Structural Decomposition

Maria Hultström

Department of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 Kista, Sweden, Phone: +46 8 752 14 96, Fax: +46 8 751 17 93, Email: maria@it.kth.se

Abstract

We present an algorithm for the decomposition of a monolithic LOTOS process into two processes, whose composition by the parallel operator is observation equivalent to the original one. Repeated application permits the generation of more than two processes. The algorithm is defined on labelled transition systems and, hence, its application to any process algebra with a similar notion of parallel composition is straightforward. The transformation is based on a given partition of the actions into two subsets, corresponding to the action sets of the generated processes. The main interest of the transformation is that it makes explicit all the parallelism of the original behaviour. The *Structural Decomposition* has several applications, like the decomposition of functionality, modularization, synthesizing of protocols, decomposition of tests, etc.

Keyword Codes: F.4.3

Keywords: Mathematical Logic and Formal Languages, Formal Languages

1 Introduction

One approach to the verification of the correctness of a design step is to perform only transformations which have been verified in advance. These are commonly known as *correctness preserving transformations* (CPTs). The generally used correctness preserving transformations are classified in [2] based on their design goals. One of these CPTs is the *functional decomposition*, whose main idea is to transform a black box description into a white box description, making its internal structure visible. The main idea of the functional decomposition in this paper, hereafter called *structural decomposition*, is to partition a monolithic specification (i.e. a specification which does not explicitly mention parallelism) into two specifications. These specifications taken in parallel will realize the original monolith. The repeated application of the algorithm permits the decomposition of one specification into a arbitrary number of specifications. Decomposition is particularly important in the design of distributed systems since a system can often be given a high-level description which ignores distribution (e.g. the service specification of a protocol); this specification must then be refined when the system is implemented.

The *Structural Decomposition* is related to other transformations described in the literature, such as, for example, [3, 4, 5, 6, 7, 8, 9, 10]. The main contribution of the *Structural*

Decomposition with respects to the former algorithms, is that the potential parallelism executing in a specification is made explicit. The former algorithms achieve a very low degree of parallelism, except in [3] and partly in [4]. The earlier work on the *Inverse Expansion* [3, 4] was based on the same idea and maintained the parallelism in the specification. However, it only worked for adjusted monolithic LOTOS specifications and could not deal with non-determinism [4] nor recursion [3]. The *Structural Decomposition* is a continuation of the work presented in [3] and is applicable on a general monolithic LOTOS specification, in contrast to [3, 4]. Moreover, it has overcome the problems of non-determinism and recursion.

The decomposition is based on a given partition of the actions in the specification into two classes where each generated sub-process performs actions belonging to its class. The idea is to introduce synchronization events which are hidden from the outside so that observation equivalence is maintained. This has been used earlier in related work, for example in [5, 4, 7]. The difference of our approach is that instead of introducing synchronization events in all places, we isolate zones which interleave with each other and thus, only need to introduce synchronization events to separate these zones from each other. In this way we are able to make the parallelism of the specification explicit.

The structural decomposition has potential for many practical applications, for example: the decomposition of the specification in parts that adjust to the resources in the system, the derivation of the specification of a protocol from its service specification, modularization, the decomposition of a global tester in an *upper* and *lower* test, etc.

2 Definitions

Definition 2.1 (Labelled transition system) *A labelled transition system LTS is defined as a 4-tuple $LTS = \langle S, \mathcal{L}, T, s_0 \rangle$ where:*

- S is a non-empty set, called the set of states
- $\mathcal{L}$ is the alphabet not containing the internal event $\mathbf{i}$, called the set of observable actions
- $T \subseteq S \times (\mathcal{L} \cup \{\mathbf{i}\}) \times S$, where $(s, \alpha, s') \in T$ is written $s \xrightarrow{\alpha} s'$ when T is clear from context, called the transition relation.
- $s_0 \in S$ is the initial state of the system.

Notations:

If $s, s', s_1, s_2 \dots$ are states of a labelled transition system then

$s \xrightarrow{a_1, a_2, \dots, a_n} s'$ will abbreviate $\exists s_1, \dots, s_{n-1} : s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s'$

$s \not\xrightarrow{\alpha}$ will abbreviate $\neg \exists s' \text{ such as } s \xrightarrow{\alpha} s' \text{ for any } \alpha \in \mathcal{L}$.

A LTS denoted LTS_* will abbreviate the 4-tuple $\langle S_*, \mathcal{L}_*, T_*, s_{0,*} \rangle$ where $*$ could be any index.

In the following we will represent a LTS as a labelled directed tree, where the vertices correspond to the states and the labelled arcs represent the transition relation. The initial state is the top-most vertex and the direction is downwards if not indicated otherwise.

Vertices which are represented more than once in the same tree are denoted with the same index.

