of process $B_{i,2}$, where such a suspension starts upon the first event of the current instance of $B_{i,2}$ and ends when the suspender enables exception X_i,*

*thereby transferring control to the next instance of $B_{i,2}$. Let each $B_{i,1}$ be a "$G@?V_i[i+2<V_i<i+5]$", i.e. ready just for an action on gate G, with RET between $(i+2)$ and $(i+5)$. Let each $B_{i,2}$ be a "G_i; **wait**(1); **signal** X_i", i.e. ready just for an action on gate G_i, with any RET, followed by $X_i()$ after one time unit.*

If there were no $B_{1,2}$ and $B_{2,2}$, an appropriate delay for action $G()$ in B would be a t between 4 and 6. Replacing $G()$ with a process B'' terminating in exactly 2 time units would, hence, require that B'' starts at a t' between 2 and 4. However, as it is, each B_i independently and unpredictably suspends its timing of action $G()$, implying that it is impossible to choose for B'' a starting time guaranteed to satisfy both B_1 and B_2.

Now let B be a "$(B_{1,1}|[G]|B_{2,1})[X > (G'; \mathbf{wait}(1); \mathbf{signal}\ X)$" instead, with $B_{1,1}$ and $B_{2,1}$ as above. Note that the described problem no longer exists, because the synchronizing elementary events e_i in $B_{i,1}$ are never suspended individually. One would, hence, start B'' when $a(e_1)$ and $a(e_2)$ are between 2 and 4, and then take care that it is suspended whenever e_1 and e_2 are.

4.6. *Inheritance of causality*

If the duration (i.e. the time from the start to the successful termination, if any) of a B'' specified as a potential implementation of an $\langle E' \rangle$ is very long, B'' cannot terminate in time if its start is delayed until $\langle E' \rangle$ is logically enabled in B. It might, hence, be desirable that events in B'' overtake events guarding $\langle E' \rangle$, although not those on which they depend. This would mean that for a subprocess B_1 of B specified as strictly preceding $\langle E' \rangle$, B_1 and B'' would in B' be in weak sequential composition parameterized with action dependencies.

There indeed exist process algebras supporting such action refinement [12]. Still, we stick to the more usual approach that B'' is allowed to run only when $\langle E' \rangle$, i.e. every e in E', is logically enabled. The reason is that although weak sequencing has been defined for E-LOTOS [9], it has not yet been included into the new semantics proposed in [10].

4.7. *Prevention of time nondeterminism*

If a B'' running as a potential implementation of an $\langle E' \rangle$ in B enables an exception signalling concurrently to an event in B' outside B'', trapping of the signal of B' potentially leads to time nondeterminism, which is in E-LOTOS a taboo. For prevention of such situations, we follow the approach widely employed already in the standard E-LOTOS semantics [4]: We define that any exception signalling from B'' is in B' preceded with an auxiliary internal action explicitly representing the decision that the signalling should occur. Such a prefix suspends all other events in B' until the signalling actually occurs, so that there is no event concurrent to the signalling.

4.8. *Commitment of event implementations*

For a B'' running as a potential implementation of an $\langle E' \rangle$, one has to define the moment of its commitment, i.e. the moment when it becomes the selected means for the (abstract) execution of e in E'. On the one hand, the commitment should be as late as possible, so that in the case that B'' fails to successfully terminate, e may still be executed in some other way (by execution of $\langle E' \rangle$ as it is, by execution of an alternative refinement of $\langle E' \rangle$ or by (possibly abstract) execution of an $\langle E'' \rangle$). On the other hand, if B'' is to disrupt a rival, this should preferably happen as soon as possible, to prevent redundant events in the rival. Unfortunately, it is usually not possible to tell in advance whether B'' will run to completion.

A compromise would be to delay the choice between competing event implementations until their external behaviours begin to differ, i.e., to let the processes run concurrently, with visible events, including exception signallings, executed strictly as shared events, where any failure of a process to cooperate with the others would mean that the process is cancelled, while the others continue to compete. Unfortunately, such gradual choice upon consecutive observational divergences cannot be easily formalized in an event structure model.

For the above reasons, we suggest that it should be possible to specify whether B'' should commit (i) upon successful termination (late commitment), i.e., when B' executes $end(B'')$, or (ii) when it begins exhibiting visible behaviour, i.e., when B' executes a gate action in B'', the prefix of an exception signalling in B'' or $end(B'')$ (early commitment). An $\langle E' \rangle$ executed as it is commits when it occurs.

4.9. *Inheritance of conflicts*

We have already decided that a B'' started as a potential implementation of an $\langle E' \rangle$ should be suspended whenever $\langle E' \rangle$, i.e. some e in E', is. A more difficult task is to decide which event in B'' should be the one suspending the implementations of the events which are in B suspended by $\langle E' \rangle$. Should it be, for example, its start, its first event, its commitment event or $end(B'')$?

One of the specialties of E-LOTOS is its suspend/resume operator. If in the current instance B_2' of B_2 in a $B_1[X > B_2$, $\langle E' \rangle$ is the first event, i.e. responsible for suspending B_1, it might be important that the time between the last resumption of B_1 and the considered suspension is exactly the time between the start of B_2' and the completion of $\langle E' \rangle$, as originally specified. It is, hence, appropriate to define that the suspensive event of B'' is $end(B'')$, because it occurs exactly when $\langle E' \rangle$ would. End-based suspension, at least permanent suspension of rival alternatives, is not uncommon in action refinement [6].

Now suppose that in a $B_1[X > B_2$, $\langle E' \rangle$ is not in B_2, but in B_1, and that B_2 permanently suspends B_1 when

B'' has already executed some, but not all of its events, in particular not $end(B'')$. The already executed events of B'' are redundant. If some of them have been visible to the environment of B', this is inconvenient, but expected. On the other hand, if an event in B'' has already suspended an event in B' outside B'', this is in principle an undesirable side effect, even though not uncommon if prevention of redundant events is also pursued. This further supports our belief that the default form of suspension in E-LOTOS should be end-based suspension, for $end(B'')$ is the only event in B'' which can never be redundant, and can, hence, always safely act as a suspender.

