More Efficient Functionality Decomposition in LOTOS[1]

Monika Kapus-Kolar
Jožef Stefan Institute, P.O.B. 3000, SI-1001 Ljubljana, Slovenia
Phone: +386 61 1773 531, Fax: +386 61 1262 102
E-mail: monika.kapus-kolar@ijs.si

Keywords: distributed service implementation, automated protocol synthesis, LOTOS

Edited by: Rudi Murn
Received: June 30, 1998 **Revised:** January 16, 1999 **Accepted:** May 18, 1999

An improved functionality-decomposition transformation for Basic LOTOS specifications is proposed which, given a specification of the required external behaviour (the expected service) of a system and a partitioning of the specified service actions among the system components, derives the behaviour of individual components implementing the service. There may be an arbitrary finite number of components, pairwise communicating by executing common actions and/or by exchanging messages over queues with infinite capacity.

1 Introduction

The top-down design strategy starts by identifying the required external behaviour of a system (its service), followed by gradual refinement of its internal structure. The crucial top-down design transformation is functionality decomposition, used to decompose a given process into a number of concurrent interacting subprocesses. It is widely used in many design cases, both for decomposing hardware and software, e.g. in design of circuits, computers, computer networks, distributed systems and telecommunication networks - for distributed implementation of various types of servers, controllers, testers etc. The implementation of a service of a particular layer in the standard reference model for open systems interconnection requires for example derivation of a suitable behaviour of the protocol entities of the layer.

By decomposition, an abstract system specification is refined into a less abstract specification better reflecting the inherent system structure, i.e. its functional components and their spatial distribution. Thereby a specification becomes more tractable for formal reasoning and implementation. The most elegant way is to start with a verified specification of the system service and then apply to it only verified correctness-preserving transformations, i.e. transformations refining the system structure without affecting its service, to avoid the need for posterior verification of the obtained detailed system specification.

If a system is specified in a formal language (Turner, 1993), 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 formalized, 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, understanding and employing 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 for example languages based on process algebras, among which we concentrate on Basic LOTOS, the core sublanguage of LOTOS (Bolognesi and Brinksma, 1987), a standard specification language originally intended for specification of Open Systems Interconnection standards, but now widely employed for specification of all kinds of systems (both software and hardware) where external behaviour and/or inter-component interaction are of primary importance - see the Applications section in (WELL).

The problem addressed in our paper is system decomposition into a pre-defined number of components pairwise communicating by executing common actions and/or by exchanging messages over unbounded reliable first-in-first-out (FIFO) channels. It is assumed that all actions in the service specification are already pre-allocated to individual components, so that the task is only to derive specification of the inter-component communication, i.e. the protocol. The transformation is expected to be compositional, i.e. to

[1] A preliminary version of the paper appeared in the IEEE CS proceedings of the EUROMICRO'97 conference.

reflect the structure of the given service specification in the derived component specifications.

Saleh in his very exhaustive survey (Saleh, 1996) identifies 37 protocol synthesis methods, many of them further diversified into several variants. Among the 37 methods, 10 are based on LOTOS or similar models. For Basic LOTOS and multi-component servers with asynchronous internal channels, probably the most representative reference is (Kant et al., 1996), whose synthesis method has evolved from a long chain of similar methods, starting with (Bochmann and Gotzhein, 1986). For servers with synchronous channels, such a reference synthesizing earlier approaches is (Brinksma and Langerak, 1995), though its method is only intended for two-party servers. Thus in our paper we take (Kant et al., 1996) as the starting point, and (Brinksma and Langerak, 1995) as an additional source of ideas. We generalize (Kant et al., 1996) to servers with both synchronous and asynchronous channels, suggest how to correct the errors identified in (Kant et al., 1996) (one also in (Brinksma and Langerak, 1995)), and provide a hint for reducing the number of the necessary protocol interactions. Our paper is a corrected and enhanced version of (Kapus-Kolar, 1997), whose main deficiency is that the adopted specification language is semantically not exactly equivalent to Basic LOTOS, as it is here.

The paper is organized as follows. Section 2 introduces the adopted specification language. In Section 3 we explain the basic principles of protocol derivation. In Section 4 we propose distributed implementation for each individual type of service behaviour or subbehaviour. Section 5 includes discussion and conclusions.

The paper should be quite self-sufficient, though to keep it reasonably short, the presentations in the remaining sections are rather concise. Thus a reader unfamiliar with LOTOS-based protocol synthesis is advised to first study the more tutorial-like papers (Kant et al., 1996; Brinksma and Langerak, 1995). The two papers should also be referred to for discussion on earlier methods. Additional pointers to comparative studies can be found in (Saleh, 1996).

Before proceeding to details, let us once more try to motivate the reader by emphasizing that the algorithm described below is not intended solely for embedding in some CAD tool. The following sections provide many hints on how to systematically design correct and efficient distributed service implementations, that might be useful for the design of any multi-component system.

2 Specification Language

The language defined in Table 1 has been conceived with an aim to include in an abstract and concise way all constructs defined in Basic LOTOS (Bolognesi and Brinksma, 1987), in the exclusive setting of the protocol derivation problem formally defined in Section 3.1.

The following typographical convention has been adopted: If $\mathcal{X}$ is some universe of elements where "x" denotes its name and stands for any letter, 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. In particular, a b stands for a behaviour (for a process exhibiting it), a c for a system component, an a for a non-δ action, a g for an interaction gate or an action on it, a u for a user-defined action name, an s for **i** or a u (for a service-action name), an m for a protocol message, an r for a gate renaming, a p for the name of an explicitly specified process, an n for a process-parameter name, a v for a process-parameter value, and an e for a service specification subexpression. Let $b' \leq b$ and $b' < b$ respectively denote that b' is a subbehaviour or a proper subbehaviour of b.

We concentrate on the semantics of the language, informally overviewed below, but its syntax is intentionally kept simple, to simplify the presentation of protocol derivation. Throughout the paper, the usual self-understood forms of "syntactic sugar" are used where convenient, e.g. switching between the infix and the prefix notation, parentheses, omission of implicitly implied parts of the specification, etc. Several illustrative service and protocol specifications can be found in Sections 3 and 4.

stop denotes inaction of the specified process, e.g. of the system as a whole, of an individual system component, or of some other partial system behaviour.

Actions with reserved names δ (in the original LOTOS syntax specified by **exit**) and **i** respectively denote successful termination and an internal action of the specified process. In a service specification they are furnished with a superscript c indicating the component controlling their execution, but the superscript doesn't belong to the action name - the selection of c influences the protocol derivation algorithm, but is irrelevant for the service itself.

u^c denotes a service primitive, i.e. a type u interaction between a system user and the system component c.

If two components c and c' communicate synchronously (the condition encoded as $\neg FIFO(c,c')$), they can exchange a protocol message m in an interaction $sync_{c,c'}!m$, where the order of c and c' is irrelevant. If a c and a c' communicate asynchronously (i.e. $FIFO(c,c')$), c can send an m to c' by a $send_{c,c'}!m$, while c' receives the message by a $rec_{c,c'}!m$. When a protocol action appears in a local behaviour specification, explicit specification of the location qualifier identifying the particular system component is unnecessary. The parameter m is supposed to be always specified as a constant, so that it can be considered a part of the action name. **sync**, **send** and **rec** actions

Name of the construct	Type(s)	Syntax
Inaction	b	**stop**
Successful termination	b	δ^c
Internal action	a,s^c	$\mathbf{i}^c$
Service primitive	a,g,s^c	u^c
Protocol synchronization	a,g	$sync_{c,c'}!m$ where $\neg FIFO(c,c')$
Protocol message transmission	a,g	$send_{c,c'}!m$ where $FIFO(c,c')$
Protocol message reception	a,g	$rec_{c,c'}!m$ where $FIFO(c,c')$
Sequential composition	b	$b_1 >> b_2$
Action prefix	b	$a; b_2$
Choice	b	$b_1 [] b_2$
Parallel composition	b	$b_1 \lvert G \rvert b_2$
Disabling	b	$b_1 [> b_2$
Hiding	b	**hide** G **in** b_1
Gate renaming	r	$g \rightarrow g'$
Renaming	b	**ren** R **in** b_1
Process definition		$p(n) := b_1$
Process instantiation	b	$p(v)$

Table 1: The specification language abstract syntax

are only allowed in the derived protocol specifications.

"$b_1 >> b_2$" denotes a process behaving after successful termination of b_1 as b_2, where δ of b_1 is interpreted in "$b_1 >> b_2$" as $\mathbf{i}$. "$a; b_2$" is the special case of the sequential composition where b_1 is an individual action, so that no $\mathbf{i}$ is needed for transfer of control to b_2.

"$b_1 [] b_2$" denotes a process ready to behave as b_1 or as b_2.