Definition 2.2 (Reach(LTS,s)) We define the set of reachable states from a given state in a LTS as: $Reach(LTS, s) = \{s' \mid s \xrightarrow{\alpha} s' \wedge \alpha \in \mathcal{L}^*\}$

Definition 2.3 (Isomorphism) Two labelled transition systems LTS_1 and LTS_2 are isomorphic (denoted $LTS_1 \equiv LTS_2$) iff there exists a mapping $f_s : S_1 \rightarrow S_2$ such as :

- f_s is a bijection
- for any $p_1, p_2 \in S_1$ and any $\alpha \in \mathcal{L}_1 \cup \mathcal{L}_2 \cup \{\mathbf{i}\}$
 $p_1 \xrightarrow{\alpha} p_2$ iff $f_s(p_1) \xrightarrow{\alpha} f_s(p_2)$
- $f_s(s_{0_1}) = s_{0_2}$

Definition 2.4 (Sub-LTS) A labelled transition system LTS_S is a sub-LTS of another labelled transition system LTS_B iff there exists a mapping $f_{sub} : S_S \rightarrow S_B$ such as :

- f_{sub} is an injection
- for any $p_1, p_2 \in S_S$ and any $\alpha \in \mathcal{L}_S \cup \{\mathbf{i}\}$
if $p_1 \xrightarrow{\alpha} p_2$ then $f_{sub}(p_1) \xrightarrow{\alpha} f_{sub}(p_2)$
- $f_{sub}(s_{0_S}) = s_{0_B}$

Example: A is a Sub-LTS of B.

Definition 2.5 (Parallel composition on LTS) We define the labelled transition system $LTS_1 \parallel [\mathcal{L}_1 \cap \mathcal{L}_2] LTS_2$ as $\langle S_{\parallel}, \mathcal{L}_{\parallel}, T_{\parallel}, s_{\parallel} \rangle$ where:

- $S_{\parallel} = S_1 \times S_2$
- $\mathcal{L}_{\parallel} = \mathcal{L}_1 \cup \mathcal{L}_2$
- $(p_1, p_2) \xrightarrow{\alpha} (p'_1, p_2) \in T_{\parallel}$ if $p_1 \xrightarrow{\alpha} p'_1 \in T_1 \wedge \alpha \in \mathcal{L}_1 \cup \{\mathbf{i}\} - \mathcal{L}_2$
 $(p_1, p_2) \xrightarrow{\alpha} (p_1, p'_2) \in T_{\parallel}$ if $p_2 \xrightarrow{\alpha} p'_2 \in T_2 \wedge \alpha \in \mathcal{L}_2 \cup \{\mathbf{i}\} - \mathcal{L}_1$
 $(p_1, p_2) \xrightarrow{\alpha} (p'_1, p'_2) \in T_{\parallel}$ if $p_1 \xrightarrow{\alpha} p'_1 \in T_1 \wedge p_2 \xrightarrow{\alpha} p'_2 \in T_2 \wedge \alpha \in \mathcal{L}_2 \cap \mathcal{L}_1$
- $s_{\parallel} = (s_{0_1}, s_{0_2})$

Definition 2.6 (Interleaved composition on LTS) We define the labelled transition system $LTS_1 \parallel\parallel LTS_2$ as $LTS_1 \parallel [\emptyset] LTS_2$.

Definition 2.7 ($LTS_1 \stackrel{rec}{\equiv} LTS_2$) $LTS_1 \stackrel{rec}{\equiv} LTS_2$ iff $\exists p_{stop} \in S_1$ such as $p_{stop} \nrightarrow$ and there exists a mapping $f_s : (S_1 - \{p_{stop}\}) \rightarrow S_2$ such as :

- f_s is a bijection
- for any $p_1 \in (S_1 - \{p_{stop}\} - \{p \mid p \xrightarrow{\alpha} p_{stop}\})$ and any $p_2 \in S_1 - \{p_{stop}\}$ and any $\alpha \in \mathcal{L}_1 \cup \mathcal{L}_2 \cup \{\mathbf{i}\}$
 $p_1 \xrightarrow{\alpha} p_2$ iff $f_s(p_1) \xrightarrow{\alpha} f_s(p_2)$

- $f_s(s_{0_1}) = s_{0_2}$

and there exists a state $p_{rec} \in S_2$ such as:

- for any $p_1 \in S_1 - \{p_{stop}\}$ and any $\alpha \in \mathcal{L}_1 \cup \mathcal{L}_2 \cup \{i\}$
 $p_1 \xrightarrow{\alpha} p_{stop}$ iff $f_s(p_1) \xrightarrow{\alpha} p_{rec}$

Definition 2.8 (Interleaved sub-LTS) LTS_{IS} is an interleaved sub-LTS of LTS_B iff:

- LTS_{IS} is a sub-LTS of LTS_B
- there exist a LTS_1 and a LTS_2 such as $((LTS_1 ||| LTS_2 \equiv LTS_{IS})$ or $(LTS_1 ||| LTS_2 \stackrel{rec}{\equiv} LTS_{IS}))$ and $T_1 \neq \emptyset \wedge T_2 \neq \emptyset$

Example: A is an interleaved sub-LTS of B and A' is an interleaved sub-LTS of B' .

3 LOTOS

In this paper we only consider Basic LOTOS [1]. Moreover, as the behaviours are specified in a monolithic style [11] only those operators needed for this style, i. e. action prefix, stop, alternative, and the parallel operator and hiding operator will be used. The application of the expansion theorem [12] to a non-monolithic behaviour permits the decomposition of any Basic LOTOS behaviour.

constant operator	stop	$\rightarrow P$
action prefix operator	$;$	$L \cup \{i\} \text{ x } P \rightarrow P$
alternative operator	$[]$	$P \text{ x } P \rightarrow P$
parallel operator	$[]$	$P \text{ x } P(L) \text{ x } P \rightarrow P$
hiding operator	hide in	$P(L) \times P \rightarrow P$

where P is the set of processes, L is the set of the visible actions, and $P(L)$ is the powerset of L . The parallel operator will be renamed to $|||$ for the case of pure interleaving.