4.10. *Inheritance of urgency*

Let B'' be an acceptable implementation of an $\langle E' \rangle$, meaning that an acceptable starting time can be found for it. Is B' obliged to start B'' (and also every other acceptable implementation of $\langle E' \rangle$, because until one of them commits, they are by definition on an equal footing)? For an urgent $\langle E' \rangle$, the answer is, of course, positive.

For a non-urgent $\langle E' \rangle$, the current opinion seems to be that B' has no such obligation for individual runs, though it must have runs in which B'' is actually started [11]. We observe that without the obligation, B' might unnecessarily refuse execution of the event in B. In E-LOTOS, such arbitrariness of B' could be acceptable for an internal action of B (as it is not directly visible to the environment) or for an exception signalling (as the environment is just its passive observer [7]), if those events were not urgent. For a non-urgent $\langle E' \rangle$, however, we find it unacceptable, because such an event is a synchronous interaction between B and its environment, which should in principle decide on the failure of $\langle E' \rangle$ in cooperation, *during* its execution. For an $\langle E' \rangle$ refined into a B'', this means during the execution of B'', which should, hence, not fail to start.

For a B'' implementing an urgent $\langle E' \rangle$, a proper start is not all that is reasonable to require. We also declare that it is the responsibility of the specifier to secure that B'', if not abandoned, timely successfully terminates or at least indicates its inability to do.

4.11. *Specifying event refinement*

After the above detailed discussion of event refinement from the internal perspective, we are ready to return to the external perspective and conceive the syntax of the operator. We propose that B' is specified as a "**refine** $Ref_1 \ldots Ref_k$ **in** B **endref**". Ref_i is a "**signal** $X_i(IPL_i)[Cnst_i]$ **is** B_i $Comm_i$" or an "**action** $G_i Ptr_i[Cnst_i]@?V_i[Cnst'_i]$ **is** B_i $Comm_i$", with "(IPL_i)", "$[Cnst_i]$", "$Comm_i$", "Ptr_i" and "$[Cnst'_i]$" optional. IPL_i is an input parameter list. $Cnst_i$ and $Cnst'_i$ are constraints. Ptr_i is a pattern. V_i is a variable of type "time". "$Comm_i$" specifies how event implementation B_i should commit and is, hence, "**early**" or "**late**".

With the operator, we mimic our flexible event renaming operator from [8], except that we do not forbid multiple alternative implementations per exception signalling, while multiple alternative renamings per exception signal are forbidden. Limiting our attention to refinable events, we define that events with a refinement specified cannot be executed in their original form, while events with no refinement specified are executed as they are. In the following, the constraint which an $n(s)$ must satisfy if a B_i specified in a Ref_i is to qualify as an implementation of s will be called $Cnst_i^+$. For example, the name **i**() which hidden actions have by definition can never satisfy a $Cnst_i^+$.

A Ref_i of the form "**signal** $X_i(IPL_i)[Cnst_i]$ **is** B_i $Comm_i$" specifies that B_i qualifies as an implementation of any exception signalling s with external name $n(s)$ an $X_i(Data)$ with $Data$ matching IPL_i in a manner satisfying $Cnst_i$. Through IPL_i, the data in $Data$ are selectively passed to B_i. Any input parameter of IPL_i is an input parameter of B' and so is any input parameter of $Cnst_i$ or B_i which is not an output parameter of IPL_i.

The default IPL_i is empty. The default $Cnst_i$ is "true". The default $Comm_i$ is "**early**". As exception signalling is urgent and B_i is not allowed to start before the event which it refines is logically enabled, it is the responsibility of the specifier to secure that B_i for every input data for which it is intended, in every possible run terminates in no time, either successfully or by raising an exception indicating a failure.

Example 3 *Let B' be a "***refine signal** $X(?V_1 : \text{int})[0 < V_1 \leq V]$ **is** B_1 **signal** $X(?V_2 : \text{int})[V_2 \geq V]$ **is** B_2 **in** B **endref***", where each process B_i is a "***signal** $X_i(V_i)$; **signal** $X'_i(\text{abs}(V_i - V))$*". All V in Ref_i represent an input parameter of B'. A B_i imports not only V, but also, under name V_i, the integer carried by the exception which it refines.*

Suppose that V is 2. An $X(0)$ in B would be executed as it is. An $X(1)$ would be executed as event sequence "$X_1(1), X'_1(1)$". An $X(2)$ would be executed as "$X_1(2), X'_1(0)$" or as "$X_2(2), X'_2(0)$". An $X(3)$ would be executed as "$X_2(3), X'_2(1)$".

A Ref_i of the form "**action** $G_i Ptr_i[Cnst_i]@?V_i[Cnst'_i]$ **is** B_i $Comm_i$" specifies that B_i qualifies as an implementation of any gate action s with external name $n(s)$ a $G_i(Data)$ with $Cnst'_i$ indicating the belief that B_i might successfully terminate with results justifying abstraction of the particular instance of B_i into $G_i(Data)$, i.e., with results on which Ptr_i can evaluate to $Data$ in a manner satisfying $Cnst_i$. The results of B_i are its output values for the variables in Bnd_{B_i}, provided in its $\delta(\ldots)$ event, and its duration V_i. A combination of the results is believed to be possible exactly if it satisfies $Cnst'_i$. Any input parameter of B_i is an input parameter of B' and so is any input parameter of Ptr_i or $Cnst'_i$ which is not in $Bnd_{B_i} \cup \{V_i\}$, i.e. imported from B_i, and

every input parameter of $Cnst_i$ which is not an output parameter of Ptr_i or in $Bnd_{B_i} \cup \{V_i\}$.

