Functionality Decomposition of Basic LOTOS Expressions with Generalised Termination, Enabling and Disabling

Monika Kapus-Kolar
Jožef Stefan Institute
Jamova 39, P.O.B. 3000, 1001 Ljubljana, Slovenia
monika.kapus-kolar@ijs.si

Abstract

A correctness-preserving transformation is proposed for functionality decomposition of Basic LOTOS expressions with generalised termination, enabling and disabling. Given a specification of the required external behaviour, i.e. the expected service of a distributed system, and a partitioning of the specified actions among the system components, the transformation derives behaviour of individual system components implementing the service. There may be an arbitrary finite number of system components pairwise communicating over auxiliary LOTOS gates.

1 Introduction

The paper studies how introduction of some of the new features currently proposed for enhancement of the formal specification language LOTOS ([1], [5]) would influence the top-down design of specifications.

The top-down design strategy starts by describing the required external behaviour of a system (its service), followed by gradual refinement of its internal structure. When designing a distributed system of hardware or software components, the crucial top-down transformation is functionality decomposition, used to decompose a given process into a number of concurrent interacting subprocesses. [2] provides pointers to several well-documented examples.

The most elegant way is to start with a verified service specification and then apply to it only verified correctness-preserving transformations, thereby avoiding a posteriori verification of the obtained detailed system specification. If a system is specified in a formal language, correctness-preserving transformations can be expressed as well-defined syntactic manipulations that can usually be completely automated.

It must be admitted that design by automated transformations is rather uncommon in the current engineering practice, but the reason is definitely not designers' aversion to such style of work. Rather it is the lack of transformations that are well formalised, proven to be correct, efficiently implemented, easy to use and understand and, above all, that correspond to practically useful design steps. In the future, high-quality tools for transformational design seem to be the best way to make formal languages and methods accessible to non-experts.

For discovering correctness-preserving specification transformations, it is convenient if a formal language offers operators for composing specifications of system components directly representing real-life composition of the components. Such are languages based on process algebras, among which we concentrate on LOTOS [5]. Basic LOTOS [1] provides operators for combining behaviours for parallel execution with various degrees of synchronisation, for sequential execution, for alternative execution, and for disruption.

Kant, Higashino and Bochmann [6], and Brinskma and Langerak [2] have defined correctness-preserving transformations for functionality decomposition in Basic LOTOS. Brinksma and Langerak assume that a behaviour is decomposed into two components interacting through a LOTOS gate (i.e. via common actions), while Kant&al. allow decomposition into an arbitrary number of components pairwise communicating over reliable first-in-first-out (FIFO) channels with unbounded message-buffer capacity. In both cases decomposition starts by allocating the actions in a given behaviour specification to individual components, and the obtained specifications of components conveniently reflect the structure of the original behaviour, for the transformation is compositional.

Now LOTOS is being enhanced into E(xtended)-LOTOS [4]. Among the proposed new language features are non-synchronised termination of parallel behaviours, exception handling, suspension/resumption of behaviours and looping. In a simple but consistent way, these features are all covered by the extension to Basic LOTOS proposed in [3], where they are implemented through generalised termination, enabling and disabling. The present paper provides a compositional correctness-preserving transformation for functionality decomposition of Basic LOTOS expressions extended with the new constructs defined in [3]. We are aware that the considered new LOTOS features might in the future end up standardised in a form more general than, or differing from that proposed in [3], but we think that such an exercise in specification transforming is nevertheless a useful contribution to their practical evaluation. The paper should show how difficult is distributed implementation of the new behaviour types, and how new language features can help in distributed implementation.

The proposed functionality-decomposition transformation is targeted towards decomposing a behaviour into an arbitrary number of components pairwise interacting through LOTOS gates. In addition we informally propose some simplifications applying to the bipartition case, and identify the problems to be faced when generalising the transformation to the asynchronously communicating components.

The rest of the paper is organised as follows: Section 2 introduces the employed formalisms and gives a formal statement of the functionality decomposition problem. In Section 3 we present our correctness-preserving compositional transformation. Section 4 contains informal guidelines for proving its correctness. In Section 5 we discuss properties of the transformation and compare it with the similar earlier solutions to the decomposition problem in LOTOS. Section 6 provides a small example of the application of the transformation. Section 7 concludes the paper.

2 Notations and formal statement of the problem

Definition 2.1: *Typographical conventions*

If $\mathcal{X}$ is some universe of elements where 'x' is its name, then variables x, x',…, x_1,… range over elements of $\mathcal{X}$ and variables X, X',…, X_1,… over subsets of $\mathcal{X}$, if not stated otherwise. ■

The language definition is adapted from [3].

Let us consider a distributed system of an arbitrary finite number of components. Each system component might have several gates for interaction with the system environment, the external system gates. Events on those gates are denoted simply by the gate name. They can be non-terminating or terminating. The behaviour of a system on its external gates is referred to as its service, i.e. the behaviour which is to be decomposed, where the external system events represent service primitives.

In addition, a component might execute internal non-terminating events and synchronise with the other components. Let each individual pair of components c and c' be connected via gate $\mathbf{sync}_{cc'}$ facilitating events of the form $\mathbf{sync}_{cc'}!msg$, where msg is the message exchanged in the event. Such parametrisation of events is not allowed by Basic LOTOS (only by full LOTOS) and is not foreseen in the required service specification, but one can pretend that each $\mathbf{sync}_{cc'}!msg$ is a gate between the two components [2]. The internal and the synchronisation events support implementation of the service.

The events described so far constitute the considered universe of the so called simple events. A simple event is characterised by

- its location (the system components participating in its execution),
- its (non-)termination character, and
- its (non-)visibility: internal events are invisible by

Table 1: Specification language

Nr.	Name	Symbol	Syntax	*Act*
(1)	Specification	*Spec*	**spec** *DefBlock* **endspec**	$Act(DefBlock)$
(2)	Definition Block	*DefBlock*	*B* **where** *ProcBlock*	$Act(B)\cup Act(ProcBlock)$
(3)	Definition Block	*DefBlock*	*B*	$Act(B)$
(4)	Process Block	*ProcBlock*	*ProcDef ProcBlock*	$Act(ProcDef)\cup Act(ProcBlock)$
(5)	Process Block	*ProcBlock*	*ProcDef*	$Act(ProcDef)$
(6)	Process Definition	*ProcDef*	**proc** $p:=B_1$ **endproc**	$Act(B_1)$
(7)	Inaction	*B*	**stop**	$\emptyset$
(8)	Action Prefix	*B*	$o;B_1$	$Act(B_1)\cup\{o\}$
(9)	Choice	*B*	$B_1[]B_2$	$Act(B_1)\cup Act(B_2)$
(10)	Parallelism	*B*	$B_1\vert[G]\vert B_2$	$Act(B_1)\cup Act(B_2)$
(11)	Process Instantiation	*B*	p	$Act(B_1)$ where $p:=B_1$
(12)	Hiding	*B*	**hide** N **in** B_1	$Act(B_1)$
(13)	Renaming	*B*	$B_1[H]$	$Act(B_1)\cup H(Act(B_1))$
(14)	Generalised Termination	*B*	t	$\{t\}$
(15)	Generalised Enabling	*B*	$B_1>t>B_2$	$Act(B_1)\cup Act(B_2)\cup\{t\}$
(16)	Generalised Disabling	*B*	$B_1[t>B_2$ $B_1[>B_2$	$Act(B_1)\cup Act(B_2)\cup\{t\}$ $Act(B_1)\cup Act(B_2)$
(17)	Compound Termination	*B*	$v*t$	$\{v,t\}$

definition, while others are basically visible (i.e. their gates are visible for synchronisation upon), but might be declared invisible within a particular context.

To be able to avoid delay between the last non-terminating event of a behaviour and its formal termination, [3] introduced compound events (compound terminations). A compound event consists of a visible non-terminating and of a terminating simple event.