"$b_1 \lvert G \rvert b_2$" denotes parallel composition of processes b_1 and b_2. Actions listed in G and δ are only executable as common actions of the two processes, while other actions the processes execute independently. By connecting processes by "|||" or "||" one shortly specifies their minimal or maximal synchronization, respectively. Specification of a parallel composition of an empty list of processes is an empty string identifier ε, while parallel composition of a singleton set of processes equals the only member of the set.

"$b_1 [> b_2$" denotes a process with behaviour b_1 potentially disrupted by behaviour b_2. While b_1 is still active, the process might terminate by executing δ in b_1.

"**hide** G **in** b_1" denotes a process behaving as b_1 with its actions listed in G renamed into $\mathbf{i}$, i.e. the gates made internal to the process (hidden from its environment). Hiding of an action doesn't influence its location, i.e. hiding of a u^c results in $\mathbf{i}^c$.

"**ren** R **in** b_1" denotes a process behaving as b_1 with its visible gates renamed as specified in R.

$p(n) := b_1$ defines a process p with behaviour b_1, and a $p(v)$ defines an instantiation of the process. Parametrization of processes constituting a service specification is not allowed, while in the derived protocol specification, the local mappings of the processes might need input parameters. Process parametrization is not allowed in Basic LOTOS, but we don't consider that a problem, for parameters can usually be avoided, if so desired, by switching to a less concise specification style interpreting the parameter value as a part of the process name. Anyway, in the full LOTOS, parametrization is legal.

A *specification* is a list of process definitions. The first process on the list is supposed to denote the behaviour that the specification is defining, i.e. the *main* process. As such, it must never be instantiated within the specification. Specifically, let *Server* denote the main process in the specification of the service that is being implemented.

In the original LOTOS syntax, explicit processes are defined on formal gates, that are associated with actual gates upon process instantiation. In our simplified language, the gate instantiation can be expressed as renaming of the gates on which a process is originally defined applied to the particular process instance (see for example the instantiation of process "Proc" in Table 22).

The relation used throughout the paper for judging equivalence of behaviours is observational equivalence $\approx$ (Bolognesi and Brinksma, 1987), i.e. we are interested only into the external behaviour of processes, that is into the actions that they make available for synchronization with their environment (all actions except $\mathbf{i}$ and actions transformed into $\mathbf{i}$ by hiding).

The protocol derivation mapping defined below in some cases introduces an ε specifying no actions. ε is equivalent to δ (as execution of no actions is a success by definition), except for an additional absorption rule (introduced for the purposes of protocol synthesis) stating the $(\varepsilon >> b)$ is equivalent to b.

$Server := Loop$
$Loop := ((Idle[> (event^1; ((report^2; \delta^1)\|\|\|(report^3; \delta^1)))) >> Loop)$
$Idle := ((test^1; Idle)[](play^1; Idle))$
$\forall\{c, c'\} : FIFO(c, c')$
$Server_1 := ((Idle_1[> ((event^1; ((send_2!1; \delta)\|\|\|(send_3!1; \delta))) >> ((rec_2!2; \delta)\|\|\|(rec_3!3; \delta)))) >> Server_1)$
$Idle_1 := ((test^1; Idle_1)[](play^1; Idle_1))$
$Server_2 := (\mathbf{rec}_1!1; report^2; send_1!2; Server_2)$
$Server_3 := (\mathbf{rec}_1!1; report^3; send_1!3; Server_3)$

Table 2: An example service and its derived protocol

In many of the protocol examples given below, elimination of ε is not the only simplification modulo observational equivalence that the specifications have undergone. In addition, process parametrization has been omitted where redundant.

3 Principles of Protocol Derivation

3.1 Problem definition

The protocol derivation problem is defined as follows. Given

- the above described system of components and channels for protocol interactions, with all FIFO channels initially empty,
- a specification of the required system service (non-blocking, with no non-executable or otherwise irrelevant parts), and
- a suitable (defined by the restrictions in Section 4) partitioning of the specified actions among the system components,

derive such behaviour of individual components that, when the **sync**, **send** and **rec** actions are hidden, the overall system behaviour is observationally equivalent to the specified service and the server never stops with any of its FIFO channels non-empty.

An illustrative example of a service and its protocol is given in Table 2. The protocol has been derived as suggested below and subsequently simplified. There is a server consisting of asynchronously communicating components 1 to 3, supporting users 1 to 3, respectively. Whenever user 1 signals a particular event, the server reports it to users 2 and 3, and subsequently becomes ready for a new signal from user 1. Protocol messages of type 1, issued by component 1, serve for reporting the signalled event to components 2 and 3, while messages 2 and 3 confirm to component 1 that the event has been reported to users 2 and 3, respectively. While there is no event to report, the server idles, i.e. allows user 1 to play and execute tests, both locally at component 1.

3.2 Mapping T

We are looking for a mapping $\mathbf{T}_c(e, z)$ which would take any service specification subexpression e and translate it into its counterpart at any individual component c, within a given context z. The mapping of individual subexpression types is given in Section 4. A $\mathbf{T}_c(e, z)$ implements the service actions within e allocated to c and the necessary protocol interactions between c and the rest of the system components.

During service execution, each instantiation of an explicit process p gives raise to a new instance of any b within the process body. Since the aim of mapping a specification e of such a b is a proper implementation of each particular instance of b, z for mapping the e must be the identifier of the particular instance of p. The particular instance of b is then unambiguously identified by "$z.Z(e)$", where $Z(e)$ identifies e within the specification of p's body. Protocol optimization by re-use of instance identifiers shall not be systematically considered, though often possible.

3.3 The basic principles of protocol construction

Like (Kant et al., 1996) and previous similar algorithms, our algorithm is based on a small set of intuitive rules that can be easily expressed in an informal way:

1) Only the server components responsible for actions in the service specification should participate in the execution of the service.
2) An individual service action must always be implemented at the component to which it has been pre-assigned.
3) Every protocol message should unambiguously identify the behaviour that it helps to implement.
4) Service choices should always be resolved exclusively by service actions. Protocol actions only communicate the decisions between server components.
5) To simplify protocol derivation, we partition the service actions in such a way that
   - conflicts between distributed implementations of concurrent service parts are a priori avoided, and
   - all service choices can be resolved locally at individual components.

6) With the previous heuristics, *the sole purpose of protocol actions is to report on local terminations of individual service parts.*

3.4 Service behaviour attributes, particularly termination types

Mapping **T** is guided by various pre-calculated attributes of individual service subbehaviours, actually (speaking in terms of the specification syntax) of the service specification subexpressions by which they are represented. Likewise, the attributes are also defined for each of the explicitly specified processes constituting the service. Computation and selection of attributes is the key activity in the protocol derivation process. After we find a set of attributes that is both consistent and known to lead to an efficient protocol, the protocol derivation itself (i.e. application of mapping **T**) is trivial. (That doesn't mean that the mapping has also been trivial to conceive and prove!)

For any attribute $X(\dots)$ whose value is a set of elements x, let $X_x(\dots)$ indicate $x \in X(\dots)$, and vice versa.

The basic attributes of a behaviour b are its starting components (SC), its ending components (EC) and its participating components (PC), respectively listing the system components executing a starting, an ending, or any service action within b. E.g., for a behaviour

$$b = (((((\mathrm{a}^1;\delta^3)|||(\mathrm{b}^2;\delta^3)) >> (\mathrm{c}^3;stop))[]\delta^1)||| \\ ((\mathrm{d}^1;\delta^1)[> (\mathrm{a}^1;\delta^4)))$$

we have $SC(b) = \{1,2\}$, $EC(b) = \{1,4\}$, $PC(b) = \{1,2,3,4\}$.

Boolean attribute $IT(b)$ indicates whether b might immediately terminate (the above b can not, while its part "$(\dots[]\delta^1)$" in isolation could). Boolean attribute $ST(b)$ indicates whether b synchronizes its termination with its environment, i.e. its δ is not "consumed" by a sequential composition operator (as is for example δ^3 in the above b). Boolean attribute $DT(b)$ indicates whether a termination of b might be decision-making (as are in the above b both occurrences of δ^1).

All the above attributes are *generic properties of behaviours.* Solving a system of recursive equations for such an attribute, one should choose a solution minimizing all the process attributes, to respect the natural semantics of the attribute.

There are two more attributes, TC and TC^+, but they are not generic properties of behaviours. They are selected by a designer, to implement his/hers specific protocol derivation strategy. The exact role of TC and TC^+ is to dictate the *termination type* of individual service parts within the distributed service implementation, as follows:

An individual service action must of course be implemented at the component to which it is pre-allocated. At any other component, we propose that it is *selectively* mapped into a **stop** or an ε (semantically equivalent to δ). (Kant et al., 1996; Brinksma and Langerak, 1995) strictly map to ε, and consequently the quality of the derived protocol suffers.

Depending on the mapping of the ending actions of a particular b, in the case that the server is running a terminating alternative of b the behaviour $\mathbf{T}_c(b,z)$ of a component c concludes either by **stop** (not preceded by δ) or by δ. In the latter case, c is a terminating component of b, i.e. $TC_c(b)$. It is important that $\mathbf{T}_c(b,z)$ concludes for all terminating alternatives of b in the same manner, i.e. always by **stop** or always by δ.

