Deriving Self-Stabilizing Protocols for Services Specified in LOTOS

Monika Kapus-Kolar
Jožef Stefan Institute, POB 3000, SI-1001 Ljubljana, Slovenia
monika.kapus-kolar@ijs.si

Keywords: distributed service implementation, automated protocol derivation, LOTOS

Received: September 6, 2002

A transformation is proposed which, given a specification of the required external behaviour of a distributed server and a partitioning of the specified service actions among the server components, derives a behaviour of individual components implementing the service. The adopted specification language is close to Basic LOTOS. Unlike in other protocol derivation algorithms based on LOTOS-like languages, distributed conflicts in the given service are allowed, and resolved by self-stabilization of the derived protocol.

1 Introduction

In top-down distributed systems design, one of the most difficult transformations is decomposition of a process into a set of co-operating subprocesses. Such a transformation is considered correct if it preserves, to the required degree, those actions of the process which are considered essential. Such actions are often referred to as the *service* that the process offers to its environment, i.e. the process is observed in the role of a *server*.

A service consists of atomic service actions, of which the most important are *service primitives*, i.e. atomic interactions between the server and its users, executed in *service access points*. In addition, one might decide to introduce some *hidden service actions*, to represent various important events within the server.

When decomposing a server, the first step is to decide on its internal architecture. It can be represented as a set of *server components* (e.g. one component per service access point), with channels for their communication. We shall assume that all components are on the same hierarchical level, for a multi-level architecture can always be obtained by gradual decomposition.

The next step is to assign individual service actions to individual server components, paying attention to the location and capability of components.

The final step is to specify details of the inter-component communication, i.e. to derive an efficient *protocol* implementing the service, where efficiency is measured in terms of the communication load. While the first two steps require creative decisions, protocol derivation can be *automated*. Given a formal specification of the architecture of a server, of its service and of its distribution, one can mechanically decide on the protocol exchanges necessary to implement the specified distributed causal relations and choices between service actions.

A protocol is typically much more complex than the service it implements. Besides, one usually does not care about the exact nature of an otherwise satisfactory protocol. Therefore, algorithms for automated protocol derivation are most welcome! They automate exactly that part of server decomposition which is the most difficult for a human, requiring simultaneous reasoning about the numerous co-operating parties.

Even if one decides for automated protocol derivation, it remains possible to strongly influence the resulting protocol, by introducing dummy hidden service actions. For example, introducing a pair of consecutive service actions executed by two different server components introduces a protocol message from the first to the second component. Prefixing each of a set of alternatives by a service action at a particular component makes the choice local to the component. In other words, instead of spending time on protocol design, one should rather concentrate on detailed service design, specifying all important dynamic decisions as explicit service actions [16]. By various allocations of the actions to server components, service implementations with various degrees of centralization are obtained.

A prerequisite for automated protocol derivation is that the service is specified in a formal language. It is desirable that the derived behaviours of individual server components are specified in the same language as the service, so that the same algorithm can be used for further decomposition.

It is desirable that a protocol derivation algorithm is to a large extent compositional, so that it can cope with large service specifications, provided that they are well structured. Moreover, a compositional algorithm reflects the service structure in the derived protocol specification, increasing the service designers' confidence into the automatically generated implementation.

It is difficult to construct a general protocol derivation algorithm with high-quality results and low complexity. Typical algorithms work on small classes of service specifications.

Protocol synthesis has been subject to intensive research since the middle eighties. An exhaustive survey can be found in [26], so we provide no systematic review of the

existing methods and refer to them only where necessary for comparison with the proposed solutions.

The protocol derivation transformation proposed in our paper is an enhancement of that in [10]. As in [10], we assume that a server consists of an *arbitrary fixed number of components* exchanging the necessary protocol messages *asynchronously*, over reliable, unbounded, initially empty first-in-first-out (FIFO) channels with a finite, but unknown transit delay. The adopted specification language is a syntactically simplified sublanguage of LOTOS [7, 2], a standard process-algebraic language intended primarily for specification of concurrent and reactive systems. Service primitives are not allowed to carry parameters, neither do we allow specification of real-time constraints. However, the principles for enhancing a basic protocol derivation method to cope with data and real time are well known [11, 12, 23].

For a service containing distributed conflicts, a precise implementation takes care that they never cause divergence in service execution. Firstly one should try to make all conflicts local to individual components, by inserting auxiliary hidden service actions, but that is acceptable only as long as no external service choice is undesirably converted into an internal server choice. For the remaining distributed conflicts, divergence prevention requires extensive inter-component communication [9, 20, 21]. Although even such protocols can be derived compositionally [17], the communication costs they introduce are usually acceptable only if exact service implementation is crucial or during the periods when server users compete strongly for the service. In a typical situation, the probability of a distributed conflict is so low that divergence should rather be resolved than prevented.

In LOTOS, there are two process composition operators allowing specification of service actions in distributed conflict, the operator of choice and the operator of disabling. In [10], only local choice is allowed. For disabling, the derived protocols are supposed to *self-stabilize after divergence*, but the proposed solution is not correct in the general case [15]. Besides, [10] has problems with implementation of parallel composition [15]. In an unpublished response to [15], Bochmann and Higashino proposed some solutions for the problems, but have not integrated them into their protocol derivation algorithm and have not been able to specify the solution for disabling in LOTOS.

We specify self-stabilization upon disabling purely in the adopted LOTOS-like language, and also suggest how to implement distributed choice. Further improvements over [10] are implementation solutions for processes with successful termination as a decisive event, for processes which might enter inaction without first declaring successful termination, for combining terminating and non-terminating alternatives, for process disabling with multiple initiators, and for interaction hiding and renaming. The proposed solutions can be seen also as an improvement over [3], another algorithm for the purpose in which we have identified a bug [15].