Formally:

Definition 2.2: *Gates and events*

i) $\mathcal{M}$ is the universe of the <u>m</u>ain gates for non-terminating events (those on the system service interface).

ii) $\mathcal{A}$ is the universe of the <u>a</u>uxiliary gates for non-terminating events (the **sync** gates).

iii) $\mathcal{N} = \mathcal{M} + \mathcal{A}$ is the universe of gates for <u>n</u>on-terminating events.

iv) $\mathcal{V} = \mathcal{M} + \{a!msg \mid a \in \mathcal{A}$, *msg* is any message$\}$ is the universe of <u>v</u>isible non-terminating events.

v) $\mathcal{I}$ is the universe of <u>i</u>nternal events, not visible for an observer. One of its members is **i**, the unnamed internal event. From the point of observation equivalence [1], all members of $\mathcal{I}$ are equivalent to **i**.

vi) $\mathcal{O} = \mathcal{V} + \mathcal{I}$ is the universe of non-terminating (go <u>o</u>n) events.

vii) $\mathcal{T}$ is the universe of <u>t</u>ermination gates that includes the names of all terminating event labels.

viii) $\mathcal{G} = \mathcal{N} + \mathcal{T}$ is the universe of <u>g</u>ates.

ix) Function *Gate* identifies the gate (and function *Gates* the gates) to which an event belongs:

$Gate(m) = m$, $Gate(t) = t$, $Gate(a!msg) = a$

$Gates(v) = \{Gate(v)\}$, $Gates(t) = \{t\}$, $Gates(i) = \varnothing$

x) $\mathcal{L} = \mathcal{M} + (\mathcal{I} \setminus \{\mathbf{i}\}) + \mathcal{T}$ is the universe of <u>l</u>ocalisable events (that can be assigned to an individual system component).

xi) $\mathcal{S} = \mathcal{O} + \mathcal{T}$ is the universe of <u>s</u>imple events.

xii) $\mathcal{E} = \mathcal{S} + (\mathcal{V} \times \mathcal{T})$ is the universe of <u>e</u>vents, where its compound members consist of a visible event v and a termination t, and are represented as $v*t$.

xiii) Function *Simp* identifies the universe of simple events on a particular set of gates, or the simple events constituting an event:

Table 2: Semantics of behaviour expressions

Nr.	B	**Axioms and inference rules**	$Gates(B)$
(7)	**stop**		$\varnothing$
(8)	$o;B_1$	$o;B_1 - o \rightarrow B_1$	$Gates(o) \cup Gates(B_1)$
(9)	$B_1[]B_2$	$B_1 - e \rightarrow B_1' \vdash B_1[]B_2 - e \rightarrow B_1'$ $B_2 - e \rightarrow B_2' \vdash B_1[]B_2 - e \rightarrow B_2'$	$Gates(B_1) \cup Gates(B_2)$
(10)	$B_1\|[G]\|B_2$	$B_1 - v \rightarrow B_1', B_2 - v \rightarrow B_2', Gate(v) \in G \vdash B_1\|[G]\|B_2 - v \rightarrow B_1'\|[G]\|B_2'$ $B_1 - e_1 \rightarrow B_1', B_2 - e_2 \rightarrow B_2', (Gates(e_1) \cup Gates(e_2)) \cap \mathcal{T} \neq \varnothing,$ $(Simp(e_1) \cap Simp(G) = Simp(e_2)) \vee (Simp(e_2) \cap Simp(G) = Simp(e_1))$ $\vdash B_1\|[G]\|B_2 - Ev(Simp(e_1) \cup Simp(e_2)) \rightarrow \mathbf{stop}$ $B_1 - o \rightarrow B_1', Gate(o) \notin G \vdash B_1\|[G]\|B_2 - o \rightarrow B_1'\|[G]\|B_2$ $B_2 - o \rightarrow B_2', Gate(o) \notin G \vdash B_1\|[G]\|B_2 - o \rightarrow B_1\|[G]\|B_2'$ $B_1 - e \rightarrow B_1', Gates(e) \cap G = \varnothing, Gates(e) \cap \mathcal{T} \neq \varnothing \vdash B_1\|[G]\|B_2 - e \rightarrow \mathbf{stop}$ $B_2 - e \rightarrow B_2', Gates(e) \cap G = \varnothing, Gates(e) \cap \mathcal{T} \neq \varnothing \vdash B_1\|[G]\|B_2 - e \rightarrow \mathbf{stop}$	$Gates(B_1) \cup Gates(B_2)$
(11)	p	$p := B_1, B_1 - e \rightarrow B_1' \vdash p - e \rightarrow B_1'$	$Gates(B_1)$ where $p := B_1$
(12)	**hide** N **in** B_1	$B_1 - e \rightarrow B_1', Gates(e) \cap N = \varnothing \vdash$ **hide** N **in** $B_1 - e \rightarrow$ **hide** N **in** B_1' $B_1 - e \rightarrow B_1', Gates(e) \cap N = \{n\}$ $\vdash$ **hide** N **in** $B_1 - Hide(n,e) \rightarrow$ **hide** N **in** B_1'	$Gates(B_1) \setminus N$
(13)	$B_1[H]$	$B_1 - e \rightarrow B_1' \vdash B_1[H] - H(e) \rightarrow B_1'[H]$	$\{H(g) \mid g \in Gates(B_1)\}$
(14)	t	$t - t \rightarrow \mathbf{stop}$	$\{t\}$
(15)	$B_1{>}t{>}B_2$	$B_1 - o \rightarrow B_1' \vdash B_1{>}t{>}B_2 - o \rightarrow B_1'{>}t{>}B_2$ $B_1 - e \rightarrow B_1', Gates(e) \cap \mathcal{T} \neq \varnothing, t \notin Gates(e) \vdash B_1{>}t{>}B_2 - e \rightarrow \mathbf{stop}$ $B_1 - e \rightarrow B_1', t \in Gates(e) \vdash B_1{>}t{>}B_2 - Hide(t,e) \rightarrow B_2$	$(Gates(B_1) \setminus \{t\})$ $\cup Gates(B_2)$
(16)	$B_1[t{>}B_2$	$B_1 - o \rightarrow B_1' \vdash B_1[t{>}B_2 - o \rightarrow B_1'[t{>}B_2$ $B_1 - e \rightarrow B_1', Gates(e) \cap \mathcal{T} \neq \varnothing \vdash B_1[t{>}B_2 - e \rightarrow \mathbf{stop}$ $B_2 - o \rightarrow B_2' \vdash B_1[t{>}B_2 - o \rightarrow B_2'{>}t{>}(B_1[t{>}B_2)$ $B_2 - e \rightarrow B_2', Gates(e) \cap \mathcal{T} \neq \varnothing, t \notin Gates(e) \vdash B_1[t{>}B_2 - e \rightarrow \mathbf{stop}$ $B_2 - e \rightarrow B_2', t \in Gates(e) \vdash B_1[t{>}B_2 - Hide(t,e) \rightarrow B_1[t{>}B_2$	$Gates(B_1)$ $\cup(Gates(B_2) \setminus \{t\})$
(17)	$v*t$	$v*t - v*t \rightarrow \mathbf{stop}$	$\{Gate(v), t\}$

$Simp(G) = \{s|Gate(s)\in G\}$,

$Simp(s) = \{s\}$, $Simp(v*t) = \{v,t\}$

xiv) Function Ev returns an event consisting of the members of an E: $Ev(\{s\}) = s$, $Ev(\{v,t\}) = v*t$. ■

Definition 2.3: *Behaviour expressions*

i) $\mathcal{B}$ is the algebra of finite behaviour expressions as defined by production rules (7) to (17) of Table 1. B, B',…, B_1,… range over behaviour expressions in $\mathcal{B}$.

ii) Behaviours may be structured into processes, so let variable p range over process names.

iii) H in Table 1 is a homomorphism on $\mathcal{E}$:

H: $\mathcal{V}\to\mathcal{V}$, $\mathcal{I}\to\mathcal{I}$, $\mathcal{T}\to\mathcal{T}$

