Functionality Decomposition by Compositional Correctness Preserving Transformation

Ed Brinksma, Rom Langerak, Peter Broekroelofs *

Tele-Informatics and Open Systems Group, Department of Computer Science
University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands
brinksma@cs.utwente.nl, langerak@cs.utwente.nl

Abstract. In this paper we present an algorithm for the decomposition of processes in a process algebraic framework. Decomposition, or the refinement of process substructure, is an important design principle in the top-down development of concurrent systems. In the approach that we follow the decomposition is based on a given partition of the actions of a system specification, such that for each partition class a subprocess must be created that realizes the actions in that class. In addition a suitable synchronization structure between the subprocesses must be present to ensure that the composite behaviour of the subprocesses is properly related to the behaviour of the original specification. We present our results for the process-algebraic specification language LOTOS and give a compositional algorithm for the transformation of the original specification into the required subprocesses. The resulting specification is observation congruent with the original, and, interestingly enough, the subprocesses inherit much of the structure of the original specification. The correctness preserving transformation has been implemented in a tool and has been used for the derivation of protocol specifications from formal descriptions of the desired service. This is possible as it can be shown that the required synchronization mechanisms between the subprocesses can be readily implemented over (reliable) asynchronous media.

1 Introduction

In order to make process algebraic calculi a useful tool for engineering concurrent systems the existing unifying theories must be complemented by non-elementary concepts and constructions that correspond to the designer's needs. In the area of open distributed systems this has led to the definition of the formal specification language LOTOS [ISO89, BoBr87], which was based on ideas in CCS [Mi89] and CSP [Ho85], but has specially adapted constructs for parallel and sequential composition, disruption and the representation of data types. In order to support the actual design of open systems such linguistic facilities must be accompanied by suitably constructed 'high-level' laws that correspond to practically useful design steps. In a top-down design strategy such *correctness-preserving transformations* can be applied to refine a high-level specification into successively better representations of the system as it will be ultimately realized, without further obligations for a posteriori proofs of correctness. Examples of such high-level transformations are the regrouping of subprocesses, the rearrangement of interaction points, from multi-way to binary synchronization, etc. [Bol92].
The transformation principle that we study in this paper, functionality decomposition, is used to decompose a given process into a number of subprocesses that interact concurrently. Such refinement of process substructure can be used to modularize monolithic behaviour into more specialized parts for which implementations or realizations can be more readily found, or to express a notion of geographical distribution, where the different parts correspond to functionalities at different locations. In the

* This work has been supported in part by the CEC research programme ESPRIT BRA (project 6021 REACT)

approach that we follow the decomposition is based on a given partition of the actions of a system specification, such that for each partition class a subprocess must be created that realizes the actions in that class. In addition a suitable synchronization structure between the subprocesses must be present to ensure that the composite behaviour of the subprocesses is properly related to the behaviour of the original specification.

The decomposition of functionality is a natural and frequently occurring design step that can be used for many purposes. Well-documented examples are the design of the PANGLOSS high performance gateway [Bog, Sch92], the design of Manufacturing Planning and Control (MPC) systems [Bie89], the design of the LOTOSPHERE MiniMail system [Sch91], and the derivation of upper and lower testers in conformance test methods [ISO9646, Tre92]. As we will show in an example, the decomposition transformation also makes an important contribution to the problem of deriving protocol specifications from service descriptions.

The work that we present uses the specification language LOTOS as a notational vehicle, but our results can be easily adapted, *mutatis mutandis*, to other process algebraic calculi that can represent similar forms of parallel composition, such as e.g. CSP [Ho85] and CIRCAL [Mil85]. The correctness criterion that we use is the notion of observation congruence, because of its elegant proof technique of constructing (rooted) weak bisimulations, and the fact that it is a rather fine relation and thus implies many other interesting semantic relations, such as for example testing preorders [DeHe 84].

This paper extends earlier work by one of the authors in [La90], where the same transformation is studied under the more restricting assumption that the behaviour of the process that is to be split is given in its fully expanded format (the so-called *monolithic specification style* in [VSvSB91]). One drawback is that this restricts the application of the transformation to a particular syntactic (normal) form. More seriously, however, is that it has the drawback that the algorithm generates elaborate synchronization schemes between subprocesses where these are not needed. By the interleaving interpretation of parallel composition information about the independence of actions in different components is lost when an already structured behaviour expression is expanded. The algorithm would in such cases enforce synchronizations to maintain the various interleaving orders, which is clearly inefficient.

A possible solution for this problem would be to conduct confluence analysis on the expanded behaviour, but it is much better to avoid the problem altogether by following a compositional approach, in which the already available structural information of the behaviour expression is taken into account. The latter approach, which proves feasible under reasonable assumptions, is the one that we follow in this paper. It turns out that most spurious synchronizations can be avoided in this way, and, interestingly enough, that the subprocesses inherit much of the structure of the original specification.

The rest of the paper is organized as follows: section 2 gives an introduction to the formalisms that we use and contains a formal statement of the problem of functionality decomposition; in section 3 we present the compositional transformation algorithm; section 4 contains (a part of) the correctness proof of our algorithm and discusses its restrictions and extensions; section 5 contains a small elaborated example of the application of the transformation and discusses the available tool support; finally, in section 6 we give an overview of related work and present our conclusions.

2 Notations and formal statement of the problem

As indicated in the introduction we use the process algebraic language LOTOS as our notational vehicle. As parametization and value passing are not essential to our formulation of the decomposition problem we restrict ourselves to so-called Basic LOTOS [BoBr87]. Note that the action i in LOTOS denotes the internal action or silent step (cf. τ in CCS).

Definition 2.1 Let L be a set of action labels and PN a set of process names. Let $g \in L \cup \{\mathbf{i}\}$, $a \in L \cup \{\mathbf{i}, \delta\}$, $G \subseteq L$, and $P \in PN$. Let H be a function $H : L \cup \{\mathbf{i}, \delta\} \rightarrow L \cup \{\mathbf{i}, \delta\}$ with $H(\mathbf{i}) = \mathbf{i}$ and $H(\delta) = \delta$.