Name of the construct	Syntax
Specification	$w ::= \mathbf{spec}\ b\ \mathbf{where}\ D\ \mathbf{endspec}$
	$D ::= \underline{\text{set}}\ \underline{\text{of}}\ d$
Process definition	$d ::= p(x)\ \mathbf{is}\ b \mid p\ \mathbf{is}\ b$
Process name	$p ::= \mathit{ProcIdentifier}$
Parameter name	$x ::= \mathit{ParIdentifier}$
Behaviour	$b ::=$
Inaction	$\mathbf{stop}$
Successful termination	$\mid \delta$
Sequential composition	$\mid b_1 \gg b_2$
Action prefix	$\mid a; b_2$
Choice	$\mid b_1 [] b_2$
Parallel composition	$\mid b_1 \mid [G] \mid b_2$
Disabling	$\mid b_1 [> b_2$
Hiding	$\mid \mathbf{hide}\ G\ \mathbf{in}\ b_1\ \mathbf{endhide}$
Renaming	$\mid \mathbf{ren}\ R\ \mathbf{in}\ b_1\ \mathbf{endren}$
Process instantiation	$\mid p(v) \mid p$
	$G ::= \underline{\text{set}}\ \underline{\text{of}}\ g$
Interaction gate	$g ::= s \mid h$
Data value	$v ::= \underline{\text{term}}\ \underline{\text{of}}\ \underline{\text{type}}\ n^*$
Index	$n ::= 1 \mid 2$
	$R ::= \underline{\text{set}}\ \underline{\text{of}}\ r$
Gate renaming	$r ::= g'/g$
Action	$a ::= \mathbf{i} \mid s \mid h \mid ho$
Service primitive	$s ::= u^c$
Service-primitive type	$u ::= \mathit{PrimIdentifier}$
Server component	$c ::= \mathit{CompIdentifier}$
Auxiliary gate	$h ::= \mathbf{s}^c_{c'} \mid \mathbf{r}^c_{c'} \mid \mathbf{a}^n_{c'} \mid \mathbf{b}_{c'} \mid \mathbf{t}$
Data offer	$o ::= !v \mid ?v \mid ?x{:}v$

Table 1: The adopted specification language

The paper is organized as follows. Section 2 introduces the adopted specification language and its service specification sublanguage, some building blocks for the derived protocol specifications, and the adopted protocol correctness criterion. Section 3 describes the adopted principles of protocol derivation. The derivation is guided by various service specification attributes. In Section 4, we introduce rules for attribute evaluation and suggest how to obtain a well-formed service specification. Section 5 comprises discussion and conclusions.

2 Preliminaries

2.1 Specification language and its service specification sublanguage

The language employed, defined in Table 1 in a Backus-Naur-like form, is an abstract representation of some LOTOS constructs, in the exclusive setting of the protocol derivation problem. Not shown in the table are parentheses for control of parsing, the syntax for sets, and shorthands.

A b denotes a behaviour, i.e. a process exhibiting it, for instance a server as a whole, an individual server component, a service part or some other partial server behaviour. For a particular server, let $\mathcal{C}$ denote the universe of its components.

$\mathbf{spec}\ \varepsilon\ \mathbf{where}\ D\ \mathbf{endspec} = \mathbf{spec}\ \delta\ \mathbf{where}\ D\ \mathbf{endspec}$	
$\varepsilon \lvert[G]\rvert b = b \lvert[G]\rvert \varepsilon = b$	$a;\varepsilon = a;\delta$
$\varepsilon \gg b = b \gg \varepsilon = b$	$\varepsilon; b = b$
$\mathbf{hide}\ G\ \mathbf{in}\ \varepsilon\ \mathbf{endhide} = \varepsilon$	$\mathbf{ren}\ R\ \mathbf{in}\ \varepsilon\ \mathbf{endren} = \varepsilon$

Table 2: Absorption rules for ε

stop denotes inaction of the specified process.

δ denotes successful termination.

In some cases, the protocol derivation mapping defined below introduces an ε specifying execution of no actions. ε is similar to δ, because execution of no actions is successful by definition. With the help of the absorption rules in Table 2, it will be possible to make the derived specifications free of ε.

i denotes an anonymous internal action of the specified process. Besides internal actions, processes execute interactions with their environment. Such an external action is primarily denoted by the interaction gate on which it occurs. If it is a service primitive, it is specified as a u^c and denotes a type u interaction between server component c and a service user. If it is an action on an auxiliary gate h, it might be associated with a data offer o, that has to match with the data offer of the process environment. The only data that our processes can handle are strings of zero or more elements 1 and/or 2.

A component c can send messages to another component c' over gate $\mathbf{s}_{c'}^{c}$, while c' receives them over gate $\mathbf{r}_{c}^{c'}$. For specific purposes, c' will sometimes call the gate $\mathbf{a}_{c}^{n}$ (accept), where n will be a partial context identifier. If c' is unable to immediately handle a message received on gate $\mathbf{r}_{c}^{c'}$, it will store it into a FIFO buffer and subsequently claim it on an internal gate $\mathbf{b}_{c}$. Gate **t** will always be an internal gate of a server component, serving for hidden interaction of its parts.

A data offer "$!v$" denotes exactly the data value specified by the term v. A data offer "$?x : v$" or "$?v$" denotes any data value which has a prefix specified by v. When the interaction occurs, one of the values legal for the data offer is selected, and if variable x is specified, stored into it for future use.

"$b_1 \gg b_2$" denotes a process first behaving as b_1, and after its successful termination as b_2, where δ of b_1 is interpreted in "$b_1 \gg b_2$" as **i**. "$a; b_2$" is the special case of the sequential composition where b_1 is an individual action, so that no **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. Sometimes we will use "[]" as a prefix operator, where choice from an empty set of processes is equivalent to **stop**.