A Basic LOTOS behaviour can be interpreted as a labelled transition system, via the axioms and inference rules of the language [13]. The generation of a LOTOS behaviour from a LTS, where the behaviour includes action prefix, stop, alternative and process definitions, is straightforward. Thus, we may use LTSes and LOTOS behaviour interchangeably and use the expressions *sub-behaviour* and *interleaved sub-behaviour*.

4 Description of the transformation

The transformation is limited to finitely branching, concrete behaviours. This means that in each state the number of choices is finite and the behaviours do not contain internal actions. To deal with internal actions each internal action is assigned to one of the given partitions, then internal actions are just treated as ordinary actions.

The *Structural Decomposition* presented in this article can be formulated as: Given $B[A_0]$, labelsets A_1 and A_2 find a labelset A_I and behaviours B_1, B_2 such as:

$$B[A_0] \approx_{obs} \text{hide } A_I \text{ in } B_1[A_1 \cup A_I] \parallel [A_I] B_2[A_2 \cup A_I] \quad (1)$$

where B is the behaviour to be decomposed and A_0 the actions used within B . The sets A_1 and A_2 partition A_0 , B_1 and B_2 are the behaviours we wish to generate and A_I is the set of internal actions introduced during the decomposition. The relation we want to maintain is observational equivalence, $\approx_{obs}$. Moreover, $A_1 \cup A_2 = A_0$ and A_1, A_2 and A_I are pairwise disjoint.

The transformation is performed by two algorithms. The first one, *Ext*, will extend the original behaviour by including new events. When these events are hidden from the outside the obtained behaviour will be observational equivalent with the original one. The second algorithm, *Rest*, will extract the behaviour of B_1 and B_2 , mentioned above. B_1 and B_2 will be strongly bisimulation equivalent to the extended behaviour when synchronizing on the introduced events. Both algorithms are conducted on labelled transition systems.

5 Extension of the Behaviour

In this section we will present a transformation of the given behaviour to an observational equivalent behaviour which may be decomposed into two behaviours. These behaviours taken in parallel maintain the observation equivalence with the initial behaviour when all synchronization events are hidden. The synchronization events are introduced during the transformation.

In the decomposition we keep the potential parallelism (here interleaving) of the original behaviour, by isolating zones which could interleave with each other and do not need to synchronize. The zones of interleaving are separated from each other by a *synchronization loop*. A synchronization loop consists of two states with synchronization transitions to each other. A zone of interleaving consists of one behaviour with actions belonging to A_1 and another behaviour with actions belonging to A_2 which interleave. When the whole behaviour consists of one zone of interleaving, with respect to the action sets A_1 and A_2 , no new events will be introduced at all.

The only change made to the original behaviour is an extension with synchronization loops at certain places.

5.1 Finding Interleaved sub-behaviours

The solution to equation 1 on page 5, where B_1, B_2 and A_I are the unknowns, is not unique. There exist an infinite number of solutions satisfying observational equivalence

due to the introduced set A_I . We want to minimize the size of A_I and the places where the synchronization events are introduced. We accomplish this by finding zones of interleaving based on the given partition of A_0 , and only introduce synchronization events to separate these zones.

In a given LTS, with partitions A_1 and A_2 of the label set, there may exist many zones of interleaving. We distinguish between two types:

- An interleaved sub-LTS with respects to the partition, is an interleaved sub-LTS with LTS_1 containing actions belonging to A_1 and LTS_2 contain actions belonging to A_2 .
- A sub-LTS with respect to the partition, is a sub-LTS with labels in one of the sets A_1 and A_2 . This correspond to a zone of interleaving where one of the behaviours is equal to **stop**.

Hereafter, an interleaved sub-LTS will denote an interleaved sub-LTS with respect to the labelsets A_1 and A_2 , and similarly for a sub-LTS.

As the purpose of the decomposition is to make the existing parallelism explicit, an interleaved sub-LTS is always preferred over a sub-LTS. Which interleaved sub-LTS to choose should probably be a decision for the designer. In most cases, he wants to minimize the introduction of internal events and, thus, selects an interleaved sub-behaviour which is maximal (i.e., one which has the maximal number of states). In some cases, however, the designer may have preferences of actions which should interleave, and to find an interleaved sub-behaviour containing these events may be more important than to select a maximal one. The selected interleaved sub-LTS need not to be unique, there may be other interleaved sub-LTSes which are maximal or contain the preferable actions.

When no interleaved sub-LTSes exist, we must take the sub-LTSes into account. The selected one will be unique with respect to the selected label-set. We will chose one which is maximal. This is defined as (the restriction operator of CCS [12]):

Definition 5.1 ($LTS_{B \setminus A}$) *The labelled transition system $LTS_{B \setminus A}$ representing the labelled transition system LTS_B restricted to events in $\mathcal{L}_B - A$ is defined as:*

Given LTS_B and the restriction set A , $LTS_{B \setminus A} = \langle S_{B \setminus A}, \mathcal{L}_{B \setminus A}, T_{B \setminus A}, s_{B \setminus A_0} \rangle$ where :

$$S_{B \setminus A} = \{s \mid s_{B_0} \xrightarrow{\alpha} s \wedge \alpha \in \mathcal{L}_B - A\}$$

$$\mathcal{L}_{B \setminus A} = \mathcal{L}_B - A$$

$$T_{B \setminus A} = \{s \xrightarrow{a} s' \mid s \in S_{B \setminus A} \wedge s \xrightarrow{a} s' \in T_B \wedge a \in \mathcal{L}_{B \setminus A}\}$$

$$s_{B \setminus A_0} = s_{0_B}$$

The maximal sub-LTS of LTS_B with respect to the labelset A_1 is obtained by $LTS_{B \setminus A_2}$, and the maximal sub-LTS of LTS_B with respect to the labelset A_2 is obtained by $LTS_{B \setminus A_1}$.