$TC_c(b)$ is always a consequence of $TC_c^+(b)$ indicating that the surrounding context of b requires c to conclude its implementation of b by δ. Besides, $EC_c(b)$ implies $TC_c(b)$, for the ending components of a b are responsible for detecting its termination.

If $TC_c(b)$, $\mathbf{T}_c(b,z)$ terminates on its own and so allows c to *sequentially* proceed to the subsequent activities. That is the usual behaviour-termination type in LOTOS-based protocol derivation. If $\neg TC_c(b)$, $\mathbf{T}_c(b,z)$ concludes by inactivity, that is *disrupted* by a subsequent activity of c specified outside $\mathbf{T}_c(b,z)$. The *opportunity for protocol optimization* lies in the fact that setting $TC_c(b)$ to *false* renders the protocol messages serving solely for termination of $\mathbf{T}_c(b,z)$ unnecessary. For better understanding, observe that for a service behaviour "$b_1 >> b_2$", δ in b_1 is invisible for the service users and hence its implementation is irrelevant, as long as control is properly transferred to the implementation of b_2.

As TC and TC^+ are attributes that are rather selected than computed, we don't provide strict rules for them - just the restrictions that must be respected are stated in Section 4. Within those limits, TC and TC^+ should be selected according to the relevant optimization criteria. An always present criterion is minimization of the (worst-case or average) inter-component communication. Another criterion, often conflicting with the former one, might be declaring for some service parts b and components c that $TC_c(b)$ is desirable, for c should terminate $\mathbf{T}_c(b,z)$ as soon as possible, e.g. to release some resources. A third criterion might be to make a pair of server components exchange a protocol message at a particular point of service execution, e.g. to convey some data. Moreover, scheduling of protocol exchanges on FIFO channels usually requires prevention of channel buffers overcrowding.

When trying to find a solution better than the old ($TC^+(b) = TC(b) = C$), one must be aware that in some cases a different solution might also result in a *less* efficient protocol, depending on subtle properties of the service structure. Thus the optimization should be performed by thorough analysis of the entire service specification. Nevertheless, one should never simply retreat to ($TC^+(b) = TC(b) = C$), for that might re-

$\mathbf{Send}_c(C,m)$	$:= \|\|\|(\{(send_{c'}!m;\delta)\|((c' \in (C \setminus \{c\})) \wedge FIFO(c,c'))\} \cup$ $\{(sync_{c'}!m;\delta)\|((c' \in (C \setminus \{c\})) \wedge \neg FIFO(c,c'))\})$
$\mathbf{Rec}_c(C,m)$	$:= \|\|\|(\{(rec_{c'}!m;\delta)\|((c' \in (C \setminus \{c\})) \wedge FIFO(c,c'))\} \cup$ $\{(sync_{c'}!m;\delta)\|((c' \in (C \setminus \{c\})) \wedge \neg FIFO(c,c'))\})$
$\mathbf{Exch}_c(C,C',m)$	$:=$ (if $(c \in C)$ then $\mathbf{Send}_c(C',m)$ else ε endif $\|\|\|$ if $(c \in C')$ then $\mathbf{Rec}_c((C \setminus \{c'\|((c \in C) \wedge (c' \in C') \wedge \neg FIFO(c,c'))\}),m)$ else ε endif)
$\mathbf{Proj}_c(G)$	$:= \{u^c\|(u^c \in G)\}$
$\mathbf{Proj}_c(R)$	$:= \{(u^c \to u'^c)\|((u^c \to u'^c) \in R)\}$
$\mathbf{Select}(C,z)$	:= a (within context z) deterministically selected element of C
$\mathbf{Term}_c(b,z)$	$:= \mathbf{Term}'_c(b,TC^+(b),z)$
$\mathbf{Term}'_c(b,C,z)$	:= if $(EC(b) = \phi)$ then $\mathbf{T}_c(b,z)$ else if $EC_c(b)$ then $(\mathbf{T}_c(b,z) >> \mathbf{Send}_c((C \setminus TC(b)),z.Z(b)))$ else if $((c \in C) \wedge \neg TC_c(b))$ then if $(\|EC(b)\| = 1)$ then $(\mathbf{T}_c(b,z)[> \mathbf{Rec}_c(EC(b),z.Z(b)))$ else $((\mathbf{T}_c(b,z)[> \delta)\|\|\mathbf{Rec}_c(EC(b),z.Z(b))$ endif else $\mathbf{T}_c(b,z)$ endif endif endif
$\mathbf{Alt}_c(b,b',z)$	$:= \mathbf{Alt}'_c(b,(PC(b') \cap TC(b')),z)$
$\mathbf{Alt}'_c(b,C,z)$	:= if $(EC(b) = \phi)$ then $\mathbf{T}_c(b,z)$ else if $(c = \mathbf{Select}((TC^+(b) \cap PC(b)),Z(b)))$ then $(\mathbf{Term}_c(b,z) >> \mathbf{Send}_c((C \setminus (PC(b) \cup TC^+(b))),z.Z(b)))$ else if $((c \in C) \wedge \neg PC_c(b) \wedge \neg TC_c^+(b))$ then $\mathbf{Rec}_c(\mathbf{Select}((TC^+(b) \cap PC(b)),Z(b)),z.Z(b))$ else $\mathbf{Term}_c(b,z)$ endif endif endif

Table 3: Functions used in mapping **T**

sult in *protocol errors*, as explained in Section 4, if not to mention that for some service types, that straightforward solution is extremely inefficient (as demonstrated by the example in Table 7, discussed in Section 4.1).

To reduce the computational complexity, the rules for attribute evaluation in Section 4 are not exact, in the sense that they sometimes assume that the protocol derivation algorithm must pay attention to a service scenario that a more careful examination of the service specification would identify as non-executable. Particularly the rules neglect the positive impact of synchronization between parallel behaviours. Consequently, the decomposition transformation might generate some redundant protocol interactions, or the transformation might be unjustly declared inapplicable to a particular action partitioning. In other words, more exact attribute evaluation rules would result in a more generally applicable transformation generating more efficient protocols. But as the proposed rules are strictly pessimistic, they are nevertheless sufficient for protocol correctness (with the harmless exception that the generated component specifications might comprise some parts that are non-executable within the context of the system), provided that the mapping **T** itself is correct.

Of course, computation of attributes might also fail, implying that it is impossible to satisfy all the given restrictions simultaneously, or result in a protocol not satisfying the adopted optimization criteria. In that case, one should try to find a better service action partitioning, or artfully insert in the service specification some dummy internal actions (as long as the service remains observationally equivalent to the original).

3.5 Some auxiliary specification-generating functions, particularly implementation of terminations

Table 3 defines a set of auxiliary specification-generating functions for the mapping **T**. The first two functions implement sending or receiving at a c of a message m between c and any c' in $(C \setminus \{c\})$, independently for each c'. $\mathbf{Exch}_c(C,C',m)$ implements the actions of c necessary to perform the task of each member of C' receiving m from each member of C other than itself. Since message exchanges in **sync** actions are bi-directional, that allows some optimizations.

$\mathbf{Proj}_c(G)$ and $\mathbf{Proj}_c(R)$ respectively project onto the gates of c a set of gates and a set of renamings. $\mathbf{Select}(C,z)$ deterministically selects within context z a component in C.

If the surrounding context of b requires a c to conclude its implementation of the terminating alternatives of b (if any) by δ, formally $TC_c^+(b)$, that might be implemented by the mapping $\mathbf{T}_c(b,z)$ itself, i.e. c included into $TC(b)$. Alternatively, it might be better to have $\neg TC_c(b)$ and to make c exit its implementation of b upon reception of special messages (sent by

$b = (((\mathrm{a}^1;\delta^1)[](\mathrm{b}^1;\mathrm{c}^2;\mathrm{d}^1;\delta^1))\|\|\|((\mathrm{a}^1;\delta^1)[](\mathrm{b}^1;\mathrm{c}^2;\mathrm{d}^1;\delta^1)))$, $\neg FIFO(1,2)$
if $TC_2(b)$ then $\mathbf{T}erm'_1(b,\{1,2\},z) := (((\mathrm{a}^1;sync!z.1;\delta)[](\mathrm{b}^1;sync!z.2;sync!z.3;\mathrm{d}^1;\delta))\|\|\|$ $((\mathrm{a}^1;sync!z.4;\delta)[](\mathrm{b}^1;sync!z.5;sync!z.6;\mathrm{d}^1;\delta)))$ $\mathbf{T}erm'_2(b,\{1,2\},z) := (((sync!z.1;\delta)[](sync!z.2;\mathrm{c}^2;sync!z.3;\delta))\|\|\|$ $((sync!z.4;\delta)[](sync!z.5;\mathrm{c}^2;sync!z.6;\delta)))$ else $\mathbf{T}erm'_1(b,\{1,2\},z) := ((((\mathrm{a}^1;\delta)[](\mathrm{b}^1;sync!z.2;sync!z.3;\mathrm{d}^1;\delta))\|\|\|$ $((\mathrm{a}^1;\delta)[](\mathrm{b}^1;sync!z.5;sync!z.6;\mathrm{d}^1;\delta))) >> (sync!z.7;\delta))$ $\mathbf{T}erm'_2(b,\{1,2\},z) := (((sync!z.2;\mathrm{c}^2;sync!z.3;stop)\|\|\|(sync!z.5;\mathrm{c}^2;sync!z.6;stop))$ $[> (sync!z.7;\delta))$ endif