"$b_1 \lvert[G]\rvert b_2$" denotes parallel composition of processes b_1 and b_2, where G specifies the degree and form of their synchronization. An action on a gate listed in G or a δ can only be executed as a common action of the two processes, while the processes execute other actions independently. The usual shorthand for "$\lvert[]\rvert$" is "$\lvert\lvert\lvert$". Sometimes we will use "$\lvert\lvert\lvert$" as a prefix operator, where parallel composition of an empty set of processes specifies an ε.

No.	e
(1)	w::= $\mathbf{spec}\ b\ \mathbf{where}\ D\ \mathbf{endspec}$
(2)	$d ::= p\ \mathbf{is}\ b$
(3)	$b ::= \mathbf{stop}$
(4)	$b ::= \delta$
(5)	$b ::= b_1 \gg b_2$
(6)	$b ::= a; b_2$
(7)	$b ::= b_1 \lvert[S]\rvert b_2$
(8)	$b ::= b_1 [] b_2$
(9)	$b ::= b_1 [> b_2$
(10)	$b ::= \mathbf{hide}\ S\ \mathbf{in}\ b_1\ \mathbf{endhide}$
(11)	$b ::= \mathbf{ren}\ R\ \mathbf{in}\ b_1\ \mathbf{endren}$
(12)	$b ::= p$
(13)	$a ::= s \mid \mathbf{i}$
	$S ::= \underline{\text{set}}\ \underline{\text{of}}\ s$
	$r ::= u_2^c / u_1^c$

Table 3: Service specification sublanguage

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

"$\mathbf{hide}\ G\ \mathbf{in}\ b_1\ \mathbf{endhide}$" denotes a process behaving as b_1 with its actions on the gates listed in G hidden from its environment. For the environment, the hidden actions are equivalent to **i**.

"$\mathbf{ren}\ R\ \mathbf{in}\ b_1\ \mathbf{endren}$" denotes a process behaving as b_1 with its visible gates (and thereby the actions on them) renamed as specified in R, where in an r, the first and the second item respectively define the new and the old name.

Explicit processes can be defined and instantiated, possibly with an input parameter. 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, gate instantiation can be expressed as renaming of the gates on which a process is originally defined applied to the particular process instance.

A specification w defines a behaviour b and the processes instantiated in it, except for the processes predefined in Section 2.2. If D is empty, "$\mathbf{where}\ D$" may be omitted. If it is a service specification (Table 3), then 1) any specified action must be a service primitive or an **i**, 2) gate renaming is allowed only locally to individual server components, and 3) all the explicitly specified processes must be without parameters. Some rows in Table 3 are numbered, so that the corresponding rows in some of the remaining tables can refer to them. In all our example service specifications, every **i** and every δ is furnished with a superscript denoting the server component responsible for it.

The relation used throughout the paper for judging equivalence of behaviours is *observational equivalence* "$\approx$" [2], i.e. we are interested only into the external behaviour of processes, that is in the actions which they make available for synchronization with their environment (all actions except **i** and actions transformed into **i** by hiding).

2.2 Some building blocks for protocol specifications

The contribution of our paper lies in functions for generating protocol specifications in the proposed language. These specifications will be based on some characteristic patterns, for generation of which we define some auxiliary functions (Table 4).

$\mathbf{S}_c(C,v) := \|\|\|_{c' \in (C \setminus \{c\})} \mathbf{s}^c_{c'}!v$
$\mathbf{R}_c(C,v) := \|\|\|_{c' \in (C \setminus \{c\})} \mathbf{r}^c_{c'}!v$
$\mathbf{E}_c(C,C',v) :=$ (if $(c \in C)$ then $\mathbf{S}_c(C',v)$ else ε endif $\|\|\|$ if $(c \in C')$ then $\mathbf{R}_c(C,v)$ else ε endif)
$\mathbf{P}_c(S) := \{u^c \mid (u^c \in S)\}$
$\mathbf{P}_c(R) := \{(u'^c/u^c) \mid ((u'^c/u^c) \in R)\}$

Table 4: Auxiliary specification-generating functions

$\mathbf{S}_c(C,v)$ generates a specification of parallel sending of protocol message v from component c to each member of C other than c. Likewise, $\mathbf{R}_c(C,v)$ specifies parallel receiving of v at c from each member of C other than c.

$\mathbf{E}_c(C,C',v)$ specifies exchange of message v in such a way that each component in C' receives it from every component in C other than itself.

$\mathbf{P}_c(S)$ and $\mathbf{P}_c(R)$ are projection functions. $\mathbf{P}_c(S)$ extracts from S the service primitives belonging to component c, while $\mathbf{P}_c(R)$ extracts from R the renamings of such primitives.

We also assume that there are three predefined processes. Processes "Loop" and "Loop(v)" execute an infinite series of "g" or "g?v" actions, respectively. Shorthands for instantiation of the processes on a gate g for a prefix v are "$\mathrm{Loop}(g)$" and "$\mathrm{Loop}(g?v)$", respectively.

Process "FIFO(v)" is an unbounded FIFO buffer ready to store messages with prefix "v" and to terminate whenever empty. A shorthand for instantiaton of the process on an input gate g_1 and an output gate g_2 for a prefix v is "$\mathrm{FIFO}(g_1,g_2,v)$". To specify that a $\mathrm{FIFO}(g_1,g_2,v)$ should accept all kinds of messages, one sets v to an empty string, that we denote by ε. Such are the buffers pairwise connecting server components. They constitute the communication medium, defined as

Medium **is** $\|\|\|_{c \neq c'} \mathrm{FIFO}(\mathbf{s}^c_{c'}, \mathbf{r}^{c'}_c, \varepsilon)$

2.3 Protocol correctness criterion

Given a service behaviour b, we derive a b_c for each individual component c. The protocol must satisfy the minimal correctness criterion that every protocol message sent is also received. We further expect that in the absence of distributed conflicts, the server behaves towards its users precisely as required (see Table 5). Note that "$\approx (b \gg \delta)$" might also be sufficient, because successful termination of a distributed server, as an act of multiple server components, does not qualify as one of the regular service actions, i.e. service actions assigned to individual components.