$H(v*t) = H(v)*H(t)$

iv) Axioms and inference rules defining semantics of behaviour expressions are given in Table 2. Attribute $Gates(B)$ lists the visible gates of B.

v) $Hide$ is a function for hiding gates:

if g=$Gate(s)$ then $Hide(g,s)\in\mathcal{I}$ else $Hide(g,s) = s$

if g=$Gate(v)$ then $Hide(g,v*t) = t$ else

if g=t then $Hide(g,v*t) = v$ else $Hide(g,v*t) = v*t$

vi) Parentheses are not explicitly shown in Table 1, yet they can be employed in the usual manner to direct parsing of expressions.

vii) The labelled transition relation $-\ e \rightarrow$ extends in the usual manner to event sequences and facilitates various types of behaviour comparison.

viii) A $B_1|[G]|B_2$ may be written as $B_1||B_2$ if the behaviours synchronise on all their events, and as $B_1|||B_2$ if the behaviours don't synchronise at all.

ix) If in a $B_1[t>B_2$ the B_2 never terminates with t, the expression is supposed to be written as $B_1[>B_2$.

x) If $*$ is an operator for composition of behaviours, $*(B_1\ldots B_n)$ is an abbreviation for $(B_1*B_2*B_3*\ldots*B_n)$.■

Definition 2.4: *Specification and its subexpressions*

i) A specification contains a behaviour expression B, representing the body of the main (initially activated) process, and definitions of the auxiliary processes, instantiated in B or in other processes defined in the specification. Its syntax is defined by production rules (1) to (6) of Table 1.

ii) $\mathcal{X}$ is the universe of specification subexpressions that includes instances of the non-terminal symbols defined in the third column of Table 1 and instances of events o, t and v occurring in its fourth column.

iii) Function Act: $\mathcal{X}\rightarrow\mathcal{S}^{P}$ defined in the fifth column of Table 1 lists the simple event labels that occur in the syntax. ■

Definition 2.5: *Functionality decomposition problem*

Given

- a specification $Spec$ of a behaviour B (a service specification) and its subprocesses with no non-executable events, where

  $Act(B)\subseteq\mathcal{L}$, $(Act(B)\backslash\mathcal{T}\backslash\mathcal{I}) = \mathcal{M}$,
- the universe $\mathcal{C}$ of components into which B is to be decomposed,
- for each pair of places $c,c'\in\mathcal{C}$, $c\neq c'$, an auxiliary gate $\mathbf{sync}_{cc'}$, where by a $\mathbf{sync}_{cc'}!msg$ the component c transmits the message msg, while the component c' by a $\mathbf{sync}_{cc'}!msg$ receives it, hence

  $\mathcal{A} = \{\mathbf{sync}_{cc'}|\ (c,c'\in\mathcal{C})\wedge(c\neq c')\}$
- functions $Comp$ and $Comps$ defining, where necessary, location of events, and such function $Hide$ that location of events is preserved:

  $Comp$: $Act(B)\rightarrow\mathcal{C}$, $L_c = \{l|\ (Comp(l)=c)\}$

  $Comps(l) = \{Comp(l)\}$, $Comps(\mathbf{sync}_{cc'}!msg)=\{c,c'\}$

  $Comps(v*t) = Comps(v)\cup Comps(t)$

  $Comps(Hide(g,e)) = Comps(e)$

derive for each $c\in\mathcal{C}$ a specification $Spec_c$ defining a behaviour B_c such that

- for $c\in\mathcal{C}$: $((s\in Act(B_c)) \Rightarrow (c\in Comps(s)))$,
- **hide** $\mathcal{A}$ **in** $(|[*]|\{B_c|\ c\in\mathcal{C}\}) \approx B$

  (see Definition 2.7 for $|[*]|$ and [1] for $\approx$)

i.e. a protocol implementing the service. It is desirable that the resulting component specifications reflect the structure of $Spec$. ■

Definition 2.6: *Participating components of a behaviour*

Behaviour attribute $PC(B)$ (Table 4) lists the components that participate in B:

$PC(B) = \{c|\exists s\in Act(B):(c\in Comps(s))\}$ ■

Definition 2.7: *Composition of disjoint sets of system components*

$$B_1|[*]|B_2 = B_1|[\{\mathbf{sync}_{cc'}|(((c\in PC(B_1))\wedge(c'\in PC(B_2)))\vee ((c'\in PC(B_1))\wedge(c\in PC(B_2))))\}]|B_2$$

where $(PC(B_1)\cap PC(B_2)) = \varnothing$ ■

3 Solution

To ensure that the resulting specifications of individual system components reflect the structure of the original system specification (specification of the required system service), let the solution to the problem in Definition 2.5 be expressed as $Spec_c$ = **Imp**($Spec$,c), for $c\in\mathcal{C}$, where the mapping **Imp**(x,c) specifies implementation of a specification subexpression x at a component c:

Definition 3.1: *Mapping* **Imp**

i) $\forall c\in\mathcal{C}$:

Imp(**spec** B **endspec**,c) = **spec T**(B,ε,c) **endspec**

Imp(**spec** B **where** …$ProcDef$… **endspec**,c) =
spec T(B,ε,c) **where** … **Imp**($ProcDef$,c) …
endspec

ii) $\forall c \in \mathcal{C}$: **Imp**(**proc** $p := B_1$ **endproc**,c) =

proc $p(\mathrm{id}) := \mathbf{T}(B_1,\mathrm{id},c)$ **endproc**

For mapping **T**, see Table 3. ■

By **Imp**, the main process B of a specification is mapped into $\mathbf{T}(B,\varepsilon,c)$ and each auxiliary process p equivalent to a B_1 into its local part $\mathbf{T}(B_1,\mathrm{id},c)$. Mapping $\mathbf{T}(B,id,c)$ (Table 3) describes implementation of an instance $<B,id.Ident(B)>$ of a B (Definition 3.2) at c.

An auxiliary process is invoked dynamically, thus several instances of it and of its subbehaviours may coexist, while there is a single instance of the main process. Upon each invocation, an auxiliary process is supplied by an unique process instance identifier (the parameter id), while the identifier of the main process is implicitly the empty string ε. Such parametrisation of processes is not allowed by Basic LOTOS (only by full LOTOS) and is not foreseen in the required service specification, but one can pretend that there is a separate non-parametrised process for each individual parameter value.

Definition 3.2: *Specification subexpressions, their instances and their identifiers*

i) In a given specification *Spec*, each subexpression (subbehaviour) x of the body of its main or of some auxiliary process is assigned its static identifier $Ident(x)$, which must be unique within the process body. Where convenient, the static identifier of a simple event will be indicated by a subscript.

ii) An instance (invocation) of a subexpression x is denoted by $<x,id>$, where id is its unique dynamic identifier. For an x from a process body, $id = id'.Ident(x)$, where id' is the identifier of the particular instance of the process and "." denotes concatenation. The identifier of the main process B is ε, hence if x is a subexpression of B, its instances are dynamically identified by $<x,Ident(x)>$. Note that an x might be repeatedly activated in a loop. In that case it is expected that implementation of x is such that a new instance of x is never activated before properly