Table 4: A service behaviour b and some of its possible implementations wrt. its termination type

$b = (\mathrm{a}^1;((\mathrm{b}^3;\delta^3)\|\|\|(\mathrm{c}^4;\delta^4)))$, $\forall\{c,c'\} : FIFO(c,c')$, $TC^+(b) = \{1,3,4\}$, $TC(b) = \{3,4\}$
$\mathbf{A}lt'_1(b,\{1,2,3\},z) := (((\mathrm{a}^1;((send_3!z.2;\delta)\|\|\|(send_4!z.2;\delta)) >> stop)[> \delta)$ $\|\|\|((rec_3!z.1;\delta)\|\|\|(rec_4!z.1;\delta)))$
$\mathbf{A}lt'_2(b,\{1,2,3\},z) := (rec_3!z.1;\delta)$
$\mathbf{A}lt'_3(b,\{1,2,3\},z) := (rec_1!z.2;\mathrm{b}^3;send_1!z.1;send_2!z.1;\delta)$
$\mathbf{A}lt'_4(b,\{1,2,3\},z) := (rec_1!z.2;\mathrm{c}^4;send_1!z.1;\delta)$

Table 5: Illustration of function $\mathbf{A}lt'$

the ending components of b) together indicating global termination of b. Function **Term**, an extension of mapping **T**, implements the additional protocol interactions. Function $\mathbf{T}erm'$ is a generalization of **Term** that signals the global termination to a component set C.

For illustration of function **Term**, consider the service behaviour b in Table 4 with requirement $TC_2^+(b)$. If we set $TC_2(b)$ to *true*, component 2 implements all the "a" actions as ε. Consequently, whenever component 1 selects an "a" alternative, it must send a special indication to component 2, for component 2 does not participate in such an alternative and needs a message for its detection and subsequent proper local termination. In the worst case, execution of b requires two such messages. If we set $TC_2(b)$ to $false$ instead, component 2 simply executes its part of b and then stops. Later its inactivity is disrupted by a message (a single one!) implemented by **Term** applied to b. Obviously, $\neg TC_2(b_1)$ is the more efficient solution.

Function **Alt** is an extension of function **Term** in the sense that *one* of the components knowing that an alternative b of a b' has been activated (and terminated) indicates the fact to the yet non-informed participants of b' that need the information. Function $\mathbf{A}lt'$ is a generalization of **Alt** that sends the information to a component set C, where necessary. Note that both $\mathbf{A}lt'$ and $\mathbf{T}erm'$ can serve for adding δ to implementation of a b at a c, but there is a slight difference: $\mathbf{T}erm'$ implements δ upon c receiving messages from *all* the ending components of b and is thus also suitable if $(\mathbf{T}_c(b,z) \neq stop)$ or if c must detect *global* termination of b. Otherwise c may enable δ already after receiving a *single* message from a *selected* participant of b (function $\mathbf{A}lt'$), for the reception may disrupt $\mathbf{T}_c(b,z)$, supposed to be equivalent to **stop**, at any time.

The effects of function $\mathbf{A}lt'$ are illustrated in Table 5. Because of $(TC_1^+(b) \wedge \neg TC_1(b))$, component 1 detects global termination of b by receiving messages from the ending components 3 and 4, introduced by the **Term** part of $\mathbf{A}lt'$. On the other hand we have $(\neg TC_2^+(b) \wedge \neg PC_2(b))$, thus $\mathbf{A}lt'$ makes component 3 (selected among {1,3,4}) indicate activation of b to component 2. 1 and 3 in the second argument of $\mathbf{A}lt'$ are irrelevant, since both components participate in b, and component 3 even terminates on its own.

3.6 Assumptions (obligations) for mappings T, $\mathbf{T}erm'$ and $\mathbf{A}lt'$

This section is only intended for readers deeply interested in technical details, while others are advised to refer to it only if having difficulties with the understanding of Section 4.

Mapping **T** is compositional, i.e. a mapping of a behaviour b is defined in terms of the mappings of its subbehaviours b'. Thus when designing $\mathbf{T}_c(b,z)$, we always make some assumptions for each individual $\mathbf{T}_c(b',z)$. One level higher, analogous statements appear as proof obligations for $\mathbf{T}_c(b,z)$. As mapping $\mathbf{T}erm'$ or $\mathbf{A}lt'$ shall often be used instead of **T**, the obligations (suitably adapted) also apply to them.

Let $Func(b,Arg)$ denote behaviour of the considered distributed system where every component c behaves like $Func_c(b,Arg)$, where $Func$ is **T**, $\mathbf{T}erm'$ or $\mathbf{A}lt'$ and Arg are the remaining arguments of the mapping. Let $Func(b,Arg)^*$ denote $Func(b,Arg)$ with protocol interactions hidden. Below we list the correctness criteria adopted for the system be-

haviour $Func(b, Arg)$ and the individual component behaviours $Func_c(b, Arg)$ within its context, assuming that all the FIFO channels are initially empty. Most of the rules are just common sense or can be understood in the light of the protocol derivation guidelines stated so far, while the others support implementation of various individual behaviour composition operators, explained in Section 4. The obligations are indeed numerous, but the correctness proof (Kapus-Kolar, 1998) for the protocol derivation transformation has identified them as necessary, as will the reader if trying to thoroughly understand the mappings in Section 4.

1) If $\neg PC_c(b)$, then
$(((Func = \mathbf{T}) \wedge \neg TC_c(b)) \vee$
$((Func(b, Arg) = \mathbf{T}erm'(b, C, \ldots)) \wedge$
$(c \notin (TC(b) \cup C))) \vee$
$((Func(b, Arg) = \mathbf{A}lt'(b, C, \ldots)) \wedge$
$(c \notin (TC^+(b) \cup (C \setminus PC(b)))))$
implies $Func_c(b, Arg) = stop$ and $TC_c(b)$ implies $Func_c(b, Arg) = \varepsilon$.
2) Any visible action offered by $Func_c(b, Arg)$ is either a service action pre-allocated to c within b, a protocol interaction associated with termination of a $b' \leq b$, or δ.
3) In b, every $b' \leq b$ is assigned its unique identifier, that is present in every protocol message associated with termination of b'.
4) If $(((Func = \mathbf{T}) \wedge TC_c(b)) \vee$
$((Func(b, Arg) = \mathbf{T}erm'(b, C, \ldots)) \wedge$
$(c \in (TC(b) \cup C))) \vee$
$((Func(b, Arg) = \mathbf{A}lt'(b, C, \ldots)) \wedge$
$(c \in (TC^+(b) \cup (C \setminus PC(b))))))$,
then $Func_c(b, Arg)$ terminates in all the terminating alternatives of b, otherwise never terminates.
5) Within $Func(b, Arg)$, for any terminating alternative of b, any $Func_c(b, Arg)$ not terminating for the alternative enters inaction before the last of the ending components c' terminates $Func_{c'}(b, Arg)$.
6) $Func(b, Arg)$ inherits from b its starting actions, with the exception that a starting δ in b is considered equivalent to a starting $\mathbf{i}$ in $Func(b, Arg)^*$, provided that there is a c guarding both the δ and the $\mathbf{i}$.
7) Within $Func(b, Arg)$, for any terminating alternative of b, any $c \in SC(b)$ is activated before the last of the ending components c' terminates $Func_{c'}(b, Arg)$.
8) Pretending that δ of $Func(b, Arg)$ is already executable when enabled by all c having it in $Func_c(b, Arg)$, $Func(b, Arg)^* \approx b$, with the exception that enabling of δ is allowed to be an internal decision of $Func(b, Arg)^*$ if $\neg ST(b)$. If b is the entire service, exact observational equivalence is required.
9) For each c, $Func_c(b, Arg)$ contains only actions that are executable within $Func(b, Arg)$.
10) No $Func_c(b, Arg)$ ever requires that the incoming protocol messages on FIFO channels are received in an order other than that of their transmissions within $Func(b, Arg)$.
11) Any protocol message sent within $Func(b, Arg)$ is also received within it.