If b contains distributed conflicts, precise service execution is expected only for those server runs which do not reveal any of the conflicts. When divergence in service execution occurs, the server should continue to support only the direction of service execution with the highest pre-assigned priority, while the directions competing with it must be abandoned as quickly as possible.

$(\mathrm{Service} \approx b) \vee ((\|\mathcal{C}\| > 1) \wedge (\mathrm{Service} \approx (b \gg \delta)))$
where $\mathrm{Service} =$ **hide** G **in** $(\|\|\|_{c \in \mathcal{C}} b_c) \|[G]\| \mathrm{Medium}$ **endhide**
$G = \cup_{c \neq c'} \{\mathbf{s}^c_{c'}, \mathbf{r}^{c'}_c\}$

Table 5: Precise service implementation

For a "$b_1 [> b_2$", it is appropriate that b_2 has a higher priority than b_1. We adopt this arrangement also for "$b_1 [] b_2$". There are, however, two exceptions. If the server components responsible for the start of b_2 manage to agree on successful termination of b_1 before b_2 starts, b_2 must be abandoned. In the case of "$b_1 [] b_2$", b_2 must be abandoned already when the components manage to agree on the start of b_1.

3 Principles of protocol derivation

3.1 Service attributes and the concept of a well-formed service specification

When mapping a service specification subexpression into its counterparts at individual server components, one refers to its various attributes. A subexpression attribute reveals some property of the subexpression itself or some property of the context in which it is embedded. Computation of service attributes is discussed in Section 4.1.

There is always a dilemma whether to conceive a very general mapping, i.e. a mapping with very few restrictions on the attributes, or a simple mapping with a very restricted applicability. We take the following pragmatic approach.

Above all, we try to avoid restrictions on the specification style (see [28] for a survey of the most typical styles) because, even if a service specification can be restyled automatically, the derived protocol specification will reflect the new style, and as such be hardly comprehensible to the designers of the original specification.

On the other hand, we rely without hesitation on restrictions which can be met simply by introducing some additional hidden service actions. Such insertion can always be automated and causes no restructuring of the service specification. Besides, there is usually more than one way to satisfy a restriction by action insertion. By choosing one way or another, it is possible to influence the derived protocol, i.e. its efficiency and the role of individual server components. Hence by relying strongly on such restrictions, we not only simplify the protocol derivation mapping, but also make space for protocol customization.

A service specification satisfying all the prescribed restrictions is a *well-formed specification*. We postpone suggestions for obtaining such a specification to Section 4.2.

3.2 Compositional approach to service implementation

When mapping a service specification in a compositional way, we map each of its constituent process specifications, including the main service process. Mapping a specification of a process p, we map specifications b of the individual parts of the behaviour specified by its body.

During service execution, each instantiation of such a p gives rise to a new instance of the behaviour specified by such a b. Each such instance is an individual service part and, as such, expected to be implemented in an independent way. In other words, such an instance represents a special context, that first of all needs a dynamically unique identifier. The identifier can then be included in all protocol messages belonging to the particular instance, to make its distributed implementation communication-closed. The simplest way to produce such an identifier is to concatenate (specified by operator "$\cdot$") z, the dynamically unique identifier of the particular instance of p, and $CI(b)$, the dynamically unique context identifier of b within the body of p [14].

Mapping a specification of a process p onto a c results in a specification of a local process p with a formal parameter "z". When the local process is instantiated, "z" propagates into its body the identifier of the particular process instance, so that it can be used in the associated protocol messages. The main service process is instantiated only once, so its "z" can be assigned statically. For a dynamically created process instance, "z" is the identifier of its instantiation. Those properties are reflected in Table 6, more precisely described below.

(1)	$\mathbf{T}_c(w,z) := \mathbf{spec}\ \mathbf{Term}_c(b,z)$ **where** $\{\mathbf{T}_c(d)\|(d \in D)\}$ **endspec**
(2)	$\mathbf{T}_c(d) := p$(z) **is** $\mathbf{Term}_c(b_1$, z)
(12)	$\mathbf{T'}_c(b,z) := p(z{\cdot}CI(b))$

Table 6: Mapping $\mathbf{T}$ for a service specification and mapping $\mathbf{T}'$ for process instantiation

$\mathbf{T}_c(b,z)$ will be the basic function for mapping a service part b onto a component c within a context z. Sometimes the implementation of a b generated by mapping $\mathbf{T}$ will be enriched with some additional protocol messages reporting its successful termination to server components not yet knowing it. The mapping which generates such enriched implementation will be called $\mathbf{Term}_c(b,z)$. Mapping $\mathbf{T}$ of a structured b combines the mappings **Term** of its constituent parts.

For a b, it might be that a c has no duties in its distributed implementation, i.e. that c is not a participating component of b (formally $\neg PC_c(b)$, i.e. not a member of $PC(b)$). In such a case, $\mathbf{T}_c(b,z)$ will be ε or $\mathbf{stop}$, while in the case of $PC_c(b)$, $\mathbf{T}_c(b,z)$ will more precisely be called $\mathbf{T'}_c(b,z)$.