The syntax of a process definition is $P := B$, where B is a behaviour expression. A set of process definitions $\{P := B_P \mid P \in PN\}$ is called a *process environment*. A LOTOS specification is a behaviour expression in the context of a process environment.

The syntax and operational semantics of Basic LOTOS behaviour expressions is given by table 1. □

Name	Syntax	Axioms and inference rules
inaction	**stop**	
successful termination	**exit**	$\mathbf{exit} \xrightarrow{\delta} \mathbf{stop}$
action prefix	$g\ ; B$	$g; B \xrightarrow{g} B$
choice	$B_1 [] B_2$	$B_1 \xrightarrow{a} B_1' \ \vdash \ B_1[]B_2 \xrightarrow{a} B_1'$
		$B_2 \xrightarrow{a} B_2' \ \vdash \ B_1[]B_2 \xrightarrow{a} B_2'$
enabling	$B_1 >> B_2$	$B_1 \xrightarrow{a} B_1', a \neq \delta \ \vdash \ B_1 >> B_2 \xrightarrow{a} B_1' >> B_2$
		$B_1 \xrightarrow{\delta} B_1' \ \vdash \ B_1 >> B_2 \xrightarrow{\mathbf{i}} B_2$
disabling	$B_1 [> B_2$	$B_1 \xrightarrow{a} B_1', a \neq \delta \ \vdash \ B_1 [> B_2 \xrightarrow{a} B_1' [> B_2$
		$B_1 \xrightarrow{\delta} B_1' \ \vdash \ B_1 [> B_2 \xrightarrow{\delta} B_1'$
		$B_2 \xrightarrow{a} B_2' \ \vdash \ B_1 [> B_2 \xrightarrow{a} B_2'$
hiding	**hide** G **in** B	$B \xrightarrow{a} B', a \in G \ \vdash$ **hide** G **in** $B \xrightarrow{\mathbf{i}}$ **hide** G **in** B'
		$B \xrightarrow{a} B', a \notin G \ \vdash$ **hide** G **in** $B \xrightarrow{a}$ **hide** G **in** B'
renaming	$B[H]$	$B \xrightarrow{a} B' \ \vdash \ B[H] \xrightarrow{H(a)} B'[H]$
parallel composition	$B_1 \vert[G]\vert B_2$	$B_1 \xrightarrow{a} B_1', a \notin G \cup \{\delta\} \ \vdash \ B_1\vert[G]\vert B_2 \xrightarrow{a} B_1'\vert[G]\vert B_2$
		$B_2 \xrightarrow{a} B_2', a \notin G \cup \{\delta\} \ \vdash \ B_1\vert[G]\vert B_2 \xrightarrow{a} B_1\vert[G]\vert B_2'$
		$B_1 \xrightarrow{a} B_1', B_2 \xrightarrow{a} B_2', a \in G \cup \{\delta\} \ \vdash$
		$B_1\vert[G]\vert B_2 \xrightarrow{a} B_1'\vert[G]\vert B_2'$
process instantiation	P	$P := B, B \xrightarrow{a} B' \ \vdash \ P \xrightarrow{a} B'$

Table 1. Basic LOTOS syntax and semantics

In the formal statement of the problem we need the following definition:

Definition 2.2 We define the set of all action labels that occur in a specification B, denoted by $Act(B)$, by:

$Act(\mathbf{stop}) = \emptyset$, $Act(\mathbf{exit}) = \emptyset$
$Act(g; B) =$ if $g \neq \mathbf{i}$ then $Act(B) \cup \{g\}$ else $Act(B)$
$Act(B[H]) = Act(B) \cup H(Act(B))$
$Act(\mathbf{hide}\ G\ \mathbf{in}\ B) = Act(B)$
$Act(P) = Act(B)$ if $P := B$
$Act(B_1 * B_2) = Act(B_1) \cup Act(B_2)$ for all other operators $*$. □

So $Act(B)$ contains all the syntactical actions, so also those actions that semantically do not occur because they are renamed or hidden. $Act(B)$ can be obtained by a simple sequential scan of the

specification. Note that $Act(B)$ is in general a superset of the set of labels $L(B)$ as defined in e.g. [Bri92]; this is because $L(B[H]) = H(L(B))$ and $L(\textbf{hide } G \textbf{ in } B) = L(B) - G$.
In this paper we restrict ourselves to a decomposition into two processes as more complicated substrctures can be achieved by its repeated application. In general the two processes are not independent but need to synchronize their behaviours somehow. For this reason we synchronize them over a newly introduced gate *sync*.
The behaviour of the two synchronizing processes should not be different from the behaviour of the initial process. This has two consequences:

- the synchronization gate *sync* has to be hidden;
- the behaviour of the implementation should be in a specific semantic relation to the behaviour of the initial architecture.

As indicated in the introduction we have chosen in this case *observation equivalence* $\approx$ [Mi89] as our implementation relation. We can now describe the problem of splitting an expression B formally as follows: find two expressions $B1$ and $B2$ such that
$$(\textbf{hide } sync \textbf{ in } B1|[sync]|B2) \approx B$$

There are probably many criteria on the basis of which functionality can be distributed. In this paper we only consider decompositions on the basis of a bipartition of the set of all actions $Act(B)$ of an expression B: given a bipartitioning of $Act(B)$, we want a decomposition into two expressions such that the actions of each expression are contained in one bipartition class.