$Server := (((\mathrm{a}^1; Proc) [] (\mathrm{b}^1; \delta^1)) >> (\mathrm{b}^3; \delta^3))$ $Proc := (\mathrm{c}^2; Proc)$ $\forall \{c, c'\} : FIFO(c, c')$
$Server_1 := (((\mathrm{a}^1; send_2!1; stop) [] (\mathrm{b}^1; \delta)) >>$ $((send_2!2; \delta) \|\|\| (send_3!2; \delta)))$
$Server_2 := (((rec_1!1; Proc_2) [> (rec_1!2; \delta)))$ $Proc_2 := (\mathrm{c}^2; Proc_2)$
$Server_3 := (rec_1!2; \mathrm{b}^3; \delta)$

Table 7: An example implementation of a service combining finite and infinite alternatives

4 Distributed Implementation of Individual Service Specification Subexpression Types

In this section we describe distributed implementation of individual service specification subexpression types. Table 6 summarizes the rules valid for any subexpression of type b, while the more specific rules are given in separate tables, explained in Sections 4.2 to 4.8. Each of the tables is typically divided into four sections: 1) definition of the syntax of the expression e that is being mapped, 2) attribute calculation rules, 3) (optional) additional restrictions on e or its attributes, and 4) mapping $\mathbf{T}$ for e. Studying $\mathbf{T}_c(b, z)$ in the tables with the specific rules, the reader may with no harm pretend that the only server components are those participating in b. The explanations of the transformations have also been conceived from that viewpoint.

4.1 Rules applying to all behaviour types

The generally applicable rules in Table 6 deserve some explanation.

Rule $((EC(b) = \phi) \Rightarrow (TC^+(b) = \phi))$ prevents server components from interpreting a non-terminating b as terminating (i.e. no c can ever terminate, if $(EC(b) = \phi)$, for that might result in a protocol error, if b is alternative to a terminating b' (Kapus-Kolar et al., 1991). With that rule, *it is no longer necessary to report every process instantiation within the service to every component*, as it is in (Kant et al., 1996). An illustrative example is given in Table 7. In comparison with the solution suggested in (Kant et al., 1996), there is an infinite saving in protocol messages - two per every instantiation of "Proc".

if $(EC(b) = \phi)$ then $TC^+(b) = \phi$ else if $\exists b' : ((b < b') \wedge \neg PC_c(b'))$ then $\neg TC_c^+(b)$ else $TC_c^+(b)$ defined in the corresponding one of the Tables 10,12,13,15,17,20 endif endif
$(DT(b) \wedge ST(b)) \Rightarrow ((\lvert EC(b)\rvert = 1) \wedge (TC^+(b) = EC(b)))$
$EC(b) \subseteq (TC^+(b) \setminus (PC(b) \setminus EC(b))) \subseteq TC(b) \subseteq TC^+(b)$
if $\neg PC_c(b)$ then if $TC_c(b)$ then $\mathbf{T}_c(b,z) := \varepsilon$ else $\mathbf{T}_c(b,z) := stop$ endif else $\mathbf{T}_c(b,z)$ defined in the corresponding one of the Tables 9,10,12,13,15,17-19,20 endif

Table 6: Rules valid for any service subexpression b introduced for the purpose of protocol synthesis

The second row of Table 6 requires that a behaviour b whose termination might be both decision-making and synchronized has a single ending component that is also the only component regarding b as terminating. To understand the requirement, observe the service behaviour

$b = (((\mathrm{a}^1;\delta^1)[> (\mathrm{i}^1;\mathrm{b}^2;\delta^1))|[\mathrm{b}^2]|(\delta^1[](\mathrm{i}^1;\mathrm{b}^2;\delta^1))$

$e = b = stop$
$SC_c(b) = EC_c(b) = PC_c(b) = false$
$IT(b) = DT(b) = false$

Table 8: The specific rules for **stop**

$e = b = \delta^{c'}$	
$SC_c(b) = EC_c(b) = PC_c(b) = (c = c')$	
$IT(b) = true$	$DT(b) = false$
$\mathbf{T}_{c'}(e,z) := b$	

Table 9: The specific rules for $\delta^{c'}$

$e = b = b_1[]b_2$
$SC_c(b) = (SC_c(b_1) \vee SC_c(b_2))$
$EC_c(b) = (EC_c(b_1) \vee EC_c(b_2))$
$PC_c(b) = (PC_c(b_1) \vee PC_c(b_2))$
$IT(b) = (IT(b_1) \vee IT(b_2))$
$ST(b_1) = ST(b_2) = ST(b)$
$DT(b) = (DT(b_1) \vee DT(b_2) \vee IT(b))$
$TC_c^+(b_1) = (TC_c(b) \wedge PC_c(b_1))$ $TC_c^+(b_2) = (TC_c(b) \wedge PC_c(b_2))$
$\exists c' : (SC(b_1) = SC(b_2) = \{c'\})$
$\mathbf{T}_c(e,z) := (\mathbf{Alt}_c(b_1,b,z)[]\mathbf{Alt}_c(b_2,b,z))$

Table 10: The specific rules for choice

of the form "$((b_1[> b_2)|[\mathrm{b}^2]|b_3)$". δ of b_1 is decision-making for $(b_1[> b_2)$ and synchronized with δ of b_3, hence both δ *must be controlled by the same component* (in our case, it is the component 1). Thus

$((\lvert EC(b_1)\rvert = 1) \wedge (TC(b_1) = EC(b_1) = EC(b_3)))$.

Moreover, as δ of $\mathbf{T}_1(b_1,z)$ and the corresponding δ of $\mathbf{T}_1(b_1[> b_2,z)$ are guarded by δ of $\mathbf{T}_1(b_3,z)$, component 1 *can't report them separately* (only upon δ of $\mathbf{T}_1(b,z)$), necessitating

$(TC^+(b_1) = TC^+(b_1[> b_2) = TC(b_1))$.

Hence the considered rule applies to "$b_1[> b_2$". Also, $(\lvert TC^+(b_1[> b_2)\rvert = 1)$ (not to speak of a decision-making δ within b_3) implies

$(TC(b) = EC(b) = EC(b_1))$,

i.e. δ of b must be in all its alternatives controlled by component 1 (and indeed it is). (Kant et al., 1996; Brinksma and Langerak, 1995) ignore the fact that a decision-making δ of b_1 within a "$b_1[> b_2$" might be synchronized, thus they sometimes generate erroneous protocols. The two methods can only be amended by getting rid of the rule $(PC_c(b) \Rightarrow TC_c^+(b))$, as we have done.

In the third row of Table 6, the not yet explained idea is that $\neg PC_c(b)$ should imply $(TC_c^+(b) = TC_c(b))$. That is because a non-participant c of a b should not participate in $\mathbf{Term}_c(b,z)$, just as not in $\mathbf{T}_c(b,z)$, as stated in the last row of the table.

The $\mathbf{T}_c(b,z)$ rule in Table 6 implies that if $\neg PC_c(b)$, $TC_c^+(b')$ for any $b' < b$ can be $false$ by definition.

4.2 Implementation of inaction and termination

The specific rules for **stop** and δ^c are defined in Table 8 and Table 9, respectively.

4.3 Implementation of choice

For implementation of choice (Table 10) we adopt the usual restriction (Bochmann and Gotzhein, 1986) that alternatives must have an unique and common starting place. Provided that distributed implementation of individual alternatives is communication-closed and preserves their starting actions, the choice is entirely local.

Let b_1 be the selected alternative. After its execution, one of its ending components proceeds to informing on the selected alternative the missing terminating participants of b. That is implemented by using function **Alt** instead of **Term**. If no message is sent to a missing participant c of b, $\mathbf{T}_c(b,z)$ is equivalent to $\mathbf{T}_c(b_2,z)$, that is in the case of b_1 not enabled, so that

$b = ((\mathrm{a}^1;\mathrm{b}^2;\delta^2)[](\mathrm{c}^1;\mathrm{d}^3;\mathrm{e}^1;\delta^1))$, $FIFO(c,c') = (\{c,c'\} \in \{1,2\})$, $TC(b) = \{1,2\}$
$\mathbf{T}_1(b,z) := ((\mathrm{a}^1;sync_2!z.1;\delta)[](\mathrm{c}^1;send_3!z.2;rec_3!z.3;\mathrm{e}^1;sync_2!z.4;\delta))$
$\mathbf{T}_2(b,z) := ((sync_1!z.1;\mathrm{b}^2;\delta)[](sync_1!z.4;\delta)$
$\mathbf{T}_3(b,z) := (rec_1!z.2;\mathrm{d}^3;send_1!z.3;stop)$

Table 11: An example implementation of choice

$e = b = b_1 >> b_2$		
$SC_c(b) = SC_c(b_1)$	$EC_c(b) = EC_c(b_2)$	$PC_c(b) = (PC_c(b_1) \vee PC_c(b_2))$
$IT(b) = IT(b_1)$	$DT(b) = DT(b_2)$	$ST(b_1) = false$, $ST(b_2) = ST(b)$
$((\neg PC_c(b_2) \wedge TC_c(b_2)) \vee EC_c(b_1)) \Rightarrow TC_c^+(b_1) \Rightarrow ((EC(b) = \phi) \vee TC_c(b) \vee EC_c(b_1) \vee SC_c(b_2))$ $TC_c^+(b_2) = TC_c(b)$		
$\mathbf{T}_c(e,z)$:=if $TC_c^+(b_1)$ then $(\mathbf{T}erm_c(b_1,z) >> \mathbf{E}xch_c(EC(b_1),SC(b_2),z.Z(b_1)) >> \mathbf{T}erm_c(b_2,z))$ else if $SC_c(b_2)$ then $(\mathbf{T}erm'_c(b_1,\{c\},z) >> \mathbf{T}erm_c(b_2,z))$ else $(\mathbf{T}_c(b_1,z)[> \mathbf{T}erm_c(b_2,z))$ endif endif		

Table 12: The specific rules for sequential composition

the component waits for a disrupting message issued at some point after completion of b.