In the following, let $\mathbf{Term}(b,z)$ denote a **Term** implementation of b, i.e. all $\mathbf{Term}_c(b,z)$ plus the protocol channels. Likewise, $\mathbf{T}(b,z)$ denotes a $\mathbf{T}$ implementation.

In an environment of competing service parts, it is important to have a simple characterization of all protocol messages belonging to a particular part b. In a $\mathbf{T}(b,z)$, such a message will carry either identifier $CI(b)$ or identifier $CI(b')$ of a subpart b' of b. To indicate that messages of the second type also belong to b, $CI(b')$ will in all cases have $CI(b)$ as a prefix. In a $\mathbf{Term}(b,z)$, the additionally introduced messages will carry identifier $CI^+(b)$. As $\mathbf{T}(b,z)$ is a part of $\mathbf{Term}(b,z)$, $CI(b)$ will have $CI^+(b)$ as a prefix. So it will be possible to specify readiness to receive any message belonging to a $\mathbf{Term}(b,z)$ simply by $?z{\cdot}CI^+(b)$ in the receptions.

The basic building blocks of context identifiers, hence also of protocol messages, are 1 and 2, because they refer typically to parts b_1 and b_2 of a b. That is, of course, not the only possible choice. By changing 2 to 0, for example, one could obtain pure binary identifiers. In any case, it is important that the number of different messages on individual channels is kept low, for message encodings can then be short. For that reason, messages (i.e. the context identifiers they contain) are re-used extensively, except where that could corrupt their dynamic uniqueness.

Example 1 *For the example service in Table 7, it is crucial that the implementations of the two concurrent instances of process Proc use different protocol messages. Likewise it is important that protocol messages are re-used, because Proc is instantiated an infinite number of times.*

The reception buffers of the three components (see Section 3.9) are not shown in the example, to make the specifications more readable. The buffers are not crucial for deadlock prevention, anyhow.

3.3 Termination types

For a b representing the entire service that is being implemented, it is evident that its successful termination (if any) must be implemented as δ (or as its ε equivalent) at each of the server components. In other words, each $\mathbf{Term}_c(b,z)$ must be terminating, i.e. each c must be a terminating component of b for mapping **Term**, formally $TC_c^+(b)$, i.e. c must be an element of $TC^+(b)$.

If a b is not the last part of the service, $TC_c^+(b)$ is not mandatory. It is sometimes better to let $\mathbf{Term}_c(b,z)$ finish by **stop** instead, i.e. $\neg TC_c^+(b)$ [14]. Such inaction at c is later disrupted by activities of c outside $\mathbf{Term}_c(b,z)$. If b never successfully terminates, formally $\neg TM(b)$, $\neg TC_c^+(b)$ is the only option.

If $TC_c^+(b)$, one has to decide whether c should detect or declare termination of b already within the $\mathbf{T}_c(b,z)$ part of $\mathbf{Term}_c(b,z)$, i.e. whether $TC_c^+(b)$ should imply $TC_c(b)$, i.e. that c is an element of $TC(b)$. If $TC_c^+(b)$ but not $TC_c(b)$, formally $RT_c(b)$, c terminates $\mathbf{Term}_c(b,z)$ upon receiving termination reports "$z{\cdot}CI^+(b)$" from all the ending components of $\mathbf{T}(b,z)$ [14] (see Table 8). Where the danger exists of such a report being received already within $\mathbf{T}_c(b,z)$, care is taken that it is different from any message referred to within $\mathbf{T}_c(b,z)$. Hence protocol $\mathbf{Term}(b,z)$

$w = \mathbf{spec\ ren}\ \mathrm{a}^\alpha/\mathrm{A}^\alpha, \mathrm{b}^\gamma/\mathrm{B}^\gamma, \mathrm{c}^\beta/\mathrm{C}^\beta\ \mathbf{in}\ \mathrm{Proc}\ \mathbf{endren}\ |||\ \mathbf{ren}\ \mathrm{d}^\alpha/\mathrm{A}^\alpha, \mathrm{e}^\gamma/\mathrm{B}^\gamma, \mathrm{f}^\beta/\mathrm{C}^\beta\ \mathbf{in}\ \mathrm{Proc}\ \mathbf{endren}$
$\quad \mathbf{where}\ \mathrm{Proc}\ \mathbf{is}\ (((\mathrm{A}^\alpha; \delta^\alpha)|||(\mathrm{B}^\gamma; \delta^\gamma)) \gg (\mathrm{C}^\beta; \mathrm{Proc}))\ \mathbf{endspec}$
$\mathbf{T}_\alpha(w, \varepsilon) \approx \mathbf{spec\ ren}\ \mathrm{a}^\alpha/\mathrm{A}^\alpha\ \mathbf{in}\ \mathrm{Proc}(1)\ \mathbf{endren}\ |||\ \mathbf{ren}\ \mathrm{d}^\alpha/\mathrm{A}^\alpha\ \mathbf{in}\ \mathrm{Proc}(2)\ \mathbf{endren}$
$\quad \mathbf{where}\ \mathrm{Proc(z)}\ \mathbf{is}\ (\mathrm{A}^\alpha; \mathbf{s}^\alpha_\beta!\mathrm{z}; \mathbf{r}^\alpha_\beta!\mathrm{z}; \mathrm{Proc(z)})\ \mathbf{endspec}$
$\mathbf{T}_\beta(w, \varepsilon) \approx \mathbf{spec\ ren}\ \mathrm{c}^\beta/\mathrm{C}^\beta\ \mathbf{in}\ \mathrm{Proc}(1)\ \mathbf{endren}\ |||\ \mathbf{ren}\ \mathrm{f}^\beta/\mathrm{C}^\beta\ \mathbf{in}\ \mathrm{Proc}(2)\ \mathbf{endren}$
$\quad \mathbf{where}\ \mathrm{Proc(z)}\ \mathbf{is}\ (((\mathbf{r}^\beta_\alpha!\mathrm{z}; \delta)|||(\mathbf{r}^\beta_\gamma!\mathrm{z}; \delta)) \gg \mathrm{C}^\beta; ((\mathbf{s}^\beta_\alpha!\mathrm{z}; \delta)|||(\mathbf{s}^\beta_\gamma!\mathrm{z}; \delta)) \gg \mathrm{Proc(z)})\ \mathbf{endspec}$
$\mathbf{T}_\gamma(w, \varepsilon) \approx \mathbf{spec\ ren}\ \mathrm{b}^\gamma/\mathrm{B}^\gamma\ \mathbf{in}\ \mathrm{Proc}(1)\ \mathbf{endren}\ |||\ \mathbf{ren}\ \mathrm{e}^\gamma/\mathrm{B}^\gamma\ \mathbf{in}\ \mathrm{Proc}(2)\ \mathbf{endren}$
$\quad \mathbf{where}\ \mathrm{Proc(z)}\ \mathbf{is}\ (\mathrm{B}^\gamma; \mathbf{s}^\gamma_\beta!\mathrm{z}; \mathbf{r}^\gamma_\beta!\mathrm{z}; \mathrm{Proc(z)})\ \mathbf{endspec}$