To denote the result of the decomposition we use a slight extension of this Basic LOTOS. The extension consists of the introduction of structured actions of the form *gate!message*; these structured actions are a feature of Full LOTOS [ISO89]. This has the intended meaning that at label *gate* a synchronization takes place on message *message*. It would be possible to do without this extension as we can simulate this in Basic LOTOS by using an action like *gate_message*. However with the structured actions we are able to write down the parallel operator in a more concise way, as $P|[gate]|Q$ means: synchronize on all actions for which the label part is *gate*. Not having the extension would force us to write between the brackets all actions starting with *gate_*. For clarity we give the operational semantics of the parallel operator as used in the extended Basic LOTOS:

Let *name* be a function for which $name(\mathbf{i}) = \mathbf{i}$, $name(g) = g$, and
$name(g!m) = g$, then the inference rules for the parallel operator are given by

1. if $B_1 \xrightarrow{a} B_1'$ and $name(a) \notin \{g_1, \ldots, g_n, \delta\}$ then
   $B_1|[g_1, \ldots, g_n]|B_2 \xrightarrow{a} B_1'|[g_1, \ldots, g_n]|B_2$
2. if $B_2 \xrightarrow{a} B_2'$ and $name(a) \notin \{g_1, \ldots, g_n, \delta\}$ then
   $B_1|[g_1, \ldots, g_n]|B_2 \xrightarrow{a} B_1|[g_1, \ldots, g_n]|B_2'$
3. if $B_1 \xrightarrow{a} B_1'$, $B_2 \xrightarrow{a} B_2'$ and $name(a) \in \{g_1, \ldots, g_n, \delta\}$ then
   $B_1|[g_1, \ldots, g_n]|B_2 \xrightarrow{a} B_1'|[g_1, \ldots, g_n]|B_2'$

We now give a precise statement of the problem of decomposition of functionality :

Given:

- an expression B with set of actions $Act(B) \subseteq A$, $sync \notin A$.
- a partitioning of A into A_1 and A_2, i.e. $A_1 \cup A_2 = A$ and $A_1 \cap A_2 = \emptyset$.

Problem: find two expressions B_1 and B_2 with the following properties:

- $Act(B_1) \cap A \subseteq A_1, Act(B_2) \cap A \subseteq A_2$
- **hide** $sync$ **in** $B_1 |[sync]| B_2 \approx B$.

In order to solve this problem in a general way we would like to have two mappings **T1** and **T2** that, given a partitioning of A, provide us with a B_1 and B_2 for every B, by having $\mathbf{T1}(B) = B_1$ and $\mathbf{T2}(B) = B_2$. In the next section we define such mappings.

3 Solution

In this section we define mappings **T1** and **T2** as discussed in the previous section. We define these mappings in a compositional way, i.e. for each operator we define $\mathbf{T1}(B)$ and $\mathbf{T2}(B)$ in terms of the operands.
It appears that for several operators we have to make some restrictions on B in order to be able to define the mappings. These restrictions are collected and discussed in section 4.2.

3.1 Inaction and successful termination

In these two simple cases the inaction or the successful termination is simply copied to both components of the decomposition:

Definition 3.1

$B = \mathbf{stop} : \mathbf{T1}(B) = \mathbf{stop},\ \mathbf{T2}(B) = \mathbf{stop}$
$B = \mathbf{exit} : \mathbf{T1}(B) = \mathbf{exit},\ \mathbf{T2}(B) = \mathbf{exit}$

□

3.2 Action prefix

The mappings for this operator are based on the following idea: if an action in e.g. $\mathbf{T1}(B)$ has happened, $\mathbf{T2}(B)$ should be notified of this fact in order to produce the appropriate behaviour after the action. This notification is done by synchronizing on messages via the $sync$ gate. So in principle the structure is as follows: suppose $a \in A_1$, then $a; B$ is decomposed into $a; sync!m; \mathbf{T1}(B)$ and $sync!m; \mathbf{T2}(B)$, respectively.
The synchronization message should be unique for each occurrence of an action. For this purpose we assume that each action is subscripted with a unique occurrence identifier, e.g. $a_i; B$. The exact nature of the occurrence identifiers is irrelevant; they could be for instance integers, handed out to action occurrences in order of appearance in an expression.
The occurrence identifiers are used to produce unique synchronization messages: an action a_i may lead to a unique synchronization message m_i. In the rest of this paper we adopt as a convention that $A_1 = \{a_i \mid i \in I\}$ and $A_2 = \{b_j \mid j \in J\}$.
We treat internal actions just like ordinary actions, i.e. they each have a unique occurrence identifier and lead to unique synchronization messages. This implies that in addition to the bipartition of $Act(B)$, the user has to specify for each internal action in an expression B whether it belongs to $\mathbf{T1}(B)$ or $\mathbf{T2}(B)$. We will not bother with formalizing this, but simply assume that $Act(B)$ has been extended by including all occurrences of internal events, and we assume the bipartition of A into A_1 and A_2 includes the bipartitioning of all occurrences of internal events. This poses no intrinsic difficulties; this point will be discussed in section 4.

It is not always necessary that an action prefix results in a synchronization. Suppose we have $a_i; B$ and all initial actions of B are in A_1. Then the first two actions of $a_i; B$ are in $\mathbf{T1}(a_i; B)$ so it is not necessary to synchronize after a_i. In such a case we get a decomposition into $a_i; \mathbf{T1}(B)$ and $\mathbf{T2}(B)$, respectively. The set of initial actions of an expression B is denoted by *init*(B) and is defined by *init*$(B) = \{a \in L \cup \{\mathrm{i}, \delta\} \mid B \xrightarrow{a} \}$.
These considerations lead to the following definition:

Definition 3.2
If $B = a_i; B'$ and $init(B') \subseteq A_1$
then $\mathbf{T1}(B) = a_i; \mathbf{T1}(B')$, $\mathbf{T2}(B) = \mathbf{T2}(B')$
else $\mathbf{T1}(B) = a_i; sync!m_i; \mathbf{T1}(B')$, $\mathbf{T2}(B) = sync!m_i; \mathbf{T2}(B')$

If $B = b_j; B'$ and $init(B') \subseteq A_2$
then $\mathbf{T1}(B) = \mathbf{T1}(B')$, $\mathbf{T2}(B) = b_j; \mathbf{T2}(B')$
else $\mathbf{T1}(B) = sync!m_j; \mathbf{T1}(B')$, $\mathbf{T2}(B) = b_j; sync!m_j; \mathbf{T2}(B')$ □

Note that this definition in the case of e.g. $B = a; \mathbf{exit}$ results in a synchronization between a and **exit**.