One simple and not very efficient algorithm for finding interleaved sub-LTSes could be described as (a full definition can be found in [14]):

Given a labelled transition system LTS_B and the partitions A_1 and A_2 we want to find sub-LTSes LTS_1 and LTS_2 such as $\mathcal{L}_1 \subseteq A_1$ and $\mathcal{L}_2 \subseteq A_2$ and $LTS_1 \parallel LTS_2$ is an interleaved sub-LTS of LTS_B . Due to the properties of interleaving all possible LTS_1 (LTS_2) will be sub-LTSes of $LTS_{B \setminus A_2}$ ($LTS_{B \setminus A_1}$). We generate all combinations of LTS_1 and LTS_2 and check if their composition actually is an interleaved sub-LTS of LTS_B , as defined in 2.8.

We leave for further research the definition of an efficient algorithm for finding interleaved sub-LTSes. An efficient algorithm would use the fact that if the transition $s_0 \xrightarrow{a} s_2 \in T_1$ and the transition $s_0 \xrightarrow{b} s_4 \in T_2$ is not an interleaved sub-LTS of LTS_B then no sub-LTSes containing the transitions $s_0 \xrightarrow{a} s_2$ and $s_0 \xrightarrow{b} s_4$ could fulfil the demands. Note that even with an efficient algorithm the whole state-space has to be considered and we are therefor limited to finite-state behaviours. An interleaved sub-LTS of LTS_B exists iff $\exists \alpha \in A_1, \exists \beta \in A_2 \mid s_0 \xrightarrow{\alpha} p_i \in T_B \ \& \ s_0 \xrightarrow{\beta} p_j \in T_B \ \& \ p_i \xrightarrow{\beta} p'_i \in T_B \ \& \ p_j \xrightarrow{\alpha} p'_j \in T_B \ \& \ p'_i \equiv p'_j$

Example: Given the behaviour B below and $A_1 = \{a, a_2\}$ and $A_2 = \{b, c, c_2\}$.

then, for clarity represented as LOTOS behaviours, there exists seven interleaved subbehaviours ($B_{1i} \parallel B_{2i}$):

$$(B_{1_1} = a; \text{stop}, B_{2_1} = b; \text{stop}), (B_{1_2} = a_2; \text{stop}, B_{2_2} = b; \text{stop}), (B_{1_3} = a; \text{stop}, B_{2_3} = c; \text{stop}), (B_{1_4} = a; \text{stop}, B_{2_4} = c; c_2; \text{stop}), (B_{1_5} = a; \text{stop}, B_{2_5} = b; \text{stop} \sqcap c; \text{stop}), (B_{1_6} = a; \text{stop}, B_{2_6} = b; \text{stop} \sqcap c; c_2; \text{stop}), (B_{1_7} = a; \text{stop} \sqcap a_2; \text{stop}, B_{2_7} = b; \text{stop})$$

Selecting one in which the number of states is maximal give us: B_{1_6} and B_{2_6} . The maximal sub-behaviours with respect to A_1 (A_2) is B_{1_7} (B_{2_6}).

5.2 Insertion of synchronization events

The algorithm *Ext*, defined below, will transform a LTS by adding synchronization loops to isolate zones of interleaving. We will choose one interleaved zone, LTS_{IZ} , from the root, due to certain criteria (see above). For all states p_i of the general LTS such that p_i is included in the interleaved zone, we will keep the transitions (and the states) leading to a state which is in the interleaved zone. The transitions not included in the interleaved zone will be preceded by a unique synchronization loop. Then, each state preceded by a synchronization loop will serve as the initial state for the new application of the algorithm.

Example: There exist one interleaved sub-LTS from the root and one sub-LTS. $A_1 = \{a, b\}$ and $A_2 = \{c\}$. B' will be obtained by the algorithm *Ext*.

Ext will take the LTS of the general behaviour to be extended, the two gate sets A_1 and A_2 and the initial state of the LTS to keep track of the initial state in the recursion. In a list *List* we will include all states already considered by the algorithm. If a state q has been considered (i.e. it was a member of an interleaved zone treated earlier), then a transition leading to this state ($p \xrightarrow{\alpha} q$) has to be followed by a synchronization loop ($p \xrightarrow{\alpha} P \xrightarrow{k_p} q, q \xrightarrow{k'_p} P$) so that it is not in conflict with the interleaved zone which included the state, q . When finding interleaved zones the states already considered are treated as a state equal to *stop*.

$\cup_{LTS}$ denotes the componentwise union of two LTSes. $\cup_{LTS_i}$ is the repeated application of $\cup_{LTS}$.