Table 3: Mapping **T**

Nr.	B	$\mathbf{T}(B,id,c)$
(7)	**stop**	**stop**
(8a) (8b) (8c)	$o^{c'}{}_{id'};B_1$	for c': $(o;(\mathbf{Send}(c',SC(B_1),id.id',t)>t>\mathbf{T}(B_1,id,c')))$ $\forall c \in (SC(B_1)\backslash\{c'\})$: $(\mathbf{sync}_{c'c}!id.id';\mathbf{T}(B_1,id,c))$ $\forall c \in (\mathcal{C}\backslash\{c'\}\backslash SC(B_1))$: $\mathbf{T}(B_1,id,c)$
(9)	$B_1[]B_2$ (Restriction 2)	$(\mathbf{T}(B_1,id,c)[]\mathbf{T}(B_2,id,c))$
(10)	$B_1\vert[G]\vert B_2$ (Restriction 4)	$(\mathbf{T}(B_1,id,c)\vert[G\cap L_c]\vert\mathbf{T}(B_2,id,c))$
(11)	$p_{id'}$	$p(id.id')$
(12)	**hide** M **in** B_1	**hide** $M\cap L_c$ **in** $\mathbf{T}(B_1,id,c)$
(13)	$B_1[H]$ (Restriction 3)	$\mathbf{T}(B_1,id,c)[H/c]$
(14)	$t^{c'}$	if $c = c'$ then t else **stop**
(15a) (15b) (15c)	$B_1>t^{c'}{}_{id'}>B_2$	for c': $(\mathbf{T}(B_1,id,c')>t>(\mathbf{Send}(c',SC(B_2),id.id',t)>t>\mathbf{T}(B_2,id,c')))$ $\forall c \in (SC(B_2)\backslash\{c'\})$: $(\mathbf{T}(B_1,id,c)[>(\mathbf{sync}_{c'c}!id.id';\mathbf{T}(B_2,id,c)))$ $\forall c \in (\mathcal{C}\backslash\{c'\}\backslash SC(B_2))$: $(\mathbf{T}(B_1,id,c)[>\mathbf{T}(B_2,id,c))$

(16a)
 	
(16b)
 	
(16c)
 	
(16d)
 	
(16e)
 	
(16f)
(16g)
	$B_1[t^{c'}{}_{id'}>B_2$
	(Restriction 5)
		if $t \in T(B_2)$ then
		$\forall c \in (\mathcal{C}\backslash\{c'\}\backslash PC(B_2))$: **stop**
		if $\lvert SC(B_2)\backslash\{c'\}\rvert>1$ then
		for c':
		$(\mathbf{T}(B_1,id,c')[t>(\mathbf{T}(B_2,id,c')>t>(\mathbf{Send}(c',PC(B_2),id.id',t)>t>\mathbf{Send}(c',SC(B_2),id.id',t))))$
		$\forall c \in (PC(B_2)\backslash SC(B_2)\backslash\{c'\})$:
		$(\mathbf{T}(B_1,id,c)[LT(t,c)>(\mathbf{T}(B_2,id,c)[>(\mathbf{sync}_{c'c}!id.id'*LT(t,c))))$
		$\forall c \in (SC(B_2)\backslash\{c'\})$:
		$(\mathbf{T}(B_1,id,c)[LT(t,c)>(\mathbf{T}(B_2,id,c)[>(\mathbf{sync}_{c'c}!id.id';\mathbf{sync}_{c'c}!id.id'*LT(t,c))))$
		else
		for c':
		$(\mathbf{T}(B_1,id,c')[t>$
		$(\mathbf{T}(B_2,id,c')>t>(\mathbf{Send}(c',(PC(B_2)\backslash SC(B_2)),id.id',t)>t>\mathbf{Send}(c',SC(B_2),id.id',t))))$
		$\forall c \in (PC(B_2)\backslash\{c'\})$: $(\mathbf{T}(B_1,id,c)[LT(t,c)>(\mathbf{T}(B_2,id,c)[>(\mathbf{sync}_{c'c}!id.id'*LT(t,c))))$
		else $(\mathbf{T}(B_1,id,c)[>\mathbf{T}(B_2,id,c))$

(17)	$m^{c'}*t^{c''}$ (Restriction 1)	if $c = c'$ then $m*t$ else **stop**

concluding execution of all its previous instances with the same dynamic identifier, so that the identifier is within the surrounding context unique in spite of its re-use. ■

Table 3 summarises the rules for decomposition of all kinds of behaviours. Decomposition of individual behaviour types is thoroughly discussed below in specialised sections. The mapping **T** uses the following not yet defined auxiliary functions:

Definition 3.3: *Starting components of a behaviour*
Behaviour attribute $SC(B)$ lists the components that participate in an initial event of B:
$SC(B) = \{c|\exists e:((B - e \rightarrow)\wedge(c\in Comps(e)))\}$ ■

Computation of $SC(B)$ is discussed in the last subsection of Section 3.

Definition 3.4: *Terminations of a behaviour*
Behaviour attribute $T(B)$ (Table 4) lists the terminations of B: $T(B) = \{t|(B - \ldots e \rightarrow) \wedge (t\in Gates(e))\}$ ■

Definition 3.5: *Local image of a termination*
When a component c detects a global termination t, it executes (if necessary) the corresponding local termination $LT(t,c)$.
if $Comp(t) = c$ then $LT(t,c) = t$ else $LT(t,c)\in (L_c\cap\mathcal{T})$
$(LT(t',c) = LT(t,c)) \Rightarrow (t = t')$
$H(LT(t,c)) = LT(H(t),c)$ ■

Definition 3.6: *Function generating parallel message transmissions*
The function generates a string specifying parallel transmissions of a message msg at a component c to a set of components C. The completion of the task is indicated by a termination t.
if $(C\backslash\{c\}) = \emptyset$ then $\mathbf{Send}(c,C,msg,t) = t$
else $\mathbf{Send}(c,C,msg,t) = (|[t]|\{(\mathbf{sync}_{cc'}!msg;t)$
$| c'\in(C\backslash\{c\})\})$ ■

Comments in Table 3 identify the following additional restrictions on the mapped behaviours:

Restriction 1: $Comp(m) = Comp(t)$

Restriction 2: $\exists c': SC(B_1) = SC(B_2) = \{c'\}$

Restriction 3: $(H(L_c)\subseteq L_c) \wedge$
$\forall t,t'\in Act(B_1):((H(t) = H(t')) \Rightarrow (t = t'))$

Restriction 4:
$(B_1|[G]|B_2 - \ldots \rightarrow B_1'|[G]|B_2') \Rightarrow$
$((((B_1' - e_1 \rightarrow) \wedge (t\in Gates(e_1)) \wedge (t\notin G)) \Rightarrow$
$((B_2' - e_2 \rightarrow) \Rightarrow$
$((e_2\in \mathcal{I}) \vee (Gates(e_2)\cap G \neq \emptyset) \vee$
$(Comps(e_2) = Comps(t))))) \wedge$
$(((B_2' - e_2 \rightarrow) \wedge (t\in Gates(e_2)) \wedge (t\notin G)) \Rightarrow$
$((B_1' - e_1 \rightarrow) \Rightarrow$
$((e_1\in \mathcal{I}) \vee (Gates(e_1)\cap G \neq \emptyset) \vee$
$(Comps(e_1) = Comps(t))))))$

Restriction 5: $(B_1 = \mathbf{stop}) \vee$
$\exists c': ((Act(B_1)\subseteq L_{c'}) \wedge (SC(B_2) = \{c'\}))$

3.1 Implementation of inaction, termination, compound termination and process instantiation

System inaction is implemented as inaction of all system components (Table 3(7)).

A simple termination event is implemented at the component of its location, while the other components are inactive (Table 3(14)).

For a compound termination (Table 3(17)) we require that its constituent simple events both belong to the same component (Restriction 1). At that component the compound termination is implemented, while the other components are inactive.

As defined by mapping **Imp**, each auxiliary process is mapped into its counterparts at individual components. A component invokes its part of a process by supplying it by its unique instance identifier, equal to the unique identifier of the process-call instance (Table 3(11)).

3.2 Implementation of action prefix

Action prefixing '$o^{c'};B_1$' is implemented in the same manner as in [2] and [6] (Table 3(8)): Execution starts by the prefix action o at c'. c' reports completion of o to the starting components of B_1 other than itself, and proceeds to B_1 (8a). The message is the unique identifier of the particular instance of o (Definition 3.2). Each of the components in $SC(B_1)\backslash\{c'\}$ receives the message and consequently enables its part of B_1 (8b).

Each of the components in $C\backslash\{c'\}\backslash SC(B_1)$ immediately proceeds to its part of B_1 (8c), but as it is not a starting component of B_1, its execution of B_1 is delayed until reception of a message signalling completion of a previous event within B_1, and thereby also completion of o.