3.3 Choice

Consider the expression $B = a_i; B_1 [] b_j; B_2$. In this expression we have a choice between an action a_i that after the decomposition resides in $\mathbf{T1}(B)$, and an action b_j that will be in $\mathbf{T2}(B)$. We call such a choice between actions from different components a *global* choice. The difficulty with such a global choice is that two demands have to be fulfilled :

- both a_i and b_j should be offered to the environment
- once the environment chooses e.g. a_i, immediately b_j should not be offered anymore.

These two demands cannot be fulfilled simultaneously by synchronization after one action has occurred since this synchronization cannot prevent an action from the other component happening first. In order to solve this problem more sophisticated solutions are needed, like the polling mechanism in [La90]. However, such a mechanism conflicts with the compositional approach in this paper. In fact, the occurrence of global choices can often also be interpreted as the inadequacy of the given partition of actions as a basis for the distribution of functionality. For this reason we restrict ourselves in this paper to those cases where global choice does not occur.

Restriction 1:
If $B = B_1 [] B_2$, then either $init(B_1) \cup init(B_2) \subseteq A_1$ or $init(B_1) \cup init(B_2) \subseteq A_2$.

With this restriction it turns out that the definition for the mappings is quite simple:

Definition 3.3
$B = B_1 [] B_2 \; : \; \mathbf{T1}(B) = \mathbf{T1}(B_1) [] \mathbf{T1}(B_1)$, $\mathbf{T2}(B) = \mathbf{T2}(B_1) [] \mathbf{T2}(B_2)$ □

3.4 Hiding and renaming

The mappings for these operators pose no problems. Only for the renaming operator there is the requirement that the renaming should be consistent in the following sense: actions from A_1 can only be renamed into actions from A_1 and actions from A_2 can only be renamed into actions from A_2. This restriction can be weakened by parameterizing the mappings **T1** and **T2** with the partition at stake, and instantiating it with a suitably renamed partition when the algorithm is applied to a renaming expression. We choose to avoid such complications, however, and work with the restriction as stated.

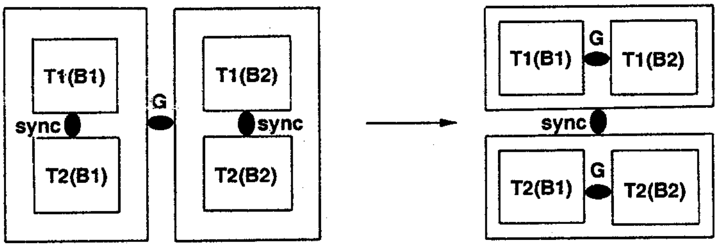
Restriction 2:
If $B = B'[H]$, then $H(A_1) \subseteq A_1$ and $H(A_2) \subseteq A_2$

Definition 3.4
$B = B'[H]$: $\mathbf{T1}(B) = \mathbf{T1}(B')[H]$, $\mathbf{T2}(B) = \mathbf{T2}(B')[H]$
$B =$ **hide** G **in** B' : $\mathbf{T1}(B) =$ **hide** G **in** $\mathbf{T1}(B')$, $\mathbf{T2}(B) =$ **hide** G **in** $\mathbf{T2}(B')$ □

If we denote the restriction of mapping H to e.g. A_1 by $H\lceil A_1$, then we could replace the H in the definition of $\mathbf{T1}(B)$ and $\mathbf{T2}(B)$ for renaming by $H\lceil A_1$ and $H\lceil A_2$, respectively; this could be more clear in practice, but it is not necessary for the correctness of the mappings.
Similarly, we could in the definitions of $\mathbf{T1}(B)$ and $\mathbf{T2}(B)$ for hiding replace the G by $G \cap A_1$ and $G \cap A_2$, respectively.

3.5 Parallelism

The mappings for the parallel operator are compositional in a direct way. The idea behind the mapping is given in the following picture:



The reason we can make this transformation is the following fact:

- synchronizations over G are either between actions from $\mathbf{T1}(B1)$ and $\mathbf{T1}(B2)$ or between actions from $\mathbf{T2}(B1)$ and $\mathbf{T2}(B2)$
- synchronizations over *sync* are either between actions from $\mathbf{T1}(B1)$ and $\mathbf{T2}(B1)$ or between actions from $\mathbf{T1}(B2)$ and $\mathbf{T2}(B2)$

Definition 3.5 $B = B_1|[G]|B_2$:
$\mathbf{T1}(B) = \mathbf{T1}(B_1)|[G]|\mathbf{T1}(B_2)$
$\mathbf{T2}(B) = \mathbf{T2}(B_1)|[G]|\mathbf{T2}(B_2)$ □

For reasons of clarity the G in the definitions of $\mathbf{T1}(B)$ and $\mathbf{T2}(B)$ for parallelism could be replaced by $G \cap A_1$ and $G \cap A_2$, respectively, without changing the semantics.

3.6 Enabling

For the enabling operator we would like to have mappings that share the structural simplicity of the mappings for parallelism, i.e. we would like to have $\mathbf{Ti}(B_1 >> B_2) = \mathbf{Ti}(B_1) >> \mathbf{Ti}(B_2)$, $i = 1, 2$. There is however one problem with this idea:

Example 3.6 Suppose $B_1 = a_i; B_1' [\,] \, \mathbf{exit}$ and $B_2 = b_j; B_2'$. Then
$\mathbf{T1}(B_1 >> B_2) = (a_i; sync!m_i; \mathbf{T1}(B_1')[\,]\mathbf{exit}) >> sync!m_j; \mathbf{T1}(B_2')$
$\mathbf{T2}(B_1 >> B_2) = (sync!m_i; \mathbf{T2}(B_1')[\,]\mathbf{exit}) >> b_j; sync!m_j; \mathbf{T2}(B_2')$
But now the decomposition has an undesirable transition sequence: the second component could first execute the **exit** and then the b_j, after which the first component could still execute the a_i. This is not possible for $B_1 >> B_2$ so the decomposition is not correct. □