$Ext(LTS_B, A_1, A_2, s) =$

Let

$LTS_{IZ} = \langle S_{IZ}, \mathcal{L}_{IZ}, T_{IZ}, s_{IZ} \rangle$ be an interleaved zone of LTS_B

$\{p_1, \dots, p_n\} = S_{IZ}$

List be a global list including all states which already been delt with

Considered(p_i) be true if $p_i \in List$

Out(p_i) be true if $\neg Considered(p_i) \wedge \exists (p_i \xrightarrow{\alpha} q) \in T - T_{IZ}$

$Q_i, Reach_i$ be defined for every i such as *Out*(p_i)

$Q_1, \dots, Q_k$ be distinct $\wedge Q_i \notin S_B$

$Reach_i$ be $\cup_q Reach(LTS_B, q) \forall q : (p_i \xrightarrow{\alpha} q) \in T - T_{IZ}$, *Reach* defined on page 3

P_i be defined for every i such as *Considered*(p_i)

$P_1, \dots, P_l$ be distinct $\wedge P_i \notin S_B$

$\langle S' = S_{IZ} \cup \{P_1, \dots, P_l, Q_1, \dots, Q_k\}$

$\mathcal{L}' = \mathcal{L} \cup \{k_1, \dots, k_k, k'_1, \dots, k'_k\} \cup \{k_{p1}, \dots, k_{pl}, k'_{p1}, \dots, k'_{pl}\}$

where $k_1, k'_1, \dots, k_k, k'_k, k_{p1}, k'_{p1}, \dots, k_{pl}, k'_{pl} \notin \mathcal{L}$ and distinct

$T' = T_{IS} \cup \{Q_i \xrightarrow{k_i} p'_i \mid Out(p_i)\} \cup \{p'_i \xrightarrow{k'_i} Q_i \mid Out(p_i)\}$

$\cup \{P_i \xrightarrow{k_{pi}} p'_i \mid Considered(p_i)\} \cup \{p'_i \xrightarrow{k'_{pi}} Q_i \mid Considered(p_i)\}$

$s'_0 = s \rangle$

$\cup_{LTS}$

$\cup_{LTS_i} Ext(\langle \{Q_i\} \cup Reach_i, \mathcal{L}, \{Q_i \xrightarrow{\alpha} q \mid \forall q, \alpha : (p_i \xrightarrow{\alpha} q) \in T - T_{IS}\} \cup \{q \xrightarrow{\alpha} q' \mid (q \xrightarrow{\alpha} q') \in T, q, q' \in Reach_i, Q_i\}, A_1, A_2, s)$

The algorithm *Ext* as defined above does not cover the case when an interleaved sub-LTS is recursive in its “final” state. *Considered* of this state should be true, and by the definition of *Ext* a synchronization loop is included. This is done to ensure the correct ordering in the decomposition.

Example: $A_1 = \{a\}$ and $A_2 = \{b\}$. The state s_0 represents the recursion.

The addition of synchronization loops will be a candidate for optimizations, treated in next subsection.

5.3 Optimizations

The extension of the original behaviour done by *Ext* may be optimized in two ways.

1. By reducing the introduced transitions.

- 1(i) In a state s which only is able to perform a synchronization, the synchronization loop can be replaced by a synchronization transition.

Example: B may be reduced to B' . $A_1 = \{a, a_1\}$ and $A_2 = \{b\}$.

- 1(ii) The synchronization loop introduced when *Considered*(p) is true is not always necessary. For example, when a sub-LTS with respects to actions in A_1 has a state which has been considered and this state forms part of a sub-LTS with respects to A_1 no synchronizations are needed.

2. By reducing the amount of introduced events. In some cases one label may be used with several synchronization transitions. Two labels k_i and k_j may receive the same label k_{ij} in two ways, where s represents any state in the extended transition system:

- 2(i) If $s \xrightarrow{k_i \beta k_j} s'$, $\beta \in (A_1 \cup A_2 \cup A_I)^*$ k_i and k_j may receive the same label k_{ij} , and likewise k'_i , k'_j may receive the same label k'_{ij} .

- 2(ii) If $s \xrightarrow{k_i} s_i$ and $s \xrightarrow{\alpha k_j} s_j$, $\alpha \in A_1^*$ and s_i, s_j are isomorphic with respect to the actions belonging to A_2 until a stop or another action $\in (A_1 \cup A_I)$ is found, then k_i and k_j may receive the same label k_{ij} and similarly for k'_i and k'_j . This can also be applied when A_1 and A_2 change place. If this optimization is used, then the decomposed behaviour should also be optimized. In every state which may perform more than one k_i transition these should be unified.

Example: In the figures the k'_i transitions are not present as they do not clarify anything for this example. $A_1 = \{a, a_2\}$, $A_2 = \{b, b_2\}$ and $A_I = \{k_1, k_2, k_3, k_4, k_5\}$. The optimized A_I will contain $\{k_{125}, k_{34}\}$, where $\mathcal{Z}(i)$ has been applied for k_1, k_5 and $\mathcal{Z}(ii)$ for the others.

6 Decomposition

The decomposition will be applied on the extended LTS with the introduced synchronization events. We want to decompose the behaviour in two behaviours B_1 and B_2 such that $\text{hide } A_I \text{ in } B_1[A_1 \cup A_I] \parallel [A_I] B_2[A_2 \cup A_I]$ is observational equivalent with the given behaviour $B[A_1 \cup A_2]$. We know by definition [12] that $B[A_1 \cup A_2]$ is observational equivalent to $\text{hide } A_I \text{ in } B'[A_1 \cup A_2 \cup A_I]$, where B' was obtained from Ext .