3.3 Implementation of choice

Implementing a '$B_1[]B_2$' (Table 3(9)) we require (Restriction 2) that both alternatives have an unique and equal starting component. The starting component locally selects one of the alternatives, while execution of the selected alternative at any other component, if necessary, is initiated later on by a message from the starting component or some other component already executing the alternative. As in the previous solutions ([2], [6]), the messages are not introduced by mapping on the level of the '[]' operator, but directly by mapping of composition operators within B_1 and B_2.

By Restriction 2 we avoid distributed decision-making. [2] and [6] adopt the same solution, since any other (e.g. cyclic polling of the starting components of $B_1[]B_2$ to make the decision [10]) seems to ruin the required compositionality of the mapping **T**, even if it is possible to use the new LOTOS constructs [3].

Although our mapping of a $B_1[]B_2$ is syntactically identical as in [2], it is semantically different. Up to now it has been a custom to ensure ([2], [6], [10]) that a potential participant of $B_1[]B_2$ was always informed on the selected alternative, even if it was not its participant. Thereby such a component detected that it had no role in the current instance of $B_1[]B_2$, and proceeded to the subsequent service parts. We observe that such a message is necessary only if the component has duties following execution of $B_1[]B_2$, so a more clear solution is not to introduce the message by mapping of $B_1[]B_2$ itself [6] or by mappings of B_1 and B_2 ([2], [10]), but by mapping of some wider expression containing $B_1[]B_2$.

3.4 Implementation of hiding and renaming

Obviously mapping **T** does not depend on the exact identity of events in the mapped specification. It is sensitive only on their location, their (non-)termination character, and for each pair of terminations, whether they are identical or not. Hiding influences none of the properties, so it commutes with **T**. Observing that the relevant events in M are only those belonging to c, a '**hide** M **in** B_1' is mapped onto a c as defined in Table 3(12).

Renaming might change location of events or make two different terminations identical. By assuming that such renaming is not allowed (Restriction 3 [2]), we get renaming commuting with **T**. Observing that the relevant renamings in H are only those belonging to c (i.e. H/c), a $B_1[H]$ is mapped onto a c as defined in Table 3(13).

3.5 Implementation of parallelism

Like in the previous solutions ([2], [6]), implementation of a '$B_1|[G]|B_2$' (Table 3(10)) follows the so called cross-cut theorem for re-grouping of parallel behaviours. Each component c takes its part of B_1 and its part of B_2 and executes them in parallel, synchronised on the local part $G \cap L_c$ of G.

But one must not forget that with the new LOTOS features [3], B_1 or B_2 might execute a non-synchronised termination, which by the language semantics immediately terminates execution of $B_1|[G]|B_2$. So if a component c executes a termination t of, let's say, B_1, intended to disrupt execution of B_2 at a c', t must be (at least virtually) instantly detected by c'. Therefore we introduce Restriction 4 suitable also for the case of more than two components and of compound terminations, where instant reporting might be within the given language impossible:

B_1 and B_2 must be such that whenever one of them, let's say B_1, enables a termination t at a c, there is no concurrently executable visible event of B_2. I.e. an event enabled by B_2 in the current state is either invisible, or local to c, so that it is immediately disabled upon t, or requires co-operation of B_1, where a properly implemented B_1 is not willing to co-operate while terminating. With that restriction it does not matter whether a c' detects t instantly or not, for at the moment of t and afterwards, c' can execute no external system event within B_2. Anyway, this is important only if $B_1|[G]|B_2$ is followed by another behaviour, for in the opposite case t is an external, and thereby by definition a global system event, instantly stopping the system.

Restriction 4 by induction allows execution of sequences of invisible events within B_2 even after termination t of B_1. Hence if the events belong to different components, they are interleaved with **sync** events. If t is a synchronised termination of $B_1|[G]|B_2$, that can not happen.

3.6 Implementation of generalised enabling

For implementation of a '$B_1 >t^{c'}> B_2$' we propose the mapping in Table 3(15) (observe the similarity with the mapping of action prefix):

The '$>t>$' operator requires that B_2 is enabled only if and after B_1 terminates by t, hence the starting components of B_2 other than c' must wait for a signal upon completion of t. The message is the unique identifier of the instance of t (Definition 3.2).

c' executes its part of B_1 and upon its eventual termination t enables the necessary transmissions of the synchronisation message. Completion of the **Send** procedure regenerates t and thereby enables B_2 at c' (15a).

Each component in $(SC(B_2)\backslash\{c'\})$ executes its part of B_1 until it is disrupted by a message reporting completion of t and introducing B_2 (15b).

A $c \in (C\backslash\{c'\}\backslash SC(B_2))$ executes its part of B_1 until it is disrupted by a starting event of its part of B_2 (15c). Because c is not a starting component of B_2, all the starting events of its part of B_2 are receptions of synchronisation messages issued within B_2, i.e. after termination t of B_1.

3.7 Implementation of generalised disabling

Implementation of a '$B_1[t^{c'}>B_2$' requires solution of two different problems, implementation of the B_2 loop and instant interruption of B_1 by starting events of B_2.

The most interesting starting events of B_2 are non-terminating, that can not be composed with **sync** events to facilitate instant signalling of interrupts between components. Thus all interrupts must be local and one must rely on Restriction 5 [2] that, unless B_1 is a **stop**, B_2 has a single starting component which is also the only participant of B_1.

Hence the non-participants of B_1 are also non-participants of B_2. For those components, the members of $C \setminus PC(B_2) \setminus \{c'\}$, '$B_1[t>B_2$' is mapped into inaction (16a).

The other components execute their part of B_1 (which is for all of them but for an unique starting component of B_2 by definition equivalent to inaction), repeatedly interrupted by their part of B_2. Upon each termination t of the current instance of B_2, c' sends the unique identifier of the current instance of t to all the other participants of B_2. Completion of the **Send** procedure(s) regenerates t and thereby enables the next B_2 at c' (16b,e). At the other participants of B_2, a report on t disrupts the current instance of B_2 and is interpreted as the local image of t that starts the next cycle of the B_2 loop (16c,d,f).

The synchronisation message upon termination t of B_2 is the same in every cycle, and so are messages sent within B_2. Therefore one must ensure that the next cycle is never enabled before all participants of B_2 complete the current cycle. If there is at most one starting place of B_2 other than c', the requirement is implemented by having the starting place receive the message introducing the next cycle after the other participants of B_2 (16e,f). Otherwise the message is first sent to all the participants of B_2 other than c', to disrupt their execution of the current cycle, and afterwards again to the starting places of B_2 other than c', to enable the next cycle (16b,c,d). Some of the messages sent to the starting places in the first step might be redundant (i.e. if the places have already stopped their part of B_2 by themselves), but they are difficult to identify.

Table 3(16g) covers the simple case of generalised disabling with no resumption, i.e. the disabling as defined in Basic LOTOS.

3.8 Computation of behaviour attributes

To perform mapping **T**, one must efficiently compute behaviour attributes $PC(B)$, $SC(B)$ and $T(B)$. Table 4 provides rules for computation of $PC(B)$ and $T(B)$, and for an approximation $SC'(B)$ of $SC(B)$, assuming that Restrictions 1 to 5 are valid and that the service specification contains no non-executable events.

$SC(B)$ is directly defined by the language semantics (Table 2). Its easily computed approximation $SC'(B)$ is such that $SC(B) \subseteq SC'(B) \subseteq PC(B)$. The property ensures that 1) the restrictions on the service specification structure are with the approximate attribute values even more strong than with the exact values, 2) all synchronisations induced by the exact attribute values are also induced by the approximations, 3) messages intended to enable events within a B are only sent to the participants of B. Hence exact values of the attribute may be in the context of our functionality-decomposition rules freely replaced by the approximations. The only inaccuracy in calculation of $SC(B)$ in Table 4 is in row (10), where while composing $SC'(B_1)$ and $SC'(B_2)$, a non-starting, but otherwise executable event might be considered starting.