The default Ptr_i is empty. The default $Cnst_i$ is "true". The default $Comm_i$ is "**early**". The default $Cnst'_i$ is $Cnst^*_i$, the precise characterization of the possible results of B_i. However, B_i might be too complicated for automatic construction of $Cnst^*_i$, and this is why we allow explicit specification of $Cnst'_i$. A specifier is not obliged to make $Cnst'_i$ precise, but must secure that for every input data for which B_i is intended, every combination of results which B_i might produce indeed satisfies $Cnst'_i$. The only knowledge on the results of B_i present in an explicit $Cnst'_i$ by default is that of the declared type of the output data and of the default type of the duration of B_i, including the knowledge that the duration is non-negative.

Example 4 *Let B' be a "***refine action** $G(!(V+1), ?V_2 : \text{rational})[V_2 > V_1]@?V_1[Cnst]$ **is** B_1 **in** B **endref***", where B_1 is a "$G_1?V@!1[V < 4][]G_2!V@!2[V < 3]$", with "$[]$" the operator of choice, the constraint $Cnst$ is "$(1 \leq V_1 \leq 2) \wedge (V + V_1 < 5)$", and the input parameter V of B' is a rational. V is an input parameter of B_1, while Ptr_1 and $Cnst$ import V from B_1. Note that B_1 internally uses V as an ordinary variable and might even produce a new value for it.*

If an instance B'' of B_1 is executed instead of a $G(Data)$ in B, the value imported for V by B'' is a Val valid upon the start of B'. In the first alternative, B'' normally executes a gate action $G_1(Val')$ with RET 1 and $Val' < 4$ a new value for V, and then indicates its successful termination and output data by a $\delta(V \Rightarrow Val')$. In the second alternative, B'' normally executes, provided that $Val < 3$, a gate action $G_2(Val)$ with RET 2, and then a $\delta(V \Rightarrow Val)$. The duration V_1 of B'' is 1 or 2, respectively. We see that Bnd_{B_1} is $\{V\}$. With $((V_1 = 1) \wedge (V < 4))$ and $((V_1 = 2) \wedge (V < 3))$, respectively, the two alternatives of B'' indeed satisfy $Cnst$ if they successfully terminate. With gate actions non-urgent, B'' might as well idle for ever, but this is not a problem, because B_1 is not intended for refinement of urgent events.

The value of V in "$G(!(V+1), ?V_2 : \text{rational})[V_2 > V_1]$" is as provided in the $\delta(\ldots)$ event of B'', i.e. Val' or Val, while V_1 is 1 or 2, respectively. B_1 qualifies as an implementation of any gate action $G(Val_1, Val_2)$ with Val_1 and Val_2 rationals for which it is possible to find such a rational V and a non-negative rational V_1 satisfying $Cnst$ that $Val_1 = (V+1)$ and $Val_2 > V_1$. For example, a $G(Val_1, 3)$ refines into B_1 if $Val_1 < 5$.

Suppose that B is a "$G(?V, !3)@?V'[V' < 6]$", i.e., ready to execute a $G(Val_1, 3)$ with RET a $Val_2 < 6$, and then a $\delta(V \Rightarrow Val_1, V' \Rightarrow Val_2)$. As Val_1 might be a rational less than 5, it is appropriate that B' starts as a possible implementation of the gate action an instance B'' of B_1, at any time less than 4. B' should also be ready to execute the gate action as it is, but only with $Val_1 \geq 5$ and before B'' commits. If the action occurs as it is, it immediately disables B''.

Now suppose that B is a "$G(?V, !3)[Cnst']$" with $Cnst'$ "$(V < 3) \vee (V > 6)$". As in its gate action $G(Val_1, 3)$, Val_1 might be a rational less than 5, so that the external name of the action satisfies $Cnst^+_1$, B' should in principle start an instance B'' of B_1, but does not, because there is no starting time for which $Cnst$ would imply that it will actually be possible to abstract B'' into a $G(Val_1, 3)$ with $((Val_1 < 5) \wedge Cnst')$, i.e. with $Val_1 < 3$. B' may still execute the gate action as it is, but only with $((Val_1 \geq 5) \wedge Cnst')$, i.e. with $Val_1 > 6$.

5. The refinement operator as an operator on EESs of E-LOTOS processes

In the following, we assume that B' is specified as a "**refine** $Ref_1 \ldots Ref_k$ **in** B **endref**" with every Ref_i introducing a process B_i as described in Section 4.11. As we will work with multiple processes, identifiers of EESs and their attributes will be, where necessary to avoid ambiguity, subscribed with the name of the modelled process.

In this section, we define how $\mathcal{E}_B$ and $\mathcal{E}_{B_i}$ combine into $\mathcal{E}_{B'}$. The definition is formulated as a construction procedure, with Sections 5.2 to 5.8 describing consecutive construction steps.

5.1. *The refinement operator as a composition operator*

As B' combines processes B and B_i, it is obviously a process composition operator. However, B' does not combine B with processes B_i directly, as the latter are just templates for event implementations. It is more appropriate to say that B' is a composition of B and of instances $B_{i,E'}$ of B_i with $\langle E' \rangle$ a refinable event in B for which B_i qualifies as an implementation.

Constructing $\mathcal{E}_{B'}$, one, hence, has to introduce an instance $\mathcal{E}_{B_{i,E'}}$ of $\mathcal{E}_{B_i}$ for every $E' \subseteq E_B$ for which it is suspected that B has an event $\langle E' \rangle$ which B' could execute as an instance $B_{i,E'}$ of B_i. For a B_i, let Θ_i be the set of all $E' \subseteq E_B$ for which $\mathcal{E}_{B_{i,E'}}$ is introduced.