Table 7: An example of multiple process instantiation

$\mathbf{T}_c(b, z) :=$ if $PC_c(b)$ then $\mathbf{T}'_c(b, z)$ else if $TC_c(b)$ then ε else **stop** endif endif
$\mathbf{Term}_c(b, z) :=$ if $TC^+_c(b)$ then if $TC_c(b)$ then $(\mathbf{T}_c(b, z)$ if $EC_c(b)$ then $\gg \mathbf{S}_c((TC^+(b) \setminus TC(b)), z{\cdot}CI^+(b))$ endif)
else $((\mathbf{T}_c(b, z)[>\delta)|||\mathbf{R}_c(EC(b), z{\cdot}CI^+(b)))$ endif
else $\mathbf{T}_c(b, z)$ endif

Table 8: Functions **T** and **Term**

has two phases, namely protocol $\mathbf{T}(b, z)$ and exchange of termination reports.

A c is an ending component of b for mapping **T**, formally $EC_c(b)$, i.e. c is a member of $EC(b)$, if it might be the last component to execute an action within $\mathbf{T}(b, z)$. If $EC_c(b)$, c must, of course, declare termination already within $\mathbf{T}_c(b, z)$, i.e. $EC_c(b)$ by definition implies $TC_c(b)$, and thereby $TC^+_c(b)$.

In many cases, we are free to decide whether $TC^+_c(b)$ should imply $TC_c(b)$ or not, but it is not always directly evident how our decision would influence the overall number of the involved protocol messages. Therefore we follow the classical solution that $TC^+_c(b)$ should always imply $TC_c(b)$ (i.e. $\neg RT_c(b)$), except where that would lead to an erroneous service implementation (discussed in the operator-specific sections). If there are no such cases, mapping **Term** systematically reduces to mapping **T**, i.e. there is a single mapping function, like in the earlier approaches [3, 10].

If $\neg PC_c(b)$, $TC_c(b)$ will always be equal to $TC^+_c(b)$, reducing $\mathbf{Term}_c(b, z)$ to a mere ε or **stop** (see function **T** in Table 8). Hence the components participating in the distributed implementation of a b remain those listed in $PC(b)$, even if we enhance the mapping function from **T** to **Term**.

For a protocol $\mathbf{T}(b, z)$, we define that it successfully terminates when all $\mathbf{T}_c(b, z)$ with $TC_c(b)$ successfully terminate. Likewise, successful termination of $\mathbf{Term}(b, z)$ requires successful termination of all $\mathbf{Term}_c(b, z)$ with $TC^+_c(b)$.

3.4 Implementation of inaction

A **stop** has no participating component, so the first rule in Table 8 implies that every server component implements it as a **stop**.

3.5 Implementation of successful termination

In some cases, it is crucial to have in mind that successful termination δ is also a kind of an action. These are the cases where it is in a decisive position, like an initial δ in a "$b_1[]b_2$" or the δ of b_1 or an initial δ of b_2 in a "$b_1[>b_2$" [14]. So one selects, as convenient, for each δ a server component responsible for its execution, its only participating component. Mapping $\mathbf{T}'$ for the component is a δ (Table 9).

(4) $\mathbf{T}'_c(b, z) := \delta$

Table 9: Mapping $\mathbf{T}'$ for successful termination

3.6 Implementation of hiding and renaming

The only properties of actions within a service part b that influence protocol message exchange are their position within b and their assignment to server components. That is not changed by hiding or local renaming, so implementation of those operations is trivial (Table 10).

(10) $\mathbf{T}'_c(b, z) := \mathbf{hide}\ \mathbf{P}_c(S)\ \mathbf{in}\ \mathbf{Term}_c(b_1, z)\ \mathbf{endhide}$
(11) $\mathbf{T}'_c(b, z) := \mathbf{ren}\ \mathbf{P}_c(R)\ \mathbf{in}\ \mathbf{Term}_c(b_1, z)\ \mathbf{endren}$

Table 10: Mapping $\mathbf{T}'$ for hiding and renaming

3.7 Implementation of action prefix

To map an "$a; b_2$" onto a participant c (Table 11), one first needs $\mathbf{P}_c(a)$, the projection of a. If c is not the executor of a, i.e. its only participant, the projection is empty. If a is a service primitive, its executor is evident from its identifier. If it is an **i**, one selects its executor as convenient.