The behaviour B' may in each state offer actions belonging to A_1 and A_2 and synchronization actions. The general idea is to keep all sequences from the initial state belonging to $A_1 \cup A_I$ (or $A_2 \cup A_I$) and remove all A_2 sequences until a synchronization event is found and then apply the algorithm on the state reached by the synchronization event. Due to the properties of B' all synchronization events will be contained in a synchronization loop, i.e. if $s \xrightarrow{c} s'$ and $c \in A_I$ then $s' \xrightarrow{c'} s, c' \in A_I$.

The transition relation for B_1 will be obtained by $Rest(LTS_{B'}, A_2, A_I)$ and the transition relation for B_2 by $Rest(LTS_{B'}, A_1, A_I)$.

$$\begin{aligned}
 Rest \quad (LTS_B, A, A_I) = & T_{B \setminus (A \cup A_I)} \quad \text{where } T_{B \setminus (A \cup A_I)} \in LTS_{B \setminus (A \cup A_I)} \\
 \cup & \forall s \in S_{B \setminus (A \cup A_I)} \quad \text{where } S_{B \setminus (A \cup A_I)} \in LTS_{B \setminus (A \cup A_I)} \\
 & \cup_s Sync(LTS_B[s/s_{0_B}], A, A_I, s) \\
 Sync \quad (LTS_B, A, A_I, s_{name}) = & \\
 & \forall s \in S_B : (s_{0_B} \xrightarrow{\beta} s) \in T_B \wedge \beta \in A \\
 & \cup_s Sync(LTS_B[s/s_{0_B}], A, A_I, s_{name}) \\
 \cup & \\
 & \forall s' \in S_B : (s_{0_B} \xrightarrow{c} s') \in T_B \wedge c \in A_I \\
 & \cup_{s'} \{(s_{name} \xrightarrow{c'} s'), (s' \xrightarrow{c'} s_{name})\} \cup Rest(LTS_B[s'/s_{0_B}], A, A_I)
 \end{aligned}$$

where:

$LTS_B[s/s_{0_B}]$ denotes LTS_B with initial state equal to s , i.e. $\langle S_B, \mathcal{L}_B, T_B, s \rangle$, and $LTS_{B \setminus A}$ defined in 5.1 on page 6.

Rest extracts the transitions of LTS_B that will be directly reachable from the state s_0 for the chosen set (A_1 or A_2). *Sync* extracts the synchronization loops which may be present directly or after interleaving actions of the other set which should be reachable from the state s_{name} . *Rest* is then applied again. For recursive behaviours, other than synchronization loops, *Rest* will not terminate. We need to add a list containing states which already have been dealt with. If a state is found which is included in the list, *Rest* should stop.

The labelled transition systems for B_1 denoted LTS_{B_1} is:

$B_1 = \langle \{s'_{0_B}\} \cup \{s' \mid s \xrightarrow{a} s' \in Rest(LTS_{B'}, A_2, A_1)\}, A_1, Rest(LTS_{B'}, A_2, A_1), s'_{0_B} \rangle$ and correspondingly for the labelled transition system B_2 , denoted LTS_{B_2} , where the subsets A_1 and A_2 change place.

Example: $A_1 = \{a_1, a\}$ and $A_2 = \{b, b_1\}$. For B_1 , *Sync* extracts the k_i transitions and for B_2 , *Sync* will not extract anything.

Proposition: Let $LTS_{B'}$ be the labelled transition system obtained from $Ext(LTS_B, A_1, A_2, s_{B_0})$ and LTS_{B_1} , LTS_{B_2} be as defined above, then:

$LTS_{B'} \sim LTS_{B_1} \parallel [A_1] \parallel LTS_{B_2}$, where $\sim$ denotes strong bisimulation.

Proof: The proof of the proposition can be found in [14].

7 Example

As an example to illustrate the transformation we have developed a simple access protocol. An user wants to access a system for a transaction. The system permits only one user at a time. The requirements are as follows:

- The system will indicate if it is free to perform a transaction, *Free*, or if it cannot do anything, *Down*.
- The user first has to specify his identity, *Id*, and then which transaction he wants to make, *Trans*.
- The system indicates if it is free, *Free*, to serve any user – the user then sends his *Id* and *Trans*. The user may also start to initiate the communication by *Id* and *Trans*. When and if the system is free will be indicated to the user.
- When a transaction is performed the system acknowledgements this by *Ack*.

- After a transaction is completed and an acknowledge is received, the user and the system will indicate the disconnection from each other, in either order, by *Rel-U* and *Rel-S*.
- After a two-way disconnection, the interaction may start with a new user. Also, after a message that the system is *Down*, the interaction may start from the beginning.

The given LOTOS specification of the system is as follows:

```

specification B[A0] : noexit
behaviour System[A0] where
process System[A0] : noexit :=
    Free; Id; Trans; Ack; Release[A0]
  [] Down; System[A0]
  [] Id; (
    Down; System[A0]
    [] Free; Trans; Ack; Release[A0]
    [] Trans; (
      Down; System[A0]
      [] Free; Ack; Release[A0]
    )
  )
where
process Release[A0] : noexit :=
    Rel-U; Rel-S; System[A0]
  [] Rel-S; Rel-U; System[A0]
endproc
endproc
endspec

```

where A_0 is the gate-set $\{Free, Id, Trans, Ack, Down, Rel-S, Rel-U\}$.

We want to partition this system into two units, one that controls the access, an *Access Unit*, and one that controls the resources, a *Resource Allocator*. The communication between these two units is internal. The given partition is $A_1 = \{Id, Trans, Rel-S\}$ and $A_2 = \{Free, Ack, Rel-U, Down\}$.