Inclusion of an E' into a Θ_i does not automatically imply that activation of $B_{i,E'}$ will actually be allowed. Hence, trying to make Θ_i construction reasonably easy, we make the conditions on $\mathcal{E}_B$ for including an E' into a Θ_i less restrictive than necessary for producing a minimal set: Among the names satisfying $Cnst^+_i$, there must be an n with the following property: There exists such a combination of values for the input parameters and ages of e in E' that with all preconditions of e in E' ignored, $\langle E' \rangle$ can immediately occur with n its external name. Besides, $\langle E' \rangle$ must not be unrefinable.

Trying to make identification of unrefinable events reasonably easy, we make the condition for including an $\langle E' \rangle$ among the unrefinable events less restrictive than necessary for producing a minimal set: An $\langle E' \rangle$ is considered unrefinable if an e in E' has a critical pair $(\xi, \tau(e, \xi))$ for which an e' in E' does not have a copy. A pair $(\xi, \tau(e, \xi))$

is critical if it denotes temporary suspension of e originating from a suspend/resume operator within B, for remember that such suspension might be problematic if it applies only to some of the members of E' (Section 4.5). It does not matter whether e has acquired the pair directly, by appearing in the left operand of such an operator (see [10], Section 3.23, Step 8) or by inheritance of the resolvable conflict (see Section 5.5). Such a pair is characterized by the property that neither ξ nor $\tau(e, \xi)$ is trivially true or false, but this is the case also for any pair denoting temporary suspension of e upon an exception signalling (see, for example, Section 5.8), where the latter kind of suspension is never problematic, as its duration is always zero. Hence, reliable identification of critical pairs is only possible during the construction of $\mathcal{E}_B$.

5.2. *Furnishing event implementations with an ending event*

When an $\langle E' \rangle$ with $E' \subseteq A_B$ is executed as a $B_{i,E'}$, it abstractly occurs upon $end(B_{i,E'})$, an event which we have in Section 4.4 explained as the $\delta(\ldots)$ event of $B_{i,E'}$ internalized and employed for special purposes. Unfortunately, in the internal perspective on B', the picture is not so clear. The $\langle E'' \rangle$ which $B_{i,E'}$ enables as its $\delta(\ldots)$ event is typically not known in advance and does not generate the basic names of e in E', as it should.

Therefore, we propose that $end(B_{i,E'})$ is rather added to $B_{i,E'}$ as a special event $e_{i,E'}$. As $\delta(\ldots)$ events of $B_{i,E'}$ consequently become useless, they must be blocked. The process obtained from $B_{i,E'}$ by the two modifications will be called $B'_{i,E'}$. We define it as an instance of the process "B_i; **i**; **stop**", slightly modified in a manner not influencing its dynamic attribute Trm ($Trm_{B'_{i,E'}}$ can, hence, still be computed as described in [10]). "B_i" specifies that the process first executes $B_{i,E'}$, up to, but not including, its $\delta(\ldots)$ event. "**i**" specifies the subsequent $end(B_{i,E'})$. Afterwards, the process just idles. Up to the additional modification, $\mathcal{E}_{B'_{i,E'}}$ can be constructed by enhancing $\mathcal{E}_{B_{i,E'}}$ as described in [10]. In the EES, one then identifies the $end(B_{i,E'})$ event $e_{i,E'}$ and furnishes it with an additional internal name, as follows:

The new name is an $n(e_{i,E'}, c_{i,E'})$ with $c_{i,E'}$ a new context comprising a meeting point m with $R(m)$ an $\{r\}$ with $e_{i,E'}$ the only member of $E(r)$. $\lambda(e_{i,E'})$ additionally requires that $n(e_{i,E'}, c_{i,E'})$ is finalized to an acceptable consistent selection of basic names for e in E', where the field which in the record $n(e_{i,E'}, c_{i,E'})$ contains the value assigned to an $n(e, c_e)$ is called $n(e_{i,E'}, c_{i,E'}).e$. More precisely, $\lambda(e_{i,E'})$ adapts $n(e_{i,E'}, c_{i,E'})$ to the current age of individual e in E', to their input data and to the output data of $B_{i,E'}$. The output data are available in the current value of $Trm_{B_{i,E'}}$, because upon the logical enabling of $e_{i,E'}$, $Trm_{B_{i,E'}}$ returns the external name of the blocked $\delta(\ldots)$ event of $B_{i,E'}$, in which the data are embedded.

Example 5 *Let B be a "***rename action** $G(\text{int})$ **is** $G'()$ **in** $G?V : \text{int}@?V'[V < V']||G?V'' : \text{int}[V'' < 3]$ **endren***". Let e_1 and e_2 denote the gate action of the first and of the second subprocess, respectively. The two constraints, respectively, define that $n(e_1, c_{e_1})$ can be any $G(Val)$ with Val an integer less than the RET of e_1, and $n(e_2, c_{e_2})$ can be any $G(Val)$ with Val an integer less than 3. In any case, the renaming sets $n(\langle\{e_1, e_2\}\rangle)$ to $G'()$.*

Suppose that $\langle\{e_1, e_2\}\rangle$ is executed as a $B_{1,\{e_1,e_2\}}$ and that the process becomes ready for a $\delta(\ldots)$ at a time which has not been precisely known in advance. As $\lambda(e_{1,\{e_1,e_2\}})$ has to secure that $n(e_{1,\{e_1,e_2\}}, c_{1,\{e_1,e_2\}})$ is an "$(e_1 \Rightarrow G(Val), e_2 \Rightarrow G(Val))$" with Val less than 3 and the RET of e_1, it has to refer to the current value of $a(e_1)$, for this is the RET of e_1. Note also that there are many basic names for e_1 and e_2 compatible with the external name $G'()$ of $\langle\{e_1, e_2\}\rangle$, and it is the task of $e_{1,\{e_1,e_2\}}$ to choose one of them.

If $E' \subseteq \Sigma$, $B'_{i,E'}$ can be simply an instance of the process "B_i; **i**; **stop**", because $e_{i,E'}$ does not have to act as the generator of the basic name of the only e in E'. Not only that the name is known in advance; it is also irrelevant for the future behaviour of B', as there is in E_B no e' for which the name would be an input parameter.