The source of this problem is the fact that once an **exit** is within the scope of an enable operator, the **exit** is (semantically) turned into an internal action, and therefore not synchronized anymore with an **exit** in the other component. This leads to a problem when the **exit** is in a choice-context since then the two components may make different unrelated choices. This problem can be avoided by adopting the following restriction:

Restriction 3:
If $B = B_1[\,]B_2$ then $B_1 \overset{\delta}{\not\rightarrow}$ and $B_2 \overset{\delta}{\not\rightarrow}$

Definition 3.7 $B = B_1 >> B_2$:
$\mathbf{T1}(B) = \mathbf{T1}(B_1) >> \mathbf{T1}(B_2)$
$\mathbf{T2}(B) = \mathbf{T2}(B_1) >> \mathbf{T2}(B_2)$ □

3.7 Disabling

This is the most tricky operator for this transformation. First of all we make a restriction that is similar to the one we made for choice, where we did not allow global choice. For if e.g. $B = a_i : \mathbf{stop}$ [> $b_j; \mathbf{stop}$ we face a similar problem as for global choice: a_i and b_j should both be offered, but at the moment e.g. b_j happens a_i instantly cannot happen anymore. This cannot be achieved by just synchronizing after actions. For this reason we want all the actions of B_1 and all initial actions of B_2 to be either all in A_1 or all in A_2.

Restriction 4: If $B = B_1$ [> B_2 then
$Act(B_1) \cup init(B_2) \subseteq A_1$ or $Act(B_1) \cup init(B_2) \subseteq A_2$

The definition of the mappings for disabling is more complicated than for the other operators. We first give the definition and then discuss it.

Definition 3.8 $B = B_1$ [> B_2 :

if $Act(B_1) \subseteq A_1$ then $\mathbf{T1}(B) = (B_1$ [> $\mathbf{T1}(B_2)) >> sync!m_\delta$; **exit**
$\mathbf{T2}(B) = (sync!m_\delta$; **exit**) [] ($\mathbf{T2}(B_2) >> sync!m_\delta$; **exit**)
if $Act(B_1) \subseteq A_2$ then $\mathbf{T1}(B) = (sync!m_\delta$; **exit**) [] ($\mathbf{T1}(B_2) >> sync!m_\delta$; **exit**)
$\mathbf{T2}(B) = (B_1$ [> $\mathbf{T2}(B_2)) >> sync!m_\delta$; **exit**

□

The main trick of this definition is that any δ happening in B_1 or B_2 is "caught" by an enabling operator, after which a synchronization takes place on a special message m_δ, followed by an **exit** in order to again generate a δ. In this way the problem of unsynchronized successful termination as illustrated in example 3.6 is avoided. Since all actions of B_1 happen in one component there is no need to apply a mapping to B_1; we only have to add *sync*!m_δ ; **exit** in a choice with the other component in order to synchronize with the other side in case a δ occurs in B_1.

There is only one situation in which definition 3.8 goes wrong, namely in the case that $B_2 \xrightarrow{\delta}$. Suppose we would have $Act(B_1) \subseteq A_1$; then **T2**(B) could perform an initial event, enabling thereby *sync*!m_δ ; **exit**, whereas **T1**(B) could still perform an action from B_1, thereby potentially causing a deadlock. It is hard to repair this defect; therefore we simply add the restriction that B_2 cannot have δ as an initial action:

Restriction 5: If $B = B_1$ [> B_2 then $B_2 \overset{\delta}{\not\rightarrow}$

3.8 Process definition and instantiation

Each process instantiation P is transformed into a process instantiation $P1$ in the first component and $P2$ in the second component. This means there have to be process definitions for $P1$ and $P2$ in the process environment, so we extend the use of **T1** and **T2** in order to split process definitions:

Definition 3.9 Process definition:
$\mathbf{T1}(P := B) \;=\; P1 := \mathbf{T1}(B)$
$\mathbf{T2}(P := B) \;=\; P2 := \mathbf{T2}(B)$
Instantiation: $\mathbf{T1}(P) = P1$, $\mathbf{T2}(P) = P2$ □

4 Correctness and discussion

4.1 Correctness

Theorem 4.1 Let B be a Basic LOTOS behaviour expression. Then
hide *sync* **in** $\mathbf{T1}(B)$ |[*sync*]| $\mathbf{T2}(B) \;\approx\; B$

Proof: by structural induction; for example, for each binary operator $*$ we prove that **hide** *sync* **in** $\mathbf{T1}(B_1 * B_2)$ |[*sync*]| $\mathbf{T2}(B_1 * B_2)$ is observation congruent with $B_1 * B_2$, under the induction hypothesis that $\mathbf{T1}(B_1)$ and $\mathbf{T2}(B_2)$ are observation congruent with B_1 and B_2 respectively. The complete proof is rather lengthy and can be found in [Bro92].

As an example we prove the above theorem for the parallel operator. We need the following two laws:

Law 1 [Bri92]
If $(G - G') \cap (Act(B_1) \cup Act(B_2)) = \emptyset$ then $B_1|[G]|B_2 \approx^c B_1|[G \cap G']|B_2$

Law 2 [VSvSB91]
If $Act(A) \cap (S2 \cup I2) = \emptyset$, $Act(B) \cap (S2 \cup I1) = \emptyset$, $Act(C) \cap (S1 \cup I2) = \emptyset$, and $Act(D) \cap (S1 \cup I1) = \emptyset$, then
$(A|[S1]|B)|[I1 \cup I2]|(C|[S2]|D) \;\approx^c\; (A|[I1]|C)|[S1 \cup S2]|(B|[I2]|D)$

hide *sync* **in** $\mathbf{T1}(B)$ |[*sync*]| $\mathbf{T2}(B)$
$=_{df}$ **hide** *sync* **in** $(\mathbf{T1}(B_1)$ |[G]| $\mathbf{T1}(B_2))$ |[*sync*]| $(\mathbf{T2}(B_1)$ |[G]| $\mathbf{T2}(B_2))$
$\approx^c$ (Law 1, $G1 = G \cap A_1$, $G2 = G \cap A_2$)
hide *sync* **in** $(\mathbf{T1}(B_1)$ |[$G1$]| $\mathbf{T1}(B_2))$ |[*sync*]| $(\mathbf{T2}(B_1)$ |[$G2$]| $\mathbf{T2}(B_2))$