If a component c might be the first to execute an action within $\mathbf{Term}(b_2, z)$, it is a starting component of b_2, formally $SC_c(b_2)$, i.e. c is a member of $SC(b_2)$. Such a c is responsible for preventing a premature start of

(13) $\mathbf{P}_c(a) :=$ if $PC_c(a)$ then a else ε endif
(6) $\mathbf{T'}_c(b,z) := (\mathbf{P}_c(a); \mathbf{E}_c(PC(a), SC(b_2), z{\cdot}CI(b)) \gg \mathbf{Term}_c(b_2,z))$

Table 11: Mapping $\mathbf{T}'$ for action prefix

$\mathbf{Term}(b_2,z)$, i.e. it must not start $\mathbf{Term}_c(b_2,z)$ until it executes a or receives a report "$z \cdot CI(b)$" on it. Hence protocol $\mathbf{T}(b,z)$ has three phases, namely execution of a, exchange of reports on a, and protocol $\mathbf{Term}(b_2,z)$.

3.8 Implementation of sequential composition

For a b specified as "$b_1 \gg b_2$", we require that b_1, at least sometimes, successfully terminates, because otherwise b_2 would be irrelevant.

Protocol $\mathbf{T}(b,z)$ (Table 12) has three phases, namely protocol $\mathbf{Term}(b_1,z)$, exchange of reports "$z \cdot CI(b)$" on its termination, and protocol $\mathbf{Term}(b_2,z)$. Where danger exists that a message belonging to the second phase is received already within a $\mathbf{Term}_c(b_1,z)$, care is taken that it is different from any message referred to within $\mathbf{Term}_c(b_1,z)$. It is crucial that every c with duties within the second or the third phase terminates $\mathbf{Term}_c(b_1,z)$ in all the terminating runs of b_1, i.e. that $TC_c^+(b_1)$ is true.

(5) $\mathbf{T'}_c(b,z) := (\mathbf{Term}_c(b_1,z) \gg \mathbf{E}_c(EC^+(b_1), SC(b_2), z{\cdot}CI(b)) \gg \mathbf{Term}_c(b_2,z))$

Table 12: Mapping $\mathbf{T}'$ for sequential composition

As in the case of action prefix, reports on termination of the first phase are sent to the starting components of b_2, but now their senders are the ending components of $\mathbf{Term}(b_1,z)$ [19]. A c is an ending component of b_1 for mapping **Term**, formally $EC_c^+(b_1)$, i.e. c is a member of $EC^+(b_1)$, if it might be the last component to execute an action within $\mathbf{Term}(b_1,z)$. It is crucial that a terminating b_1 has at least one ending component, and that in every non-terminating run of such a b_1, there is at least one ending component c not terminating $\mathbf{Term}_c(b_1,z)$, so that start of $\mathbf{Term}(b_2,z)$ is prevented.

We want the second phase (i.e. termination reporting) to completely isolate $\mathbf{Term}(b_2,z)$ from $\mathbf{Term}(b_1,z)$, so that protocol messages from $\mathbf{Term}(b_1,z)$ and termination reports may be re-used within $\mathbf{Term}(b_2,z)$. That is particularly important for implementation of iteration and tail recursion, as in Example 2. To achieve the isolation, we take care that upon the start of $\mathbf{Term}(b_2,z)$, components receiving within it no longer want to receive within $\mathbf{Term}(b_1,z)$.

Example 2 *In Table 13, we implement a service consisting of two consecutive parts. It might happen that the first part does not terminate, but a premature start of the second part is nevertheless prevented.*

3.9 Implementation of parallel composition

For a b specified as "$b_1|[S]|b_2$", we assume that all actions specified in b_1 or b_2, including δ, are actually executable within b, i.e. that they are all relevant.

Protocol $\mathbf{T}(b,z)$ (Table 14) consists basically of protocols $\mathbf{Term}(b_1,z)$ and $\mathbf{Term}(b_2,z)$ running in parallel and locally synchronized on service primitives from S.

If there are any distributed conflicts in b_1 and/or b_2, formally $AD(b)$, $\mathbf{Term}(b_1,z)$ and/or $\mathbf{Term}(b_2,z)$ are typically imprecise implementations of b_1 and b_2, unable to synchronize properly on S. So if S is non-empty, $AD(b)$ is forbidden.

If S is empty, b_1 and b_2 are nevertheless synchronized on their successful termination (if any). If termination of b is subject to a distributed conflict within b_1 and/or b_2, formally $TD(b)$, negotiation of more than one component is required within $\mathbf{Term}(b_1,z)$ and/or $\mathbf{Term}(b_2,z)$. That is unacceptable, for such termination is a decisive termination (see below). So $TD(b)$ is forbidden.

For independent concurrent execution of $\mathbf{Term}(b_1,z)$ and $\mathbf{Term}(b_2,z)$, it should be sufficient to take care that their protocol message spaces are disjoint [10]. Unfortunately, it turns out that on a shared channel, unprompt reception in one of the protocols might hinder reception in the other. In the case of a non-empty S, that might even lead to a deadlock [15].

Kant and Higashino suggested that each c could solve the problem by prompt reception of messages into a pool, for further consumption by $\mathbf{Term}_c(b_1,z)$ or $\mathbf{Term}_c(b_2,z)$. So in Table 14, we introduce for each part $\mathbf{Term}_c(b_n,z)$ for each channel from a c' to c that is shared (formally $SH_{c',c}(b)$), a FIFO buffer for incoming messages. Such a buffer is, unlike $\mathbf{Term}_c(b_n,z)$, always ready to receive from the channel on gate $\mathbf{r}^c_{c'}$, thereby removing the possibility of blocking. $\mathbf{Term}_c(b_n,z)$ can subsequently claim the received messages from the buffer on a hidden gate $\mathbf{b}_{c'}$. As demonstrated in the following example, such buffers might be necessary even if S is empty. On the other hand, buffers are often redundant, but that is hard to establish.