4 Correctness

A detailed correctness proof for the proposed functionality-decomposition transformation can be found in [8], while here we explain it only informally.

By induction on process calls one first proves that correctness of mapping **Imp** for specifications without auxiliary processes implies correctness of **Imp** for specifications with auxiliary processes. Therefore it is sufficient to prove **T** in the absence of process calls. The crucial point here is that every synchronisation message unambiguously identifies its sender, the subexpression instance whose termination it signals, and the type of the termination.

Table 4: Behaviour attributes $PC(B)$ and $T(B)$, and an approximation $SC'(B)$

Nr.	B	$PC(B)$	$T(B)$	$SC'(B)$
(7)	**stop**	$\emptyset$	$\emptyset$	$\emptyset$
(8)	$o^c;B_1$	$\{c\} \cup PC(B_1)$	$T(B_1)$	$\{c\}$
(9)	$B_1[]B_2$	$PC(B_1) \cup PC(B_2)$	$T(B_1) \cup T(B_2)$	$SC'(B_1) \cup SC'(B_2)$
(10)	$B_1\|[G]\|B_2$	$PC(B_1) \cup PC(B_2)$	$T(B_1) \cup T(B_2)$	$SC'(B_1) \cup SC'(B_2)$
(11)	p where $p := B_1$	$PC(B_1)$	$T(B_1)$	$SC'(B_1)$
(12)	**hide** M **in** B_1	$PC(B_1)$	$T(B_1)$	$SC'(B_1)$
(13)	$B_1[H]$	$PC(B_1)$	$H(T(B_1))$	$SC'(B_1)$
(14)	t^c	$\{c\}$	$\{t\}$	$\{c\}$
(15)	$B_1>t^c>B_2$	$PC(B_1) \cup PC(B_2) \cup \{c\}$	$(T(B_1) \setminus \{t\}) \cup T(B_2)$	$SC'(B_1)$
(16)	$B_1[t^c>B_2$	$PC(B_1) \cup PC(B_2) \cup \{c\}$	$T(B_1) \cup (T(B_2) \setminus \{t\})$	$SC'(B_1) \cup SC'(B_2)$
(17)	m^c*t^c	$\{c\}$	$\{t\}$	$\{c\}$

The proof of **T** proceeds basically as in [2] and [6], i.e. by induction on the structure of the behaviour which is being decomposed. For mapping **T** of a B we introduce the following proof obligations:

1) Each component specification contains only events in which the component participates.
2) The distributed implementation of B is observationally equivalent to B.
3) The distributed implementation inherits from B its starting and its participating components.
4) A component can initiate termination of B only in a state which facilitates its global completion even in the case that the termination event is hidden within the surrounding context, i.e. in a state when the other components execute nothing visible.

Correctness of implementation of inaction, a simple termination or a compound termination is evident, since it is entirely local.

For correctness of implementation of choice it is important that implementations of the alternatives inherit their unique starting component. That component performs the selection, while other components are gradually invited into the alternative by the already activated components.

Proving correctness of implementation of an '$o^{c'};B_1$' one observes that:

- Execution of B_1 can not start until c' executes o, hence the behaviour is properly started.
- Synchronisation on t is properly implemented and might be disrupted only by a termination of B_1, executed by a component which has already entered B_1, but that is harmless since such a termination is by definition global. If there are no non-synchronised terminations and $SC(B_1)$ is computed exactly, such disruption is even impossible, since such a termination would be concurrent to a remote starting event of B_1 and that would contradict the properties of B_1. Hence one can pretend that synchronisation on t is always completed.
- At each $c \in (\{c'\} \cup SC(B_1))$, synchronisation on o (plus the resulting invisible termination) is equivalent to an internal event with unknown, but finite duration. Upon its completion, the component enables its part of B_1.
- The components in $(\mathcal{C} \setminus \{c'\} \setminus SC(B_1))$ simply wait until their execution of B_1 is (if ever) enabled by a message issued within B_1, i.e. after completion of o.
- Proper termination of the behaviour is implied by proper termination of B_1.

Proving implementation of hiding and renaming is simple, since with the adopted restrictions the two operations commute with **T**.

Proving implementation of parallelism one observes that execution might be disrupted by a non-synchronised termination, but it is not necessary to worry about it, because it is by definition global. Otherwise the cross-cut theorem for regrouping of parallel behaviours applies. The parallel behaviours by their proper start imply a proper start, and by their proper termination together with Restriction 4 proper termination of their composition.

Proving correctness of implementation of a '$B_1 >t^{c'}> B_2$' one observes that:

- Execution of B_2 can not start until c' executes t, hence proper starting of '$B_1 >t^{c'}> B_2$' is implied by proper starting of B_1.
- Upon t, the other components might by executing only invisible events within B_1, which may be disrupted without any harm and at any time.
- Synchronisation on t is properly implemented and might be disrupted only by a termination of B_2, executed by a component which has already entered B_2, but that is harmless since such a termination is by definition global. If there are no non-synchronised terminations and $SC(B_2)$ is computed exactly, such disruption is even impossible, since such a termination would be concurrent to a remote starting event of B_2 and that would contradict the properties of B_2. Hence one can pretend that synchronisation on t is always completed.
- At each $c \in (\{c'\} \cup SC(B_2))$, synchronisation on t (plus the resulting invisible termination) is equivalent to an internal event with unknown, but finite duration. Upon its completion, the component enables its part of B_2.
- A $c \in (\mathcal{C} \setminus \{c'\} \setminus SC(B_2))$ simply waits until its execution of B_2 is (if ever) enabled by a message issued within B_2, i.e. after termination t of B_1.
- Proper termination of the behaviour is implied by proper termination of B_1 (if other than t) and B_2.

Proving correctness of implementation of a '$B_1 [t^{c'}> B_2$' one observes that:

- Proper starting of the behaviour is implied by proper starting of B_1 and B_2.
- Execution of the next instance of B_2 can not start until c' executes termination t of the current instance.
- At that point, the other components might be executing only invisible events, which may be disrupted without any harm and at any time.
- The components in $PC(B_2)$ can not proceed until they synchronise on t.
- Synchronisation on t is properly implemented and might be disrupted only by a termination of some future cycle of B_2, executed by a component which has already entered it, but that is harmless since such a termination is by definition global. If there are no non-synchronised terminations and $SC(B_2)$ is computed exactly, such disruption is even impossible, since such a termination would be concurrent to a remote starting event of B_2 and that would contradict

the properties of B_2. Hence one can pretend that synchronisation on t is always completed.

- At each $c \in PC(B_2)$, synchronisation on t plus the resulting invisible termination are equivalent to an internal event with unknown, but finite duration. Upon its completion, the component enables the next cycle of the loop.
- Although synchronisation messages employed within B_2 or signalling its completion are reused in consecutive cycles of the loop, their recipients know to which cycle they belong, for the next cycle is never started before global completion of the previous one.
- Hence the B_2 looping part of $B_1[t{>}B_2$ is properly implemented, and so is B_1, because it is according to Restriction 5 a **stop** or executed locally at a c''. Execution of B_1 can affect execution of the concurrently running loop of B_2 only by a termination, but the case may be neglected, since such a termination is by definition global. On the other hand, c'' disables execution of B_1 upon every starting event of B_2, and resumes it upon each report of a termination t of B_2, where this on/off switching is local to c''. A termination of B_2 other than t is by definition global.
- If B_1 terminates, that can only be after termination of an instance of B_2 and before the start of the next instance of B_2, i.e. when the B_2 loop can only be executing invisible events from the last cycle or upon its termination. On the other hand if B_2 terminates, that can only be while execution of B_1 is suspended.

5 Discussion

5.1 Restrictions

Restriction 1 can not be easily avoided. For suppose that a compound event would consist of an m at a c and a t at a c'. If m and t must be executed exactly in the same instant, c and c' must first agree on the time of their execution. In a weaker variant it is only important that there is no other event executed between m and t. In that case either

- all concurrent components must agree on that, or
- m must be immediately reported to all of them, but that requires that m is combined with synchronisation events in all the directions, and that is impossible because composition of two or more non-terminating events is not foreseen [3].