We rename $\mathbf{T1}(B_1)$ and $\mathbf{T2}(B_1)$ into $\mathbf{T1'}(B_1)$ and $\mathbf{T2'}(B_1)$, with the renaming $[\mathit{sync1}/\mathit{sync}]$. Analogous for $\mathbf{T1}(B_2)$, $\mathbf{T2}(B_2)$ and $[\mathit{sync2}/\mathit{sync}]$. This can be done since $\mathbf{T1}(B_1)$ only synchronizes with $\mathbf{T2}(B_1)$ and $\mathbf{T1}(B_2)$ only synchronizes with $\mathbf{T2}(B_2)$:

$\approx^c$ **hide** $\mathit{sync1}, \mathit{sync2}$ **in** $(\mathbf{T1'}(B_1) \mid[G1]\mid \mathbf{T1'}(B_2)) \mid[\mathit{sync1}, \mathit{sync2}]\mid (\mathbf{T2'}(B_1) \mid[G2]\mid \mathbf{T2'}(B_2))$

$\approx^c$ (Law 2. The constraints are satisfied:
$(\mathit{Act}(\mathbf{T1'}(B_1)) \cup \mathit{Act}(\mathbf{T2'}(B_1))) \cap \{\mathit{sync2}\} = \emptyset$
and $(\mathit{Act}(\mathbf{T1'}(B_2)) \cup \mathit{Act}((\mathbf{T2'}(B_2))) \cap \{\mathit{sync1}\} = \emptyset$.)
hide $\{\mathit{sync1}, \mathit{sync2}\}$ **in** $(\mathbf{T1'}(B_1) \mid[\{\mathit{sync1}\}]\mid \mathbf{T2'}(B_1)) \mid[G]\mid (\mathbf{T1'}(B_2) \mid[\{\mathit{sync2}\}]\mid \mathbf{T2'}(B_2))$

$\approx^c$ (Several obvious laws for hiding, see [VSvSB91])
(**hide** $\{\mathit{sync1}\}$ **in** $\mathbf{T1'}(B_1) \mid[\{\mathit{sync1}\}]\mid \mathbf{T2'}(B_1))$
$\mid[G]\mid$ (**hide** $\{\mathit{sync2}\}$ **in** $\mathbf{T1'}(B_2) \mid[\{\mathit{sync2}\}]\mid \mathbf{T2'}(B_2))$

$\approx^c$ (Renaming *sync1* and *sync2* into *sync*, induction hypothesis)
$(B_1 \mid[G]\mid B_2)$

$=_{df}$ B

4.2 Restrictions

The mappings in the previous section were defined under the following five restrictions:

1. If $B = B_1[]B_2$, then either $\mathit{init}(B_1), \mathit{init}(B_2) \subseteq A_1$ or $\mathit{init}(B_1), \mathit{init}(B_2) \subseteq A_2$
2. If $B = B'[H]$, then $H(A_1) \subseteq A_1$ and $H(A_2) \subseteq A_2$
3. If $B = B_1[]B_2$ then $B_1 \overset{\delta}{\not\to}$ and $B_2 \overset{\delta}{\not\to}$.
4. If $B = B_1$ [> B_2 then $\mathit{Act}(B_1) \cup \mathit{init}(B_2) \subseteq A_1$ or $\mathit{Act}(B_1) \cup \mathit{init}(B_2) \subseteq A_2$
5. If $B = B_1$ [> B_2 then $B_2 \overset{\delta}{\not\to}$

Restriction 2 seems reasonable and poses no real difficulties as actions in B' can often be syntactically renamed, instead of being renamed by $[H]$, in order to meet the restriction.

Also restrictions 3 and 5 are not very restrictive. For example, consider (a; **stop** [] **exit**) >> b; **stop**. This expression does not meet restriction 3. However, it can be changed into the weak bisimulation congruent expression (a; **stop** [] i; **exit**) >> b; **stop** that does not violate restriction 3. Similarly, (a; **stop** [> **exit**) >> b; **stop** (violating restriction 5) can be replaced by the observation congruent (a; **stop** [> i; **exit**) >> b; **stop**.

Restrictions 1 and 4, prohibiting global choice and global disabling, are met by a large class of specifications. Often it is quite unnatural to specify a choice between actions at different locations. Still there are specifications that inherently have such a global choice. In such a situation we might be able to circumvent restriction 1 by incorporating a kind of polling mechanism along the lines of [La90]. This is for further study.

We do not see how restriction 4 could be avoided in a natural way. The only way seems to be to expand the disabling operator away, and then applying a polling mechanism for the resulting global choices.

4.3 Internal actions

In section 3 it was remarked that internal actions are to be treated just like observable actions and have to be bipartitioned by the user. In practice the constraints 2 and 4 take away a lot of freedom of choice, thereby lessening the burden of bipartioning for the user. Many choices can be made automatically. For example, in the expression $a; B \; [] \; \mathbf{i}; B'$ the only possibility is that $\mathbf{i}$ is allocated to the same component as a, in order to meet restriction 1. In fact the only thing the user really has to decide is in which component symmetric nondeterministic choices like $\mathbf{i}; B \; [] \; \mathbf{i}; B'$ have to take place.