In the example in Table 11, component 2 is a missing terminating participant of the second alternative, thus it is notified of its execution. On the other hand, component 3 does not participate in the first alternative, but is not required to terminate it, thus it receives no notification.

Initial (decision-making) terminations of b are no longer forbidden (see Table 16), as they are in (Brinksma and Langerak, 1995). That is because each initial δ is now controlled by the component c' making the choice and guards termination of $\mathbf{T}_c(b,z)$ for any other $c \in PC(b)$.

4.4 Implementation of sequential composition and action prefix

Implementation of sequential composition is specified in Table 12. Proper sequencing of b_1 and b_2 requires that each $c \in EC(b_1)$ reports (by a message $z.Z(b_1)$) termination of $\mathbf{T}_c(b_1,z)$ to each $c' \in SC(b_2)$ (Kant et al., 1996). An analogous solution is employed for implementing action prefix (Table 13).

1) A $c \in TC^+(b_1)$ executes $\mathbf{T}erm_c(b_1,z)$, then its part of the exchanges of message $z.Z(b_1)$ sent from $EC(b_1)$ to $SC(b_2)$, and finally $\mathbf{T}erm_c(b_2,z)$.
2) If $(\neg TC_c^+(b_1) \wedge SC_c(b_2))$, the receptions of $z.Z(b_1)$ at c are implemented by upgrading $\mathbf{T}_c(b_1,z)$ into $\mathbf{T}erm'_c(b_1,\{c\},z)$; afterwards, c proceeds to $\mathbf{T}erm_c(b_2,z)$.
3) If $(\neg TC_c^+(b_1) \wedge \neg SC_c(b_2))$, c concludes $\mathbf{T}_c(b_1,z)$ by inaction, that might be disrupted by $\mathbf{T}erm_c(b_2,z)$ or some later activity.

The various situations are illustrated by the example in Table 14, where one can also observe a situation of b_1 having both a terminating and a non-terminating alternative.

As terminations of b_1 are not terminations of b, there is no strict rule for $TC_c^+(b_1)$, but we require that

1) $(\neg PC_c(b_2) \wedge TC_c(b_2))$ implies $TC_c^+(b_1)$, for otherwise $\mathbf{T}erm_c(b_2,z)$, unguarded in $\mathbf{T}(b,z)$, could prematurely disrupt $\mathbf{T}_c(b_1,z)$, and
2) $TC_c^+(b_1)$ implies $((EC(b) = \phi) \vee TC_c(b) \vee EC_c(b_1) \vee SC_c(b_2))$,

to prevent disruption of $\mathbf{T}erm_c(b_1,z)$ by actions upon termination of b in the alternatives where c doesn't participate in b_2.

4.5 Implementation of disabling

Implementing disabling (Table 15) we encounter similar problems as when implementing choice. The starting actions of the disrupting behaviour b_2 are alternatives to the actions (including δ) in the potentially disrupted b_1, thus we require (Brinksma and Langerak, 1995) b_2 to have an unique starting component c' that is also the only participant of b_1. (Kant et al., 1996) has tried to avoid the restriction, but unsuccessfully, with *potential protocol errors* (Kapus-Kolar, 1999). The difficult problem is that it is not sufficient to individually implement b_1 and b_2, if b_1 is terminating. We need a special termination scheme, like the one explained in the following.

If b_2 is activated, a termination of b is always a termination of b_2, and thus properly detected by all the participants of b needing the information, since by the adopted restriction, $(PC(b) = PC(b_2))$. However, if b_1 terminates without being disrupted by b_2, c' must subsequently report, where necessary, the termination to the other participants of b. For a receiving c'', the situation is exactly as in the case of "$b_1[]b_2$". At c', however, the transmission must not be attached sequentially to the b_1 part (as in (Kant et al., 1996)), because in that position it would be disruptable by the b_2 part (Kapus-Kolar, 1999). Hence it must be

$e = b = s^{c'}; b_2$			
$SC_c(b) = (c = c')$	$EC_c(b) = EC_c(b_2)$	$PC_c(b) = ((c = c') \vee PC_c(b_2))$	
$IT(b) = false$	$DT(b) = DT(b_2)$	$ST(b_2) = ST(b)$	$TC_c^+(b_2) = TC_c(b)$
$\mathbf{T}_c(e,z)$:=if $(c = c')$ then $(s^{c'}; (\mathbf{Send}_c(SC(b_2), z.Z(b_1)) >> \mathbf{Term}_c(b_2, z)))$ else if $SC_c(b_2)$ then $(\mathbf{Rec}_c(\{c'\}, z.Z(b_1)) >> \mathbf{Term}_c(b_2, z))$ else $\mathbf{Term}_c(b_2, z)$ endif endif			

Table 13: The specific rules for action prefix

$b = (((\mathrm{a}^1; \mathrm{b}^3; ((\mathrm{c}^1; \delta^1) \|\|\| (\mathrm{d}^2; \delta^2))) [] (\mathrm{e}^1; \mathrm{f}^4; stop)) >> ((\mathrm{g}^1; \mathrm{h}^4; \mathrm{j}^1; \delta^1) \|\|\| (\mathrm{k}^2; \delta^2)))$ $FIFO(c,c') = (\{c,c'\} = \{1,2\})$, $TC(b) = \{1,2,3,4\}$
$\mathbf{T}_1(b,z) := (((\mathrm{a}^1; send_3!z.1; \mathbf{rec}_3!z.2; \mathrm{c}^1; \delta) [] (\mathrm{e}^1; send_4!z.3; stop))$ $>> (sync_2!z.4; \mathrm{g}^1; send_4!z.5; \mathbf{rec}_4!z.6; \mathrm{j}^1; \delta))$ $\mathbf{T}_2(b,z) := (\mathbf{rec}_3!z.2; \mathrm{d}^2; sync_1!z.4; \mathrm{k}^2; \delta)$ $\mathbf{T}_3(b,z) := (\mathbf{rec}_1!z.1; \mathrm{b}^3; ((send_1!z.2; \delta) \|\|\| (send_2!z.2; \delta)))$ $\mathbf{T}_4(b,z) := ((\mathbf{rec}_1!z.3; \mathrm{f}^4; stop) [> (\mathbf{rec}_1!z.5; \mathrm{h}^4; send_1!z.6; \delta))$

Table 14: An example implementation of sequential composition

executed after c' exits both parts, implying that the message is also sent if b_2 is executed (Brinksma and Langerak, 1995). Therefore $\neg TC_{c''}^+(b_1)$, for the message reports completion of b rather than b_1.

If c' is also the only ending component of b_2, c'' needn't be a terminating component of b_2 and interprets the reception as a disruption of $\mathbf{T}_c(b_2, z)$ (see the example in Table 16). In the opposite case, components (including c') must first terminate b_2 (if currently active), before sequentially proceeding to synchronization upon completion of b (Brinksma and Langerak, 1995). However, the second solution is only applicable if for every $c'' \in ((TC(b) \cap PC(b)) \setminus \{c'\})$, all the initial actions of $\mathbf{Term}_{c''}(b_2, z)$ are also receptions of messages sent by c'. The simplest way to secure that is to require $|PC(b)| = 2$, as it is in (Brinksma and Langerak, 1995).

Initial (decision-making) terminations of b_2 are no longer forbidden, as they are in (Brinksma and Langerak, 1995). That is because each initial δ is now controlled by c' and guards termination of $\mathbf{T}_c(b, z)$ for any other $c \in PC(b)$.

4.6 Implementation of parallel composition

Parallel composition of service parts (Table 17), regardless of the extent of their synchronization, introduces no additional protocol messages. Each component simply executes in parallel its local implementations of individual service parts, locally synchronized as specified by the parallel composition operator. To minimize communication costs, we allow separate termination optimization of individual parallel service parts.

If all components communicate synchronously (as for example in process "First" in Table 22), the *cross-cut theorem* (Eijk, 1990) for re-grouping of parallel processes applies: One may pretend that it is not that protocol interactions in the implementations of b_1 and b_2 share internal system channels and differ only in the message contents, but that the message contents is also a part of the channel name, i.e. that the two implementations use different system channels. Hence as far as synchronization between the service actions in b_1 and b_2 is ignored, the distributed server runs the parallel service parts' implementations independently. Moreover, synchronization upon a service action between the parallel service parts is at sole discretion of the component executing the action, i.e. a local matter.