Restriction 2 deals with choice. We are primarily interested in solutions which don't change the mapping of individual alternatives. Since selection of one alternative disrupts the other and since distributed decision-making can be implemented by looping [10], one would expect that the new LOTOS extensions could help.

Suppose that a pair of alternatives has several starting components. A solution would be to immediately report the actual starting event to the other starting components, but that would usually require composition of two or more non-terminating events. If the starting components are given the opportunity to select an alternative one at a time in a loop [10], it is not necessary to report a starting event immediately, but it nevertheless has to be reported, so it must be followed by some specially introduced event, and that ruins compositionality of **T**.

We don't see how Restriction 3 could be avoided without ruining compositionality of **T**, but as [2] suggests, the restriction usually poses no real difficulties.

Restriction 4 can be avoided by combining each non-synchronised termination by synchronisation events for its instant reporting to the disrupted components (not only to the starting components of the subsequent behaviour), but with the adopted language extensions [3] this is only possible if the termination is not already combined with a non-terminating event and only reporting to a single destination is necessary (e.g. in the case of functionality bipartitioning). Implementation of instant reporting of terminations is easy, but exact identification of situations where that is necessary is complicated. Instant reporting of a termination t of a behaviour B can be implemented in a compositional way by associating B with an auxiliary parallel behaviour which synchronises with B on t and combines t with its signalling.

As Restriction 2 can not be suitably avoided, neither can be Restriction 5, because implementation of a '$B_1[t{>}B_2$' repeatedly requires global selection between the starting events of B_2 and all the events in B_1.

5.2 Synchronisation events, process parameters and local images of terminations

For every pair of components c and c', originally communicating over auxiliary gates $\mathbf{sync}_{cc'}$ and $\mathbf{sync}_{c'c}$, it is allowed to rename the two gates into a single auxiliary gate (connecting the two components as an unordered pair). That is because the only difference between transmission and reception of a message between a pair of components is the name of the gate, and the sender and the recipient of the message are already implied by the semantics of mapping **T**.

For short synchronisation messages it is important that identifiers of service specification subexpressions and of their instances are reused whenever possible. In general, the possible reuses can be identified only by detailed analysis of the derived implementation, but sometimes also by analysis of the service specification. As already noted, instance identifiers may be reused upon entering a new cycle of a loop, so one should tend to use loops instead of recursion. For subexpression identifiers it is known that they need to be unique only within the process body to which they belong. In addition we observe that the mapping **T** (Table 3) is such that the identifiers are relevant only for action prefixes, for terminations appearing in the generalised enabling and disabling operators, and for process calls.

As already noted, parametrisation of processes can be avoided by defining a separate process for each possible parameter value. This is particularly simple if a process is called only from the main process and only once. Or sometimes a process requires no synchronisation between components, so its parameter and thereby the identifiers of its call are entirely irrelevant.

In Table 3 we can observe that an $LT(t,c)$ is never visible to the service users, so the exact name of the local image of t is relevant only within the context of c. It can be conveniently set to t, since the location of the event is defined by the context in which it is specified.

5.3 Comparison with earlier solutions

The main difference from the similar earlier solutions [2] and [6] is interpretation of synchronisation-message receptions as disruptions. The scheme has been primarily introduced for implementation of non-synchronised terminations, but as disruptions provide a means for accelerating execution, we employ them whenever convenient. Let us now identify the differences from the earlier solutions, in particularly from [6], as [2] is mostly just its bipartition variant for synchronous channels:

Table 3(8): If B_1 is just a termination not visible to the service users, the synchronisation message might be redundant.

Table 3(9): A potential participant of one of the alternatives is not by definition informed if the other alternative is selected, hence one message per such a component less.

Table 3(10): Unlike in the previous solutions, the parallel behaviours precede a synchronised termination $t^{c'}$ by sending messages to c'. That is because in [3] terminations are regular events, while in non-extended LOTOS not. If the t is not combined by an m, the messages might be redundant.

Table 3(15): Speaking in terms of [6], the ending components (i.e. those executing the events immediately preceding the termination $t^{c'}$) of the selected variant of B_1 report their completion of B_1 to c', which then reports completion of t to the starting components of B_2. In the solution of [6] (and similarly in [2]), each ending component of B_1 (not just of its selected variant) would send a message to each starting component of B_2. And again, if B_2 is just a non-visible termination, the messages enabling it in our solution might be redundant.

Our mapping of disabling (Table 3(16)) is based on the same assumptions on the structure of the mapped behaviour as in [2], but for the same reasons as in Table 3(9), if resumption of B_1 is not foreseen, there are no additional synchronisation messages.

To conclude, for a service specification written in the Basic LOTOS style (no non-synchronised terminations of parallel behaviours, only non-generalised disablings), our mapping is more elegant than the previous solutions, because all communication is concentrated on the two sequential operators ';' and '>>'. If terminating events are carefully employed (no consecutive terminations, a termination always combined with a non-terminating event or allocated to some starting component of the subsequent behaviour), the number of the generated synchronisation messages is also reduced.

5.4 Bipartition

[2] claims that bipartition is sufficient since more complicated service-provider structuring can be achieved by its repeated application. Unfortunately this is not always true, because the resulting specifications might not be properly structured for application of **T**. In particular it might be difficult to avoid the introduced parametrisation of processes, and when trying to avoid Restriction 4 by forbidding compound terminations, mapping **T** actually introduces them for instant termination-reporting. In any case, a multi-component implementation obtained by repeated bipartition is usually slower than an equivalent implementation obtained in a single decomposition step, because not all pairs of components are directly connected and their communication sometimes has to be mediated by other components.

5.5 Asynchronous communication

Components in distributed systems, particularly if geographically separated, often communicate asynchronously over FIFO channels. The decomposition transformation proposed in [6] is originally intended for such channels, and the transformation in [2] can be trivially adapted for them. Our transformation is intrinsically inappropriate for asynchronous channels, because synchronisation messages are received as disruptions and a disrupted behaviour might exit with non-empty input channel buffers, thereby blocking further use of the channels, for implementation of every service subbehaviour is intended to be communication-closed. In addition, in implementation of generalised disabling each cycle of the interrupting loop requires distributed termination, whose simplicity in the proposed mapping **T** is entirely due to the synchronous nature of the auxiliary channels.

However, for a service specification in the Basic LOTOS style (no non-synchronised terminations and only non-generalised disablings), the proposed decomposition transformation seems to be correct also for FIFO channels, provided that computation of attribute *SC* is exact.

5.6 Multiple terminations at the same level

Generalised termination and enabling will probably be standardised in a form that will allow several terminations of a behaviour at the same level. We observe that even in that case the synchronisation scheme underlying the proposed decomposition transformation still applies:

- Executing a non-final termination, the component sends the necessary reports on its completion and subsequently regenerates the termination for local use.
- Executing its part of a behaviour, a component may be disrupted by a report on a remote termination of the behaviour, upon which the component executes the local image of the termination for local use or immediately enables the adequate subsequent behaviour.

6 An example

Figure 1 presents an example service specification. The behaviour consists of a process Work local to c_1 and a concurrent process Communicate. Work repeatedly reads and writes. By Communicate, c_1 connects to c_2, receives data until their end, and finally initiates a shut down procedure. The communication primitives are hidden from the service users. Execution of ShutDown starts when both Work and Communicate are ready. It disconnects c_1 from c_2 and indicates that the system is idle. During system execution, there are several occasions for occurrence of an error at c_1, virtually because c_1 is not satisfied with a message received from c_2 during a phase of Communicate or during ShutDown. An error immediately disrupts regular execution and transfers control to error reporting. Each time a piece of data is received at c_1, Communicate interrupts Work to store the data. In addition, each piece of data received and the end mark of the data stream is logged by c_3.