4.4 Asynchronous communication

The components $\mathbf{T1}(B)$ and $\mathbf{T2}(B)$ as defined in the previous section interact by synchronous communication. It may not always be realistic to expect that such a synchronous communication can be realized. Most notably, if $\mathbf{T1}(B)$ and $\mathbf{T2}(B)$ reside at geographically different locations we may not be able to implement in an efficient way synchronous communication. In this case asynchronous communication using some reliable communication medium is needed.
This means we have to replace the *sync* actions by *send* and *receive* actions, in the following way:
- component $\mathbf{T1}(B)$ sends and receives messages over process *Medium* via gates *send*1 and *receive*1
- component $\mathbf{T2}(B)$ sends and receives messages over process *Medium* via gates *send*2 and *receive*2

For example, definition 3.2 has to be changed into the following definition:

Definition 4.2
If $B = a_i; B'$ and $init(B') \subseteq A_1$
then $\mathbf{T1}(B) = a_i; \mathbf{T1}(B')$, $\mathbf{T2}(B) = \mathbf{T2}(B')$
else $\mathbf{T1}(B) = a_i; send1!m_i; \mathbf{T1}(B')$, $\mathbf{T2}(B) = receive2!m_i; \mathbf{T2}(B')$

If $B = b_j; B'$ and $init(B') \subseteq A_2$
then $\mathbf{T1}(B) = \mathbf{T1}(B')$, $\mathbf{T2}(B) = b_j; \mathbf{T2}(B')$
else $\mathbf{T1}(B) = receive1!m_j; \mathbf{T1}(B')$, $\mathbf{T2}(B) = b_j; send2!m_j; \mathbf{T2}(B')$ □

The correctness proof of the transformation using asynchronous communication is quite involved. Since the components are now less tightly coupled, the construction of a bisimulation relation for proving the correctness is not that easy. We plan to study the correctness for the asynchronous case with the help of an alternative semantics for LOTOS that is defined in [La92].

5 Example and tool support

We give an example of the transformation by considering a simple example of a service. The service *SimpleService* starts with a connect phase in which an entity at location *a* can establish a connection with an entity at location *b*. After the connection has been established the two entities can exchange data in both directions.

```
SimpleService := Connect >> (DataAB ||| DataBA)

Connect := a_conreq ; b_conind ; exit
DataAB  := a_datareq ; b_dataind ; DataAB
DataBA  := b_datareq ; a_dataind ; DataBA
```

The protocol derived using our transformations:

```
    SimpleProt := hide sync in Connect1 >> (DataAB1 ||| DataBA1)
                               |[sync]|
                               Connect2 >> (DataAB2 ||| DataBA2)
```

```
Connect1 := a_conreq ; sync!m1 ; sync!m2 ; exit
Connect2 := sync!m1 ; b_conind ; sync!m2 ; exit

DataAB1 := a_datareq ; sync!m3 ; sync!m4 ; DataAB1
DataAB2 := sync!m3 ; b_dataind ; sync!m4 ; DataAB2

DataBA1 := sync!m5 ; a_dataind ; sync!m6 ; DataBA1
DataBA2 := b_datareq ; sync!m5 ; sync!m6 ; DataBA2
```

Often a designer would like to change the messages $m1$, $m2$ etc. into messages with more meaningful names; this would make the specification more readable. We have not done this in order to clearly show the effect of the transformation as defined in section 3. Note how the structure of the service specification is preserved in the two components after transforming.

The above transformation could have been executed automatically by using the transformation tool *Cleaver* ([Bro92]). This is a prototype tool implementing the transformations in section 3. It makes use of an abstract syntax for LOTOS called *Common Representation* (CR); the CR was developed in the ESPRIT/LOTOSPHERE project [EevE92] for the integrated LOTOS tool environment LITE [CaSa91]. *Cleaver* was written in C with the help of a metatool, the term processor *Kimwitu* [vEB91]. Currently work is being undertaken in order to change *Cleaver* from a prototype into an industrially applicable tool that can be integrated into the LITE environment.

6 Conclusion

In this paper we have presented a compositional algorithm for the decomposition of processes in a process algebraic framework based on a partition of their action sets. Our presentation was given in terms of the specification language LOTOS, but the result carries over to other formalisms with similar combinators for parallel composition such as CSP and CIRCAL [Ho85, Mil85]. We have sketched the correctness proof of the algorithm and given the detailed proof for one of the LOTOS combinators, viz. parallel composition. The algorithm can be applied only in the context of a number of restrictions that measure in some sense the adequacy of the given partition as the basis for the distribution of functionality and in particular avoid the creation of so-called global choices. We have analysed the proposed restrictions and indicated how they may be circumnavigated if so desired. In particular we have indicated how the method can be adapted to achieve synchronization over a reliable asynchronous communication medium. We have given a simple example of its application for the derivation of a protocol from a simple service description. We have also included a short report on a tool, *Cleaver*, that has been implemented to support the application of this correctness preserving transformation on Basic LOTOS specifications.

As we reported in the introduction our work is an extension of that reported in [La90]. There global choices are handled by inserting a polling mechanism, but the algorithm is noncompositional and works on fully expanded specifications only, which greatly increases the number of synchronization actions between parallel components (before expansion). In [vES91] it is shown how the transformation from [La90] could be combined with another transformation, the *regrouping of parallel processes* from [Bol90], in order to formally derive a protocol. In [Bro92] it is demonstrated that the same derivation can be carried out by only using the improved transformation reported here.

It is interesting and informative to compare our current work to other related approaches. One of the first attempts to study a design transformation from a formal point of view can be found in [Gr88].

There the implementability of synchronization events over an asynchronous medium is studied as a LOTOS to LOTOS transformation. The transformation results are shown to be testing equivalent to their originals. An explicit semantic confluence condition is given instead of our static syntactic restrictions. Decomposition is only feasible if processes already have a (synchronous) parallel composition as their outermost operator, and the algorithm is therefore a distribution rather than a decomposition transformation.

In [KBK89] it is shown how to derive a protocol from a service by incorporating message passing over a reliable medium, using a kind of attribute grammar. It is more general in the sense that more than two protocol entities can be handled at a time. It is rather restricted, however, as it does not include the general parallel, enabling, and disabling combinators. The correctness of the transformation is not discussed. A proposal for handling global choice by inserting dummy interactions is suggested though not elaborated, but it does not seem to preserve any branching time semantics.

A similar approach can be found in [Mas89]. There a large subset of Full LOTOS is handled. Some restrictions are made that are quite similar to ours, e.g. no initial exits in a choice and no global choice. However, again no correctness proof is provided, probably due to the complexity of attribute grammar formalism that is used. In our opinion the proven correctness of is an essential ingredient in a formal design transformation.