5.3. *Furnishing event implementations with a guard*

For a $B_{i,E'}$ with $E' \subseteq A_B$, it is often required that it does not start immediately upon the logical enabling of $\langle E' \rangle$, but with some delay. To represent the fact that B' chooses between the (possibly infinitely many) acceptable delays for $B_{i,E'}$ nondeterministically, one enhances $B'_{i,E'}$ into a $B''_{i,E'}$ specified as a "**var** $V_{i,E'}$: time **in hide** $G_{i,E'}$ **in** $G_{i,E'}?V_{i,E'}@!0[Cnst_{i,E'}]$ **endhide**; **wait**$(V_{i,E'})$ **endvar**; $B'_{i,E'}$", thereby enhancing $\mathcal{E}_{B'_{i,E'}}$ into $\mathcal{E}_{B''_{i,E'}}$ as described in [10].

In the expression for $B''_{i,E'}$, $V_{i,E'}$ denotes a special-purpose local variable of type "time". "**hide** $G_{i,E'}$ **in** $G_{i,E'}?V_{i,E'}@!0[Cnst_{i,E'}]$ **endhide**" makes $B''_{i,E'}$ execute an immediate hidden action setting $V_{i,E'}$ to a value satisfying constraint $Cnst_{i,E'}$, if any. "**wait**$(V_{i,E'})$" makes $B''_{i,E'}$ idle for the selected time $V_{i,E'}$, before it starts $B_{i,E'}$ by starting $B'_{i,E'}$. If no acceptable value for $V_{i,E'}$ exists, the presence of $B_{i,E'}$ in B' is irrelevant, for $B''_{i,E'}$ just idles, i.e., $B_{i,E'}$ is not activated.

Unlike $B_{i,E'}$, $B''_{i,E'}$ is supposed to start as soon as all e in E' are logically enabled simultaneously. After that, $B''_{i,E'}$ and e in E' are always suspended (and later possibly resumed) either simultaneously or not at all. Together with the fact that, by construction of the EES of B, no e in E' has a postcondition referring to an e' whose value has not been finalized already before the logical enabling of e, this implies that for the purpose of conceiving $Cnst_{i,E'}$, it is safe to pretend that $B''_{i,E'}$ and e in E' run in isolation.

For an e in E', let $a_{e,E'}$ denote its age $a(e)$ upon the first logical enabling of $\langle E' \rangle$, i.e. at the moment to which

$Cnst_{i,E'}$ refers. At that moment, all the input data of e in E' are also already available, because otherwise the events would not be logically enabled. This is all the information which $B''_{i,E'}$ might need for checking whether a specific external name would be acceptable for $\langle E' \rangle$ executed with a specific additional delay.

Remember that Ref_i is an "**action** $G_i Ptr_i[Cnst_i]$ @?$V_i[Cnst'_i]$ **is** B_i $Comm_i$". $Cnst_{i,E'}$ is a constraint restricting $V_{i,E'}$ exactly to values securing that for any combination of output data and duration V_i of $B_{i,E'}$ satisfying $Cnst'_i$, Ptr_i can evaluate to such $Data$ satisfying $Cnst_i$ that $G_i(Data)$ is in B an acceptable external name for $\langle E' \rangle$ executed at a time when the age of individual e in E' is $(a_{e,E'} + V_{i,E'} + V_i)$. Note that a more precise $Cnst'_i$ requires consideration of fewer combinations of the output data and the duration of $B_{i,E'}$, thereby increasing the chances of finding an acceptable $V_{i,E'}$. This might be extremely important, for example, if there are no alternative implementations specified for $\langle E' \rangle$.

Example 6 *Let B' be a "***refine action** G@?$V[V \leq 3]$ **is** B_1 **in** B **endref***" with B a "*$(?V_1 :=$ **any** time; G@?$V'_1[V'_1 < V_1])|[G]|($**wait**$(1); G$@?$V_2[V_2 < 5])$*". Let e_1 and e_2, respectively, denote the gate actions of the two subprocesses of B, and e' the immediate hidden event in "*$?V_1 :=$ **any** time*" which nondeterministically sets the value of V_1.*

$\langle \{e_1, e_2\} \rangle$ *is logically enabled at time 1, because of the delay introduced for e_2 by "***wait**(1)*". At that moment, V_1 is available as $n(e', c_{e'}).\$2.V_1$. An adequate $Cnst_{1,\{e_1,e_2\}}$ is "*$\forall V$: time.$((V \leq 3) \Rightarrow ((a(e_1) + V_{1,\{e_1,e_2\}} + V < n(e', c_{e'}).\$2.V_1) \wedge (a(e_2) + V_{1,\{e_1,e_2\}} + V < 5)))$*". Note that it links the starting event e'' of $\mathcal{E}_{B''_{1,\{e_1,e_2\}}}$ to e_1, e_2 and e' in $\mathcal{E}_B$. When $B''_{1,\{e_1,e_2\}}$ starts, $a(e_1)$ and $a(e_2)$ are 1 and 0, respectively, implying that e'' sets $V_{i,\{e_1,e_2\}}$ to a non-negative value less than $(n(e', c_{e'}).\$2.V_1 - 4)$ and 2, if any.*

If Ref_i is a "**signal** $X_i(IPL_i)[Cnst_i]$ **is** B_i $Comm_i$", $B_{i,E'}$ need not be delayed, but still has to be guarded. Hence, we again enhance $B'_{i,E'}$ into a $B''_{i,E'}$ (and $\mathcal{E}_{B'_{i,E'}}$ into $\mathcal{E}_{B''_{i,E'}}$), except that the process is now specified as an "**if** $Expr_{i,E'}$ **then** $Ptr_{i,E'} := Expr'_{i,E'}; B'_{i,E'}$ **else stop endif**". $B''_{i,E'}$ starts by evaluating Boolean expression $Expr_{i,E'}$ indicating whether the input data of $\langle E' \rangle$ secure that the only possible $n(\langle E' \rangle)$ satisfies $Cnst_i^+$. If the predicate is true, the assignment "$Ptr_{i,E'} := Expr'_{i,E'}$" prepares $Data$ for the subsequently executed $B_{i,E'}$. $Expr'_{i,E'}$ is an expression representing $Data$ as a function of the input data of $\langle E' \rangle$. $Ptr_{i,E'}$ to which the result of $Expr'_{i,E'}$ is matched is defined as a pattern securing that $Data$ is passed to $B_{i,E'}$ as specified by IPL_i.