We first apply the algorithm *Ext*, obtaining the extended behaviour with synchronization loops, considering the optimizations that may be done. We obtain two new different events, k_0 and k_1 . In here, they will be called *sync1* and *sync2* for clarity. Thus, $A_I = \{sync1, sync2\}$. Then applying our algorithm *Rest* we obtain the behaviours for the *Access Unit* and the *Resource Allocator*.

```

process AU[  $A_1 \cup A_I$  ] : noexit :=
  sync1; ( sync1; AU[  $A_1 \cup A_I$  ]
    [ ] sync2; AU[  $A_1 \cup A_I$  ] )
  [ ] Id; Transaction[  $A_1 \cup A_I$  ]
where
  process Transaction[  $A_1 \cup A_I$  ] : noexit :=
    sync1; ( sync1; Transaction[  $A_1 \cup A_I$  ]
      [ ] sync2; AU[  $A_1 \cup A_I$  ] )
    [ ] Trans; Release[  $A_1 \cup A_I$  ]
  where
    process Release[  $A_1 \cup A_I$  ] : noexit :=
      sync1; ( sync1; Release[  $A_1 \cup A_I$  ]
        [ ] sync2; AU[  $A_1 \cup A_I$  ] )
      [ ] sync2; sync2; Rel-S;
      sync2; AU[  $A_1 \cup A_I$  ]
    endproc
  endproc
endproc

process RA[  $A_2 \cup A_I$  ] : noexit :=
  sync1; ( sync1; RA[  $A_2 \cup A_I$  ]
    [ ] Down; sync2; RA[  $A_2 \cup A_I$  ] )
  [ ] Free; sync2; Ack; sync2; Rel-U;
  sync2; RA[  $A_2 \cup A_I$  ]
endproc

```

In this example there were two zones of interleaved sub-behaviours, represented as LOTOS behaviours (*Id*; *Trans* ||| *Free*) and (*Rel* – *U* ||| *Rel* – *S*). In each case, these were the only interleaved sub-behaviours which existed.

Analyzing the behaviours obtained: The Access Unit, *A.U.*, may receive the identity and the wanted transaction from the user, but in each state it could participate in a synchronization with the Resource Allocator to permit the important message that the system is down. The Resource Allocator, *R.A.*, may indicate when it is free or when something is wrong ensure a synchronization with the *A.U.*, indicated to the user by *Down*. After the *R.A.* has sent the message *Down*, *A.U.* and *R.A.* have to synchronize to be able to start from the beginning. When the Resource Allocator is free and the user has sent his *Id* and the type of transaction, *A.U.* and *R.A.* synchronize to enable the *R.A.* to allocate a resource for performing the transaction. Then, *A.U.* has to wait for the *R.A.* to send an acknowledgement to the user, before allowing the user to indicate

that he wants to release the system. The *R.A.* indicates that he can release the user, and *A.U.* and *R.A.* have to synchronize when ready to start from the beginning.

The specification given by:

$$\begin{aligned} & \text{hide } \textit{sync1}, \textit{sync2} \text{ in} \\ & \quad \textit{AU}[\textit{Id}, \textit{Trans}, \textit{Rel} - \textit{S}, \textit{sync1}, \textit{sync2}] \\ & \quad \quad || \textit{sync1}, \textit{sync2} || \\ & \quad \textit{RA}[\textit{Free}, \textit{Down}, \textit{Ack}, \textit{Rel} - \textit{U}, \textit{sync1}, \textit{sync2}] \end{aligned}$$

is observational equivalent with the specification B.

The transformation presented in this paper will be implemented in the tool LOLA [15], to permit application on real-size examples.

8 Conclusions and related work

In this paper we have presented an algorithm for the decomposition of a general monolithic LOTOS behaviour into two sub-behaviours, which composed in parallel maintain observational equivalence with the original one. Repeated application of the transformation permits the generation of more than two sub-behaviours. The algorithm is based on the notion of labelled transition systems and the application to any other formalism with similar operator for the parallel composition is straightforward. The transformation is based on a partition of the action set into two subsets, which should be given by the designer. We have presented how to isolate zones of interleaving and we only need to introduce synchronizations to separate these zones. In this way we make the parallelism of the original specification explicit, which is the major result of this paper.

We have also presented a simple example for the illustration of our transformation. We will define a more efficient algorithm for finding interleaved sub-behaviours and implement the transformation in the tool LOLA [15], enabling the evaluation of real-size examples.

This work can be extended in several directions. First of all, the transformation should permit visible synchronization events between the two sub-behaviours. In this way, not only will the parallelism obtained from interleaving be made explicit, but also the synchronizations in combination with interleaving actions will form isolated zones. This extension is straightforward if no non-determinism is present. The algorithms presented in this paper could be joined with the algorithm for Visible Communication in [3] to perform the desired result. When treating non-determinism, the extension becomes more complex, as the distribution of the non-determinism between the two obtained sub-behaviours is not evident. Adopting the simplest solution, that one sub-behavior will be deterministic with respect to the synchronization actions, and the other one will contain all the non-determinism, leads to an extension that is easy to add. Another obvious extension is to consider LOTOS specifications which include data types.