A related and formally well-investigated problem is that of factorization of behaviours into parallel components in a process algebraic set-up. First results are due to Parrow [Par89] and Shields [Shi89] that study the solution of equations of the form $P||X \approx Q$ for given P and Q, where $||$ is some (generalized) parallel composition combinator. The difference with our approach is that we work, so to speak, from Q and specify the distribution of its actions to guide the decomposition, and do not suppose or require further knowledge about the desired substructure in the form of P.

Future work on the decomposition of functionality transformation includes its extension to Full LOTOS, i.e. the inclusion of data structures in communication and parameterization, and the consequent adaptation of the tool *Cleaver*. This should be relatively straightforward. The extension to full LOTOS suggests, however, the possibility of new decomposition criteria, such as the partition of the action labels (gates in LOTOS parlance) in combination with the type attributes that characterize the data that is communicated. This suggests the possibility of combining *gate-splitting* transformations [Bol92] with decomposition. Another aspect that still needs some attention is the incorporation of synchronization over asynchronous media in the proof for the compositional algorithm, as it is currenly only available for the [La90]-version. As we indicated earlier we expect no fundamental problems there, and hope exploit the benefits of the partial-order semantics for LOTOS as given in [La92] there.

Acknowledgements. Thanks to Bart Botma for comments and discussions. The anonymous referees are thanked for pointing out an error in the draft version.

References

[Bie89] Frank P.M. Biemans, A reference model for manufacturing planning and control, Doctoral Dissertation, University of Twente, October 1989.

[BoBr87] T. Bolognesi, E. Brinksma. *Introduction to the ISO specification language LOTOS.* Comp. Networks and ISDN Systems, 14 (1987), pp. 25-59.

[Bog] Kees Bogaards, *A methodology for the architectural design of open distributed systems,* doctoral dissertation, University of Twente, 1990.

[Bol90] Tommaso Bolognesi, A graphical composition theorem for networks of LOTOS processes, Tenth International Conference on Distributed Computing Systems, pp. 88-95, IEEE Computer Society Press, 1990.

[Bol92] Tommaso Bolognesi (ed.), *Catalogue of LOTOS Correctness Preserving Transformations*, LOTOSPHERE Final Deliverable Task 1.2, LO/WP1/T1.2/N0045/V03, The LOTOSPHERE Consortium, 1992.

[Bri92] Ed Brinksma, Formele analyse van gedistribueerde systemen, Lecture notes (in Dutch), University of Twente, 1992.

[Bro92] P. Broekroelofs. *Bipartitioning of LOTOS specifications.* Masters thesis, Memoranda Informatica 92-50, University of Twente, 1992.

[CaSa91] Maurizio Caneve, Elena Salvatori (eds.), LOTOSPHERE WP2, Lite User Manual, Ref. Lo/WP2/N0034/V06, November 1991.

[EevE92] Henk Eertink, Peter van Eijk (eds.), LOTOSPHERE WP2, The Lite common representation, Ref: Lo/WP2/T2.1/UIT/N0009/V9, 1992.

[vEB91] Peter van Eijk, Axel Belinfante, The term processor Kimwitu, Manual and Cookbook, version 2, University of Twente, December 1991.

[vES91] Peter van Eijk, Jeroen Schot, An exercise in protocol synthesis, Proc. FORTE 91 Conference on Formal Description Techniques, pp. 117-131, (North-Holland, 1992).

[Gr88] J.F. Groote. *Implementation of events in LOTOS-specifications.* M.Sc. Thesis, University of Twente, 1988.

[ISO9646] ISO, Information Processing Systems, Open Systems Interconnection, *OSI Conformance Testing Methodology and Framework*, IS 9646 (1991).

[Ho85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[ISO89] ISO, LOTOS - A formal description technique based on the temporal ordering of observational behaviour, IS 8807, 1989.

[KBK89] F.Khendek, G.v.Bochmann, C.Kant, New results on deriving protocol specifications from service specifications, Proc. SIGCOM '89 Symposium Communications Architecture & Protocols, Computer Communications Review, Vol.19, No.4, September 1989.

[La90] Rom Langerak, Decomposition of functionality: a correctness preserving LOTOS transformation, Proc. Tenth IFIP International Symposium on Protocol Specification, Testing and Verification, pp. 203-218, (North-Holland, 1990).

[La92] R.Langerak. *Bundle event structures: a non-interleaving semantics for LOTOS.* Fifth International Conference on Formal Description Techniques, October 1992, Lannion.

[Mi89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mil85] George J. Milne, Circal and the Representation of Communication, Concurrency, and Time, TOPLAS Volume 7 Number 2, 270–298, 1985.

[Mas89] Thierry Massart, A protocol synthesizer for LOTOS service specifications, Report IIHE /HELIOS-B - 89-101, Free University of Brussel, December 1989.

[DeHe 84] R. De Nicola, M. Hennessy, Testing Equivalences for Processes, Theoretical Computer Science 34, (1984), 83– 133.

[Par89] J. Parrow, Submodule Construction as Equation Solving in CCS, Theoretical Computer Science 68, 175–202 (1989).

[Sch91] Jeroen Schot (ed.), Mini-Mail Structures and Constructs, Lo/WP3/T3.3/UT/N0018/V02, The LOTOSPHERE Consortium, 1991.

[Sch92] Jeroen Schot, The role of architectural semantics in the formal approach of distributed systems design, Doctoral Dissertation, University of Twente, February 1992.

[Shi89] M.W. Shields, Implicit System Specification and the Interface Equation, The Computer Journal 32(5), 399–412 (1989).

[Tre92] Jan Tretmans, *A Formal Approach to Conformance Testing*, doctoral dissertation, University of Twente, 1992.

[VSvSB91] C.A. Vissers, G. Scollo, M.v. Sinderen, H. Brinksma, Specification styles in distributed systems design and verification, Theoretical Computer Science 89, pp. 179-206, 1991.