Example 7 *Let B' be a "***refine signal** $X((?V$: int, $?V'$: bool$))[V > 2]$ **is** B_1 **in** B **endref***" with B a "*$(G_1 [] G_2 ?V_1$: int$)$; **signal** $X((V_1, \text{true}))$*". Let e_1, e_2 and e_3, respectively, denote the action on gate G_1, the action on gate G_2 and the exception signalling in B.*

*When e_3 is enabled, the current value of V_1 is either the one valid upon the start of B, in $\mathcal{E}_B$ called simply V_1 [10], or the one generated in e_2 and available as $n(e_2, c_{e_2}).\$2.V_1$. Consequently, e_1 and e_2 are also among the input data of e_3, because they tell where to read V_1. An adequate $B''_{1,\{e_3\}}$ is "***if** $(e_1 \wedge (V_1 > 2)) \vee (e_2 \wedge (n(e_2, c_{e_2}).\$2.V_1 > 2))$ **then** $(?V$: int, $?V'$: bool$) :=$ (**if** e_1 **then** V_1 **else** $n(e_2, c_{e_2}).\$2.V_1$ **endif**, true$); B'_{1,\{e_3\}}$ **else stop endif***".*

Finally, every e in an E' in a Θ_i also needs an additional constraint: $\lambda(e)$ must additionally forbid finalization of $n(e)$ to values satisfying $Cnst_i^+$.

5.4. *Furnishing exception signallings in event implementations with prefixes*

For the reasons described in Section 4.7, every e in a $\Sigma_{B''_{i,E'}}$ must get an auxiliary hidden prefix event e'. The prefixes enhance $B''_{i,E'}$ into a $B'''_{i,E'}$.

The home context $c_{e'}$ of such a prefix e' of an e must be a new context comprising a meeting point m with $R(m)$ an $\{r\}$ with e' the only member of $E(r)$. $\lambda(e')$ must require that $n(e', c_{e'})$ is finalized to **i**(). As e' must be urgent, $\varphi(e')$ is "true"

As e' may be logically enabled only when e would originally be, every precondition of e must be, together with its trigger, redirected to e'. As e' may occur only if e is not trapped in $B''_{i,E'}$, it must get an additional precondition true exactly when $\lambda(e)$ upon the logical enabling of e provides a value for $n(e)$, the external name of e (if in a specific run of $B''_{i,E'}$, $\lambda(e)$ fails to provide such a name, e is virtually non-existent, implying that its prefix must not occur). To secure that e occurs only after e', it must get a precondition "e'".

5.5. *Implementing inheritance of preconditions*

As a $B'''_{i,E'}$ must be enabled in B' exactly when $\langle E' \rangle$ is in B, one in every $E_{B'''_{i,E'}}$ gives every event a copy of every precondition and its trigger which an event in E' has.

5.6. *Refining references to refined events*

Wherever a precondition, a postcondition or a condition trigger belonging to an e in E_B or in an $E_{B'''_{i,E'}}$ refers to a data object belonging to another event e' in E_B, i.e. to e' itself, to $a(e')$ or to $n(e', c_{e'})$, the intention is not to influence, but to read the value of the object. If the reference is to e' or to $n(e', c_{e'})$, and e' belongs to an E'' for which a $B_{i',E''}$ exists, the reference must be refined too, as follows:

A reading of an e refines into a reading of the disjunction of e and every $e_{i,E'}$ with e in E'.

A reading of an $n(e, c_e)$ refines into a checking whether e itself or some $e_{i,E'}$ with e in E' is true, followed by a reading of $n(e, c_e)$ or $n(e_{i,E'}, c_{i,E'}).e$, respectively.

5.7. *Implementing commitment of event implementations*

For every $B_{i,E'}$, the members of E' are given a precondition "false" with trigger "$e_{i,E'}$", so that if $\langle E' \rangle$ is executed as $B_{i,E'}$, no $\langle E'' \rangle$ with non-empty $E' \cap E''$ can occur as it is.

Every $e_{i,E'}$ is given a precondition "false" whose trigger is true exactly if an e in E' is true, so that if an $\langle E'' \rangle$ with non-empty $E' \cap E''$ occurs as it is, $\langle E' \rangle$ cannot be executed as $B_{i,E'}$.

For every $B_{i,E'}$ and $B_{i',E''}$ with $(i,E') \neq (i',E'')$ and $E' \cap E''$ non-empty, $e_{i',E''}$ is given a precondition "false" with trigger "$e_{i,E'}$", so that if $\langle E' \rangle$ is executed as $B_{i,E'}$, $\langle E'' \rangle$ cannot be executed as $B_{i',E''}$.

For every $B'''_{i,E'}$ with $Comm_i$ "**early**", every e in E' or in an $E_{B'''_{i',E''}}$ with $(i,E') \neq (i',E'')$ and $E' \cap E''$ non-empty must get an additional precondition "false" whose trigger is true exactly if an e' in $A_{B''_{i,E'}}$ is true and $n(e') \neq \mathbf{i}()$, or if an e' in $(A_{B'''_{i,E'}} \setminus A_{B''_{i,E'}})$ is true, so that if a gate action or a prefix of an exception signalling in $B''_{i,E'}$ occurs, $B'''_{i,E'}$ becomes the only means for execution of e in E'.