As mentioned in the introduction, our work is a continuation of [3], where we presented algorithms for the decomposition of an adjusted monolithic LOTOS behaviour which could be expressed without introducing synchronizations. These algorithms were a further elaboration of those presented in [5]. Except in [3, 4, 5], there do not exist to our knowledge

any other work on the functional decomposition with the point of view presented in this article. There do exist similar works which do not maintain the existing parallelism of a monolith:

- [5] includes, except a first approach to the algorithms mentioned above, an algorithm which works on a general monolithic LOTOS specification by introducing internal synchronization events between, more or less, every transition in the original behaviour. Hence, the executing parallelism in the specification is lost.
- In [4] one of the algorithms of [3] has been further elaborated and may deal with recursive behaviours. It also contains an elaboration of the algorithm in [5] mentioned above, where optimizations of the introduction of internal events is defined.
- [7] contains algorithms of different complexity for decomposing a monolithic LOTOS process into two subprocesses. The interaction between the two subprocesses is either synchronous or asynchronous communication. These algorithms maintain the observational equivalence, but do not keep the parallelism.
- In [6], an algorithm is presented for obtaining an arbitrary number of protocol entities from a given service specification via introducing synchronization events. The overall behaviour is restricted to choices between events of the same protocol entity and the present parallelism is not kept.
- [8] is an extension of [7] where the decomposition of a not expanded, i. e. non-monolithic, LOTOS process is treated. The relation maintained is observation congruence. As the algorithm takes the structural information of the behaviour into account, some parallelism is kept, but for an expanded LOTOS process or a part which has been expanded, the parallelism is not made explicit. The overall behaviour is restricted to choices between events belonging to the same partition.

Another type of transformations is based on protocol synthesizing. The idea here is to find one missing submodule specification, which together with the already specified submodules preserve some equivalence relation with the general specification. Some examples of this idea can be found in:

- In [9], this idea is based on CCS and on a procedure for solving equations of the form: $(A|X) \setminus L = B$, where X is the unknown. The obtained submodule is a result of automatic transformations and guidance of the user. Observational equivalence is maintained between the given specification and the composition of the submodules.
- In [10] the solving of equation systems of form: $C_1(X) \sim P_1, \dots, C_n(X) \sim P_n$ is treated, where C_i is a context, P_i is an arbitrary process, $\sim$ is the bisimulation and X is the unknown process to be found (if possible). The decidability of these equations are studied. Another related issue, the decomposition of a logical specification formula F into sub-formulas $F_1, \dots, F_n$ which should express properties of the decomposed processes is also examined.

Related work can also be found in [16] where decomposition is used as a vehicle for verifying parallel systems.

Acknowledgements. I wish to express my gratitude to Juan Quemada who came up with the idea to isolate zones of interleavings and for his encouragement. Thanks goes to Joachim Parrow for his valuable comments and to Fredrik Orava, who clarified my ideas on the example. I also wish to thank Santiago Pavón and the anonymous referees for their comments and suggestions of improvements.

References

- [1] T. Bolognesi and E. Brinksma. An introduction to the iso specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
- [2] LotosPhere Project. Correctness preserving transformations. Technical Report Lo/WP1/T1.2/N0020, ESPRIT Ref:2304, May 1990.
- [3] S. Pavon, M. Hultström, J. Quemada, D. de Frutos, and Y. Ortega. Inverse expansion. In *Proc. 4th International Conference on Formal Description Techniques*, 1991.
- [4] S. Pavon, J. Quemada, D. Larrabeiti, and M. Hultström. Expansion inversa: Aplicacion inversa del teorema de expansion para descomponer un sistema lotos en subprocesos en paralelo. In *III Jornadas de Concurrencia, 1993, Universidad Politecnica de Valencia*, 1993.
- [5] Santiago Pavón Gómez. *Contribución al Análisis y Transformación de Especificaciones LOTOS*. PhD thesis, E.T.S.I. Telecomunicación, Madrid, Spain, 1990.
- [6] F. Khendek, G. v. Bochmann, and C. Kant. New results on deriving protocol specifications from service specifications. In *SIGCOM '89*, 1989. Computer Communications Review 19:4.
- [7] R. Langerak. Decomposition of functionality: A correctness preserving lotos transformation. In *Proc. IFIP WG 6.2 Symp. on Protocol Specification, Testing, and Verification X*, pages 229–242, Ottawa, Canada, 1990. IFIP WG 6.1, North-Holland.
- [8] E.Brinksma, R. Langerak, and P. Broekroelofs. Functionality decomposition by compositional correctness preserving transformation. *Computer Aided Verification, CAV 93, Crete, Greece*, 1993.
- [9] J. Parrow. Submodule construction as equation solving in CCS. *Theoretical Computer Science*, 68:175–202, 1989.
- [10] Xinxin Liu. *Specification and Decomposition in Concurrency*. PhD thesis, University of Aalborg, Denmark, 1992.
- [11] Vissers, Scollo, and van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In *IFIP Symp. on prot*, 1988.
- [12] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [13] ISO. LOTOS a Formal Description Technique based on the Temporal Ordering of Observational Behaviour. IS 8807, TC97/SC21, 1988.
- [14] Maria Hultström. Structural decomposition. Technical Report TRITA-IT R 94:13,ISSN 1103-534X,ISRN. KTH/IT/R -94:13 -SE, Royal Institute of Technology, 1994. to appear.
- [15] J. Quemada, S. Pavon, and A. Fernandez. State Exploration by Transformation with LOLA. In *Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, June 1989.
- [16] J.F. Groote and F. Moller. Verification of parallel systems via decomposition. In *Proceedings of CONCUR 92*, W.R. Cleaveland (ed), *Lecture Notes in Computer Science* 630, pp62–76. Springer-Verlag, 1992.