Figures 2 to 4 specify behaviour derived for the three components constituting the distributed system whose service is defined in Figure 1. The specifications are already simplified, for example unnecessary **stop** subexpressions and some parentheses removed, some enabling operators replaced by action prefixing, some synchronisations combined with the immediately following terminations, etc. Because the only case of multiple instances of a process is invocation of Job whose body does not use the process parameter, it has been possible to entirely avoid process parametrisation, by propagating process parameters directly into process bodies.

The resulting distributed implementation of the specified service can be interpreted as the synchronisation aspect of a protocol. For its practical application, service primitives and synchronisation messages in the protocol specification should be furnished with adequate data parameters, i.e. our transformation extended to full LOTOS.

spec
hide conn_req,conn_ind,con_res,conn_con,data_req,data_ind,end_req,end_ind,disc_req,disc_ind,disc_res,dis_con
in (((Work$_1$[resume$^1_2$>Store$_3$)|[data_save^1,shut_down^1]|Communicate$_4$)
>shut_down$^1_5$>ShutDown$_6$)
>error$^1_7$>ReportError$_8$
where
proc Work:= **stop**[next_job^1>
((Job[read^1/task^1,next_job^1/done^1]|[next_job^1]|Job[write^1/task^1,next_job^1/done^1])[]
shut_down^1) **endproc**
proc Job:= task^1;done^1 **endproc**
proc Store:= data_save^1;data_store^1;resume^1 **endproc**
proc Communicate:= Connect$_1$>connected$^1_2$>Receive$_3$ **endproc**
proc Connect:= conn_req$^1_1$;conn_ind$^2_2$;con_res$^2_3$;((conn_con^1∗connected^1)[]error^1) **endproc**
proc Receive:= **stop**[next_msg$^1_1$>
((data_req$^2_2$;((data_ind$^1_3$;data_log$^3_4$;data_save^1∗next_msg^1)[]error^1))[]
(end_req$^2_5$;((end_ind$^1_6$;end_log$^3_7$;shut_down^1)[]error^1)) **endproc**
proc ShutDown:= disc_req$^1_1$;disc_ind$^2_2$;disc_res$^2_3$;((disc_con^1∗idle^1)[]error^1) **endproc**
proc ReportError:= (error_report_1$^1_1$;**stop**)|||(error_report_2$^2_2$;**stop**) **endproc**
endspec

Figure 1: An example service specification

```
spec
 hide conn_req,conn_con,data_ind,end_ind,disc_req,dis_con in
 (((Work[resume>Store)|[data_save,shut_down]|Communicate)
   >shut_down>ShutDown)
 >error>(sync_12!7;ReportError)
where
 proc Work:= stop[next_job>
                 ((Job[read/task,next_job/done]|[next_job]|Job[write/task,next_job/done])[]
                  shut_down) endproc
 proc Job:= task;done endproc
 proc Store:= data_save;data_store;resume endproc
 proc Communicate:= Connect>connected>(sync_12!4.2;Receive) endproc
 proc Connect:= conn_req;sync_12!4.1.1;sync_21!4.1.3;((conn_con*connected)[]error) endproc
 proc Receive:= stop[next_msg>
                  (((sync_21!4.3.2;((data_ind;sync_13!4.3.3;sync_31!4.3.4;data_save*next_msg)[]error))[]
                    (sync_21!4.3.5;(end_ind;sync_13!4.3.6;sync_31!4.3.7;shut_down)[]error)))
                   >next_msg>(sync_13!4.3.1;sync_12!4.3.1; next_msg)) endproc
 proc ShutDown:= disc_req;sync_12!6.1;sync_21!6.3;((disc_con*idle)[]error) endproc
 proc ReportError:= error_report_1;stop endproc
endspec
```

Figure 2: The first component of the example server

7 Conclusions

A compositional correctness-preserving transformation for functionality decomposition of Basic LOTOS expressions with generalised termination, enabling and disabling has been proposed, accompanied by informal guidelines for proving its correctness and by a small example of its application.

It has been established that unrestricted support of the new language features would be difficult, while with reasonable restrictions on the form of the decomposed behaviour the transformation is simple, elegant and efficient. It seems that the employed hints could be employed also in the setting of pure Basic LOTOS and for components communicating over reliable FIFO channels.

Further work should adapt the transformation to other interesting types of internal system channels and to the future developments of E-LOTOS. It should also extend the transformation to full LOTOS, i.e. to include data parameters. Our experience with distributed mappings of LOTOS-like languages shows that the second task is conceptually simple, but computationally complex: One has to follow how execution of the service influences the distribution of data knowledge among the system components and their needs for the missing data. This information implies the data elements which must be attached to the individual synchronisation messages. Another interesting extension would be to allow dynamic allocation of service primitives, for example to follow migration of service users between the system components [9].

```
spec
 hide conn_ind,con_res,data_req,end_req,
      disc_ind,disc_res in
 (Communicate[>(ShutDown[>(sync_12!7*error)))
 >error>ReportError
where
 proc Communicate:=
        Connect[>(sync_12!4.2;Receive) endproc
 proc Connect:= sync_12!4.1.1;conn_ind;con_res;
                sync_21!4.1.3;stop endproc
 proc Receive:= stop[next_msg>
                 (((data_req;sync_21!4.3.2;stop)[]
                   (end_req;sync_21!4.3.5;stop))
                  [>(sync_12!4.3.1*next_msg)) end-
proc
 proc ShutDown:= sync_12!6.1;disc_ind;disc_res;
                 sync_21!6.3;stop endproc
 proc ReportError:= error_report_2;stop endproc
endspec
```

Figure 3: The second component of the example server

```
spec
 stop[next_msg>
 (((sync_13!4.3.3;data_log;sync_31!4.3.4;stop)[]
   (sync_13!4.3.6;end_log;sync_31!4.3.7;stop))
  [>(sync_13!4.3.1*next_msg))
endspec
```

Figure 4: The third component of the example server

References

[1] Bolognesi, E. Brinskma, “Introduction to the ISO specification language LOTOS,” *Computer Networks and ISDN Systems*, vol. 14, pp. 25-59, 1987.

[2] E. Brinskma, R Langerak, “Functionality decomposition by compositional correctness preserving transformation,” *SACJ/SART*, no. 13, pp. 2-13, 1995.

[3] “Generalised termination, enabling and disabling,” Spanish/Canadian contribution to the E-LOTOS expert group [4] meeting in Liege, Dec. 1995.

[4] Interim reports of the ISO/IEC JTC1/SC21/WG7/E-LOTOS expert group, available by FTP on server ‘ftp.dit.upm.es’ (directory pub/lotos/elotos).

[5] ISO, “Information processing systems - open systems interconnection - LOTOS - a formal description technique based on temporal ordering of observational behaviour,” International Standard 8807, ISO, Geneva, Feb. 1989.

[6] Kant, T. Higashino and G. v. Bochmann, “Deriving protocol specifications from service specifications written in LOTOS,” in *Proc. 12th IEEE Int. Phoenix Conf. on Computer Communications*, Scottsdale, USA, March 93, pp. 310-318.

[7] M. Kapus-Kolar, “New results on deriving protocol specifications from service specifications,” in *Proc.6th Mediterranean Electrotechnical Conf.*, Ljubljana, SI, 1991, pp. 1093-1096.

[8] M. Kapus-Kolar, *Functionality Decomposition of Basic LOTOS Expressions with Generalised Termination, Enabling and Disabling*, Technical report 7414, Jožef Stefan Institute, Ljubljana, 1996.

[9]M. Kapus-Kolar, “A protocol derivation tool for services supporting mobile users,” in *Proc. 4th Electrotechnical and Computer Sc. Conf. ERK’95*, Portorož, SI, 1995, pp. 51-54.

[10] R. Langerak, “Decomposition of functionality: A correctness-preserving LOTOS transformation,” in L. Logrippo, R. Probert, H. Ural (eds.) *Protocol Specification, Testing, and Verification X*, North-Holland, 1990, pp. 229-242.