5.8. *Implementing temporary suspension upon exception signallings*

The following applies to every exception signalling e in a $\Sigma_{B''_{i,E'}}$ and its prefix e' introduced as in Section 5.4: To secure that e' suspends B until e occurs, every e'' in E_B must get a precondition e with trigger "e'". This is necessary also for e'' in $E_{B'''_{i',E''}}$ with $(i',E'') \neq (i,E')$, for e' must also suspend every concurrent event implementation.

5.9. *Terminations of processes with refined events*

The constructed $\mathcal{E}_{B'}$ has all the properties expected for an EES of an E-LOTOS process, implying that B' is an acceptable operand for any E-LOTOS operator, including the just defined refinement operator. Further composition of B' might require employment of $Trm_{B'}$, the attribute indicating when B' is ready for successful termination or for an exception signalling, whose enabling in the case that the exception is trapped also causes termination of B'.

$Trm_{B'}$ in principle behaves as follows: When there is an e in a $\Sigma_{B''_{i,E'}}$ with $(e' \wedge \neg e)$ for e' the prefix of e in $\mathcal{E}_{B'}$, i.e., when a $B'''_{i,E'}$ has just interrupted all other activities in B' by enabling an exception signalling, $Trm_{B'}$ returns the exception $Trm_{B''_{i,E'}}$. Else, when Trm_B returns a $\delta(Data)$ or an $X(Data)$ with no refinement specified, so does $Trm_{B'}$. Otherwise, $Trm_{B'}$ returns "**none**". However, if one simply combines Trm_B and the various $Trm_{B''_{i,E'}}$ as just described, the resulting expression $Trm'_{B'}$ might be referring to an e or an $n(e,c_e)$ with e belonging to an E' for which a $B_{i,E'}$ exists. To refine $Trm'_{B'}$ into $Trm_{B'}$, one must refine every such reference as described in Section 5.6.

Example 8 *Let B' be a "***refine action** $G!\text{true}@?V_1$ **is** B_1 **signal** $X(!\text{true} : \text{bool})$ **is** B_2 **in** B **endref***" with B a "$G?V : \text{bool}$; **signal** $X(V)$", B_1 a "***signal** X_1", and B_2 a "***signal** X_2".*

*In $\mathcal{E}_B$, let e_1 denote the $G(V)$ event, e_2 the $X(V)$ event, and e_3 the implicitly specified $\delta(\ldots)$ event. Trm_B is equivalent to "***if** $e_2 \wedge \neg e_3$ **then** $\delta(V \Rightarrow n(e_1,c_{e_1}).\$2.V)$ **elseif** $e_1 \wedge \neg e_2$ **then** $X(n(e_1,c_{e_1}).\$2.V)$ **else none endif***".*

*In $\mathcal{E}_{B''_{1,\{e_1\}}}$, let e_4 denote the event selecting the initial delay $V_{1,\{e_1\}}$, e_5 the unexecutable event whose aging times the initial delay (see [10]), and e_6 the $X_1()$ event, which in $\mathcal{E}_{B'}$ gets a prefix e'_6. $Trm_{B''_{1,\{e_1\}}}$ is equivalent to "***if** $e_4 \wedge (a(e_5) \geq n(e_4,c_{e_4}).\$2.V_{1,\{e_1\}}) \wedge \neg e_6$ **then** $X_1()$ **else none***".*

*In $\mathcal{E}_{B''_{2,\{e_2\}}}$, let e_7 denote the $X_2()$ event, which in $\mathcal{E}_{B'}$ gets a prefix e'_7. $Trm_{B''_{2,\{e_2\}}}$ is equivalent to "***if** $n(e_1,c_{e_1}).\$2.V \wedge \neg e_7$ **then** $X_2()$ **else none***".*

*$Trm'_{B'}$ is equivalent to "***if** $e'_6 \wedge \neg e_6$ **then** $Trm_{B''_{1,\{e_1\}}}$ **elseif** $e'_7 \wedge \neg e_7$ **then** $Trm_{B''_{2,\{e_2\}}}$ **elseif** $Trm_B \neq X(\text{true})$ **then** Trm_B **else none endif***". It refers to e_1, $n(e_1,c_{e_1})$ and e_2. The presence of $B_{1,\{e_1\}}$ and $B_{2,\{e_2\}}$ implies that those references must be refined.*

6. Discussion and conclusions

6.1. *Refinement of compound events*

The proposed operator facilitates semantic refinement of any non-anonymous (i.e. externally visible) event of a process B, except of its successful termination event, for which a refinement operator already exists in the standard E-LOTOS. Like, for example, [2,5], we allow refinement of events which are interactions of multiple subprocesses of B. This has not been easy to implement, as we have adopted the E-LOTOS semantics from [10], which, unlike other true concurrency semantics for LOTOS and similar languages, intentionally avoids explicit representation of compound events, thereby making process models more concise. Formalizing the refinement operator, we had to introduce some objects corresponding to compound events of B (for example, $e_{i,E'}$ events), but as we did that only for the compound events suspected to be candidates for refined execution, the benefit of using EES process models has to a large extent been preserved.

6.2. *Alternative event implementations*

To our knowledge, this is the first operator allowing specification of multiple alternative refinements per

event. This relieves the specifier from worrying whether the specified refinements refer to different kinds of events. When events carrying data are being refined, the convenience is most welcome, for it might be necessary that the refinement operator refers to infinite families of event names, which might easily intersect. Multiple alternative refinements per event also help when it is not known in advance for which of them the environment of B' will provide the necessary cooperation. Besides, even if two alternative refinements B_i and $B_{i'}$ of a specific event can be adequately combined into a single refinement "$B_i [] B_{i'}$" (because they both commit upon the first event), some other events might have refinement B_i, but not refinement $B_{i'}$, so that a more concise specification might be feasible if B_i is specified as a separate refinement.