(Kant et al., 1996; Brinksma and Langerak, 1995; Kapus-Kolar, 1997) *erroneously* (Kapus-Kolar, 1999) presume that the cross-cut theorem also applies to asynchronous communication. The simplest way to establish virtual independence between the implementations of b_1 and b_2 in the presence of FIFO channels is to require that there is no asynchronously communicating pair of components belonging to $(PC(b_1) \cap PC(b_2))$. (For example, for process "First" in Table 22 it is crucial that $\neg FIFO(1,2)$.) If the parallel composition is pure interleaving, however, the above restriction is not necessary (as for example in process "Second" in Table 22). That is because the components are known to always be able to receive protocol messages in the global order in which they have been sent, in both the implementation of b_1 and of b_2 (Kapus-Kolar, 1998)

4.7 Implementation of hiding and renaming

Table 13 indicates that the only property of an action s^c that is relevant for mapping $\mathbf{T}$ is its location. Hence hiding commutes with $\mathbf{T}$ and its implementa-

$e = b = b_1[> b_2$	
$SC_c(b) = (SC_c(b_1) \vee SC_c(b_2))$	$EC_c(b) = (EC_c(b_1) \vee EC_c(b_2))$
$PC_c(b) = (PC_c(b_1) \vee PC_c(b_2))$	$ST(b_1) = ST(b_2) = ST(b)$
$IT(b) = (IT(b_1) \vee IT(b_2))$	$DT(b) = ((EC(b_1) \neq \phi) \vee IT(b_2) \vee DT(b_2))$
$TC_c^+(b_1) = EC_c(b_1)$; if $(EC(b) = PC(b_1))$ then $TC_c^+(b_2) = PC_c(b_1)$ else $TC_c^+(b_2) = TC_c(b)$ endif	
$\exists c' : ((PC(b_1) = SC(b_2) = \{c'\}) \wedge$ $((EC(b) \subseteq \{c'\}) \vee (((TC(b) \cap PC(b)) \setminus \{c'\}) = \phi) \vee (\lvert PC(b)\rvert = 2)))$	
$\mathbf{T}_c(e,z) :=$ if $(EC(b_1) = \phi)$ then if $PC_c(b_1)$ then $(b_1[> \mathbf{Term}_c(b_2,z))$ else $\mathbf{Term}_c(b_2,z)$ endif else if $PC_c(b_1)$ then $((b_1[> \mathbf{Term}_c(b_2,z)) >> \mathbf{Send}_c((TC(b) \cap PC(b)), z.Z(b_1)))$ else if $TC_c(b)$ then if $(EC(b_2) = PC(b_1))$ then $(\mathbf{T}_c(b_2,z)[> \mathbf{Rec}_c(PC(b_1), z.Z(b_1)))$ else $(\mathbf{Rec}_c(PC(b_1), z.Z(b_1))[]$ $(\mathbf{Term}_c(b_2,z)$ if $(EC(b_2) \neq \phi)$ then $>> \mathbf{Rec}_c(PC(b_1), z.Z(b_1))$ endif)) endif else $\mathbf{T}_c(b_2,z)$ endif endif endif	

Table 15: The specific rules for disabling

$b = ((\mathrm{a}^1; \mathrm{b}^1; \delta^1)[> (\delta^1[](\mathrm{d}^1; ((\mathrm{e}^2; \delta^2) \lvert\lvert\lvert (\mathrm{f}^3; \delta^3))) >> (\mathrm{g}^1; \delta^1))))$
$\forall \{c, c'\} : FIFO(c, c')$, $TC(b) = \{1, 3\}$
$\mathbf{T}_1(b,z) := (((\mathrm{a}^1; \mathrm{b}^1; \delta)[> (\delta[](\mathrm{d}^1; ((send_2!z.2; \delta) \lvert\lvert\lvert (send_3!z.2; \delta)) >> ((rec_2!z.3; \delta) \lvert\lvert\lvert (rec_3!z.3; \delta))$ $>> (\mathrm{g}^1; \delta))))$ $>> (send_3!z.1; \delta))$
$\mathbf{T}_2(b,z) := (rec_1!z.2; \mathrm{e}^2; send_1!z.3; stop)$
$\mathbf{T}_3(b,z) := ((rec_1!z.2; \mathrm{f}^3; send_1!z.3; stop)[> (rec_1!z.1; \delta))$

Table 16: An example implementation of disabling

tion is trivial (Table 18). To enforce the commuting for renaming, we require that all renamings are local to individual server components (Table 19). An example implementation of hiding and renaming is given in Table 22.

4.8 Implementation of process definition and instantiation

An explicit process p is implemented at a component c as an explicit process $p_c(n)$ (Table 20), where formal parameter n carries the process instance identifier (Table 21). For the main process *Server* we presume that it might be necessary for its termination to be a common action of the server environment and all the server components. An example implementation of multiple concurrent instances of a process is process "Second" in Table 22. Note that in the example, simplification of all $z.Z(b)$ into $Z(b)$, suggested for the example in Table 7, would result in an erroneous protocol.

5 Discussion and Conclusions

We have proposed a correctness-preserving transformation for functionality decomposition based on specifications written in Basic LOTOS (Bolognesi and Brinksma, 1987), the core sublanguage of the standard specification language LOTOS. Given a specification of the required external behaviour (the expected service) of a system and a partitioning of the specified service actions among the system components, the transformation derives behaviour of individual components implementing the service. A correctness proof is provided in (Kapus-Kolar, 1998).

$e = b =$ **hide** G **in** b_1	
$SC_c(b) = SC_c(b_1)$	$EC_c(b) = EC_c(b_1)$
$PC_c(b) = PC_c(b_1)$	$IT(b) = IT(b_1)$
$ST(b_1) = ST(b)$	$DT(b) = DT(b_1)$
$TC_c(b_1) = TC_c(b)$	
$\mathbf{T}_c(e,z) :=$ **hide** $Proj_c(G)$ **in** $\mathbf{T}_c(b_1,z)$	

Table 18: The specific rules for hiding

Our algorithm enhances and integrates the algorithms of (Kant et al., 1996; Brinksma and Langerak, 1995). As the two algorithms are themselves a synthesis of the earlier similar approaches, and thoroughly compared to them, in the following we only compare our algorithm to the two algorithms.

- Unlike (Brinksma and Langerak, 1995), our algorithm is applicable to *multi-party* servers.
- It is applicable to servers with *both synchronous and asynchronous inter-component channels*, while (Kant

$e = b = b_1 \vert G \vert b_2$	
$SC_c(b) = (SC_c(b_1) \vee SC_c(b_2))$	$EC_c(b) = (EC_c(b_1) \vee EC_c(b_2))$
$PC_c(b) = (PC_c(b_1) \vee PC_c(b_2))$	$IT(b) = (IT(b_1) \wedge IT(b_2))$
$ST(b_1) = ST(b_2) = true$	$DT(b) = (DT(b_1) \vee DT(b_2))$
$TC_c^+(b_1) = TC_c^+(b_2) = TC_c(b)$	
$(G = \phi) \vee \neg\exists\{c, c'\} \subseteq (PC(b_1) \cap PC(b_2)) : FIFO(c, c')$	
$\mathbf{T}_c(e, z) := (\mathbf{T}erm_c(b_1, z) \vert \mathbf{P}roj_c(G) \vert \mathbf{T}erm_c(b_2, z))$	

Table 17: The specific rules for parallel composition

$Server := (First \vert [a^1, c^2] \vert Second)$ $First :=$ **hide** b^2 **in** $((((a^1; \delta^1) \vert\vert\vert (b^2; \delta^2)) >> (c^2; \delta^2)) \vert [b^2] \vert (d^1; b^2; \delta^2))$ $Second := ((ren\ A^1 \rightarrow a^1\ B^3 \rightarrow b^3\ C^2 \rightarrow c^2\ in\ Proc) \vert\vert\vert (ren\ A^1 \rightarrow x^1\ B^3 \rightarrow y^3\ C^2 \rightarrow z^2\ in\ Proc))$ $Proc := (((A^1; \delta^1) \vert\vert\vert (B^3; \delta^3)) >> (C^2; \delta^2))$ $FIFO(c, c') = (\{c, c'\} \neq \{1, 2\})$
$Server_1 := (First_1(1) \vert [a^1] \vert Second_1(2))$ $First_1(z) := ((a^1; sync_2!z.1; \delta) \vert\vert\vert (d^1; sync_2!z.2; \delta))$ $Second_1(z) := ((ren\ A^1 \rightarrow a^1\ in\ Proc_1(z.1)) \vert\vert\vert (ren\ A^1 \rightarrow x^1\ in\ Proc_1(z.2)))$ $Proc_1(z) := (A^1; sync_2!z.1; \delta)$
$Server_2 := (First_2(1) \vert [c^2] \vert Second_2(2))$ $First_2(z) :=$ **hide** b^2 **in** $((b^2; sync_1!z.1; c^2; \delta) \vert [b^2] \vert (sync_1!z.2; b^2; \delta))$ $Second_2(z) := ((ren\ C^2 \rightarrow c^2\ in\ Proc_2(z.1)) \vert\vert\vert (ren\ C^2 \rightarrow z^2\ in\ Proc_2(z.2)))$ $Proc_2(z) := (((sync_1!z.1; \delta) \vert\vert\vert (rec_3!z.1; \delta)) >> (C^2; \delta))$
$Server_3 := Second_3(2)$ $Second_3(z) := ((ren\ B^3 \rightarrow b^3\ in\ Proc_3(z.1)) \vert\vert\vert (ren\ B^3 \rightarrow y^3\ in\ Proc_3(z.2)))$ $Proc_3(z) := (B^3; send_2!z.1; \delta))$