6.3. *Observational atomicity of event implementations*

We have adopted the very natural approach of [5,11,6] that refined events are not considered executed until their implementation successfully terminates. The difficulty with the approach is that in any $B'''_{i,E'}$, all events except $e_{i,E'}$ are executed speculatively and become redundant if $e_{i,E'}$ for some reason never occurs. If an event in B has multiple competing implementations, the problem is even more acute. We take care that redundant events in a $B'''_{i,E'}$ never have any influence on (abstract) execution of events in E_B other than those in E', but the fact remains that those which are visible might be bothering for the environment of B'.

One of the situations where redundant visible events arise is when a $B'''_{i,E'}$ commits after a $B'''_{i',E'}$ has already executed some visible events. One should, hence, avoid solving the problem of potentially unsuccessful event implementations by making their commitment late, for this might lead to long periods during which alternative implementations of an event execute their visible events concurrently. It would also be desirable to change the adopted definition of early commitment from choice upon the first visible event to gradual choice upon observational divergences. This would make early commitment more attractive, as such delaying of the choice typically increases the probability that a successful event implementation will be selected.

When redundant visible events in a $B'''_{i,E'}$ arise for a reason other than commitment of an alternative implementation of an e in E', the reason is always a conflict which events in $E_{B'''_{i,E'}}$ inherited from $\langle E' \rangle$ in B. Hence, a less strict inheritance of conflicts would also increase observational atomicity of event implementations. Besides, one could for a $B'''_{i,E'}$ with $\langle E' \rangle$ in B disruptive for an $\langle E'' \rangle$ define that it disrupts a $B'''_{i',E''}$ earlier than upon $end(B'''_{i,E'})$, where it would again be desirable to delay the resolution of the conflict until a critical observational divergence occurs. For an urgent $\langle E'' \rangle$, however, the latter approach is applicable only with special care.

In any case, prevention of redundant visible events is a safety requirement for B' which is often in conflict with requirements for its liveness and fairness, implying that it is in general impossible to find a solution ideal for all environments in which B' might be embedded. Particularly problematic are asymmetric conflicts, where one of the involved events often gets logically enabled unpredictably earlier than the other, or it is for an inferior event not known in advance whether it will be re-enabled if it is disabled. In E-LOTOS, the sources of such conflicts are the disabling operator, the suspend/resume operator and exception signallings, which are problematic also for their urgency. So far, action refinement in the presence of asymmetric conflicts has received very little attention for timed LOTOS. Actually, only timeouts have been considered [5,11].

6.4. *Inheritance of causality*

In Section 4.6, we mentioned that it would be desirable to allow that a $B_{i,E'}$ starts before $\langle E' \rangle$ is logically enabled in B, so that it can terminate in time even if its duration is long. Here, the question arises to what extent should fine-grain events be allowed to overtake events which in principle come earlier, for one must be aware that when an event overtakes another on which it conceptually depends, B' becomes obliged to proceed in the direction of the latter [9]. As such a decision might be premature for the particular environment, it is typically required that the kinds of events which commute are specified explicitly [12], but this means additional work for the specifier. An interesting compromise would be to allow overtaking, but only where the resulting commitments cause no conflict resolution.

6.5. *Concluding remarks*

All the earlier action refinement operators for timed LOTOS work with dialects allowing only the most basic forms of process composition and no data handling. We have proposed an event refinement operator for E-LOTOS with true concurrency with which it is possible to also handle events which use and/or generate data, have an only partially predictable external appearance and/or are shared by multiple processes, unpredictably disabled and re-enabled, subject to complicated timing constraints and/or urgent. For each kind of visible events, it is possible to specify multiple alternative refinements. It is also possible to selectively activate measures for prevention of redundant visible events, although one must be aware that such measures, particularly the extremely simple ones which our operator employs, potentially compromise the liveness or the fairness of the process.

More advanced methods for prevention of redundant visible events are, hence, an important topic for further study. Another would be to allow less strict inheritance

of causality, so that event implementations can have a longer duration and run more concurrently. Both enhancements seem to call for a process model even more advanced than EES.

References

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

[2] J.-P. Courtiat, D.-E. Saïdouni, Action Refinement in LOTOS, Proc. PSTV'93, IFIP Transactions C-16, North-Holland, Amsterdam, 1993, pp. 341–354.

[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, M. Majster-Cederbaum, J. Wu, Refinement of actions in a real-time process algebra with a true concurrency model, Electronic Notes in Computer Science 70(3) (2002) 620–640.

[6] H. Fecher, M. E. Majster-Cederbaum, Action refinement applied to late decisions, Formal Aspects of Computing 18(2) (2006) 211–230.

[7] M. Kapus-Kolar, Restoring the concept of observability in E-LOTOS, Computer Standards and Interfaces 22(1) (2000) 55–60.

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

[9] M. Kapus-Kolar, Towards weak sequencing for E-LOTOS, Computer Standards and Interfaces 28(1) (2005) 59–73.

[10] M. Kapus-Kolar, Enhanced event structures: Towards a true concurrency semantics for E-LOTOS, Computer Standards and Interfaces 29(2) (2007) 205–215.

[11] G. Qin, J. Wu, Action refinement for real-time concurrent processes with urgency, Proc. ARTS'04, Electr. Notes Theor. Comput. Sci. 139(1) (2005) 123–144.

[12] A. Rensink, H. Wehrheim, Process algebra with action dependencies, Acta Informatica 38(3) (2001) 155-234.

[13] D.-E. Saïdouni, J.-P. Courtiat, Syntactic action refinement in presence of multiway synchronisation, Proc. SoSL'93, Springer, 1994, pp. 289–303.

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

Monika Kapus-Kolar received her B.Sc. degree in electrical engineering from the University of Maribor, Slovenia, and her M.Sc. 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 real-time, concurrent and reactive systems.