Table 22: An example implementation of parallel composition, hiding, renaming and process instantiation

$e = b =$ **ren** R **in** b_1	
$SC_c(b) = SC_c(b_1)$	$EC_c(b) = EC_c(b_1)$
$PC_c(b) = PC_c(b_1)$	$IT(b) = IT(b_1)$
$ST(b_1) = ST(b)$	$DT(b) = DT(b_1)$
$TC_c(b_1) = TC_c(b)$	
$((u_1^c \rightarrow g) \in R) \Rightarrow \exists u_2 : (g = u_2^c)$	
$\mathbf{T}_c(e, z) :=$ **ren** $\mathbf{P}roj_c(R)$ **in** $\mathbf{T}_c(b_1, z)$	

Table 19: The specific rules for renaming

$e = (p := b_1)$	
$SC_c(p) = SC_c(b_1)$	$EC_c(p) = EC_c(b_1)$
$PC_c(p) = PC_c(b_1)$	$IT(p) = IT(b_1)$
$ST(b_1) = ST(p)$	$DT(p) = DT(b_1)$
$TC_c^+(b_1) = TC_c(p)$	
$\mathbf{T}_c(e, z) := (p_c(n) := \mathbf{T}erm_c(b_1, n))$	

Table 20: The specific rules for process definition

et al., 1996) only supports asynchronous communication. As such, our algorithm has wide applicability (see the Introduction) and is also suitable for hardware/software co-design.

- The algorithm *corrects* an error identified in (Brinksma and Langerak, 1995) and several identified in (Kant et al., 1996). It is also more general, in the sense that it supports *implementation of decision-making terminations.*
- The algorithm provides means for the generation of *more efficient protocols* (with less intercomponent communication), based on the following observation: If there are two consecutive service parts b_1 and b_2, the only server components that really must detect the termination of b_1 are those executing its ending actions. Other participants of b_1 can as well conclude its execution by inaction that is later disrupted by actions announcing execution of b_2, for it is the start of b_2 - not the formal termination of b_1 - that is relevant to the service users. Even if there is no b_2 following b_1, it might still be more efficient for a non-ending participant of b_1 not to care about its termination, but rather conclude its part of b_1 with inaction later disrupted by termination-signalling messages from the ending participants of b_1.
- The algorithm is *more flexible*, for it allows one to employ the above communication-reduction principle to an extent best meeting his/her various optimization criteria, reduction of inter-component communication often being just one of them. If the principle is only employed where mandatory for protocol correctness,

$e = b = p$	
$SC_c(b) = SC_c(p)$	$EC_c(b) = EC_c(p)$
$PC_c(b) = PC_c(p)$	$DT(b) = DT(p)$
if $(p = Server)$ then $ST(p) = true$ else $ST(p) = (ST(p) \vee ST(b))$	
if $(p = Server)$ then $TC_c(p) = true$ else $TC_c(p) = TC_c(b)$	
$\mathbf{T}_c(e, z) := p_c(z.Z(b))$	

Table 21: The specific rules for process instantiation

the algorithm reduces to a corrected version of (Kant et al., 1996; Brinksma and Langerak, 1995) adapted for multi-party servers with synchronous and/or asynchronous inter-component channels.

It seems that one should prefer our algorithm to (Kant et al., 1996; Brinksma and Langerak, 1995), though it could be further improved by more exact computation of service specification subexpression attributes and restrictions, that would widen its applicability and/or increase the efficiency of the derived protocols. Other items *for further study* are the same as for the two former algorithms:

- introduction of inter-component co-ordination schemes that would render the various restrictions on the service specification structure unnecessary, i.e. allow *distributed decision-making*, as for a very limited setting suggested in (Langerak, 1990). There is presently no adequate solution that would not ruin the compositionality of the algorithm, thereby making the service/protocol relationship difficult to understand.
- extension to service actions with *data parameters* and to *timed service actions*. Our experience (Kapus-Kolar, 1991a,b) shows that it is typically possible to convey data and timing information piggybacked in the protocol messages already present if data and time are ignored, i.e. in the messages introduced by the above presented algorithm. Of course, provided that the messages are sent at appropriate points of service execution. Hence again the message-scheduling flexibility of our algorithm proves convenient, particularly in the presence of real-time requirements and protocol channels with substantial transit delay.
- generalization to *unreliable protocol channels*, though it presently seems that it would be better to solve the problem *below* the application layer of the system. For recovery from errors requires returning to previous states, but the concept of an explicit state is not defined in LOTOS.

The algorithm as it is now is useful as a set of *hints on how to systematically design correct and efficient distributed service implementations*. To complete the work, it would be desirable to implement the algorithm within a CAD tool. The mapping itself is trivial to implement, but the communication optimization part is complicated, if one wants to give the algorithm the best possible performance. Even optimization criteria are not well known; it would be convenient to have them derived automatically from a more detailed specification of the system and the service, particularly from requirements regarding action parameters and quantitative timing, and the information on the invocation probability for individual service parts. Thus we decided to postpone the implementation of the algorithm till completion of a thorough study on the subject.

References

[1] Bochmann, G. v., Gotzhein, R., Deriving Protocol Specifications from Service Specifications, in *Proc. ACM SIGCOMM'86 Symp.*, ACM, 1986, 148-156.

[2] Bolognesi, T., Brinksma, E., Introduction to the ISO Specification Language LOTOS, *Computer Networks & ISDN Systems 14(1)*, 25-59 (1987).

[3] Brinksma, E., Langerak, R., Functionality Decomposition by Compositional Correctness Preserving Transformation, *SACJ/SART 13*, 2-13 (1995).

[4] Eijk, P. v., Tools for LOTOS Specification Style Transformation, in *Formal Description Techniques II* (S. T. Vuong, ed.), North-Holland, 1990, 43-51.

[5] Kant, C., Higashino, T., Bochmann, G. v., Deriving Protocol Specifications from Service Specifications Written in LOTOS, *Distributed Computing 10*, 29-47 (1996).

[6] Kapus-Kolar, M., Deriving Protocol Specifications from Service Specifications Including Parameters, *Microprocessing and Microprogramming 32*, 731-738 (1991).

[7] Kapus-Kolar, M., Deriving Protocol Specifications from Service Specifications with Heterogeneous Timing Requirements, in *Proc. 3rd IEE Int. Conf. on Software Engineering for Real-Time Systems*, IEE, London, 1991, 266-270.

[8] Kapus-Kolar, M., Employing Disruptions for More Efficient Functionality Decomposition in LOTOS, in *Proc. 22nd EUROMICRO Conf.*, IEEE Computer Society, 1997, 464-471.

[9] Kapus-Kolar, M., *Employing Disruptions for More Efficient Functionality Decomposition in LOTOS*, Technical Report 7878, Jožef Stefan Institute, Ljubljana, 1998.

[10] Kapus-Kolar, M., Comments on Deriving Protocol Specifications from Service Specifications Written in LOTOS, to appear in *Distributed Computing 12(4)*, 1999.

[11] Kapus-Kolar, M., Rugelj, J., Bonač, M., Deriving Protocol Specifications from Service Specifications, in *Proc. 9th IASTED Int. Symp. Applied Informatics* (M. H. Hamza, ed.), Acta Press, Anaheim-Calgary-Zürich, 1991, 375-378.

[12] Langerak, R., Decomposition of Functionality: A Correctness-Preserving LOTOS Transformation, in *Protocol Specification, Testing and Verification X* (L. Logrippo, R. Probert, H. Ural, eds.), North-Holland, 1990, 229-242.

[13] Saleh, K., Synthesis of Communication Protocols: An Annotated Bibliography, *Computer Communication Review 26(5)*, 40-59 (1996).

[14] Turner, K. J. (ed.), *Using Formal Description Techniques - An Introduction to ESTELLE, LOTOS and SDL*, John Wiley, New York, 1993.

[15] *WELL - World-wide Environment for Learning LOTOS*, "http://www.cs.stir.ac.uk/~kjt/research/-well/well.html".