Example 3 *In the first part of Table 15, there is a parallel composition implemented properly.*

In the second part, the reception buffers are omitted, and there is a scenario "$\mathrm{a}^\alpha; \mathbf{s}^\alpha_\beta!1; \mathrm{d}^\alpha; \mathbf{s}^\alpha_\beta!2$*" leading to a dead-*

$w = \mathbf{spec}\ ((\mathrm{a}^\alpha; \mathrm{Proc})[](\mathrm{b}^\alpha; \delta^\beta)) \gg (\mathrm{b}^\gamma; \delta^\gamma)$ **where** Proc **is** $(\mathrm{c}^\beta; \mathrm{c}^\alpha; \mathrm{Proc})$ **endspec**
$\mathbf{T}_\alpha(w,1) \approx \mathbf{spec}\ (\mathrm{a}^\alpha; \mathbf{s}^\alpha_\beta!11; \mathrm{Proc})[](\mathrm{b}^\alpha; \mathbf{s}^\alpha_\beta!12; \delta)$ **where** Proc **is** $(\mathbf{r}^\alpha_\beta!11; \mathrm{c}^\alpha; \mathbf{s}^\alpha_\beta!11; \mathrm{Proc})$ **endspec**
$\mathbf{T}_\beta(w,1) \approx \mathbf{spec}\ ((\mathbf{r}^\beta_\alpha!11; \mathrm{Proc})[](\mathbf{r}^\beta_\alpha!12; \delta)) \gg \mathbf{s}^\beta_\gamma!1; \delta$ **where** Proc **is** $(\mathrm{c}^\beta; \mathbf{s}^\beta_\alpha!11; \mathbf{r}^\beta_\alpha!11; \mathrm{Proc})$ **endspec**
$\mathbf{T}_\gamma(w,1) \approx \mathbf{spec}\ \mathbf{r}^\gamma_\beta!1; \mathrm{b}^\gamma; \delta$ **endspec**

Table 13: An example combining finite and infinite alternatives

(7) $\mathbf{T'}_c(b,z) := (\mathbf{Par}_{c,1}\|[\mathbf{P}_c(S)]\|\mathbf{Par}_{c,2})$ where $\mathbf{Par}_{c,n} :=$ **hide** $\{\mathbf{b}_{c'}\|SH_{c',c}(b)\}$ **in ren** $\{(\mathbf{b}_{c'}/\mathbf{r}^c_{c'})\|SH_{c',c}(b)\}$ **in** $\mathbf{Term}_c(b_n,z)$ **endren** $\|[\{\mathbf{b}_{c'}\|SH_{c',c}(b)\}]\|\ \|\|_{SH_{c',c}(b)}\mathrm{FIFO}(\mathbf{r}^c_{c'},\mathbf{b}_{c'},z{\cdot}CI^+(b_n))$ **endhide**

Table 14: Mapping $\mathbf{T'}$ for parallel composition

$w =$ **spec** $(((\mathrm{a}^\alpha;\delta^\alpha)\|\|\|(\mathrm{b}^\beta;\delta^\beta))\gg(\mathrm{c}^\beta;\delta^\beta))\|[\mathrm{b}^\beta]\|(\mathrm{d}^\alpha;\mathrm{b}^\beta;\delta^\beta)$ **endspec**
$\mathbf{T}_\alpha(w,\varepsilon)\approx$ **spec** $(\mathrm{a}^\alpha;\mathbf{s}^\alpha_\beta!1;\delta)\|\|\|(\mathrm{d}^\alpha;\mathbf{s}^\alpha_\beta!2;\delta)$ **endspec**
$\mathbf{T}_\beta(w,\varepsilon)\approx$ **spec hide** $\mathbf{b}_\alpha$ **in** $(\mathrm{b}^\beta;\mathbf{b}_\alpha!1;\mathrm{c}^\beta;\delta)\|[\mathbf{b}_\alpha]\|\mathrm{FIFO}(\mathbf{r}^\beta_\alpha,\mathbf{b}_\alpha,1)$ **endhide** $\|[\mathrm{b}^\beta]\|$**hide** $\mathbf{b}_\alpha$ **in** $(\mathbf{b}_\alpha!2;\mathrm{b}^\beta;\delta)\|[\mathbf{b}_\alpha]\|\mathrm{FIFO}(\mathbf{r}^\beta_\alpha,\mathbf{b}_\alpha,2)$ **endhide endspec**
$\mathbf{T}_\alpha(w,\varepsilon)\approx$ **spec** $(\mathrm{a}^\alpha;\mathbf{s}^\alpha_\beta!1;\delta)\|\|\|(\mathrm{d}^\alpha;\mathbf{s}^\alpha_\beta!2;\delta)$ **endspec**
$\mathbf{T}_\beta(w,\varepsilon)\approx$ **spec** $(\mathrm{b}^\beta;\mathbf{r}^\beta_\alpha!1;\mathrm{c}^\beta;\delta)\|[\mathrm{b}^\beta]\|(\mathbf{r}^\beta_\alpha!2;\mathrm{b}^\beta;\delta)$ **endspec**
$w =$ **spec** $(((\mathrm{a}^\alpha;\delta^\alpha)\|\|\|(\mathrm{b}^\beta;\delta^\beta))\gg(\mathrm{c}^\beta;\delta^\beta))\|\|\|(\mathrm{d}^\alpha;\mathrm{e}^\beta;\delta^\beta)$ **endspec**
$\mathbf{T}_\alpha(w,\varepsilon)\approx$ **spec** $(\mathrm{a}^\alpha;\mathbf{s}^\alpha_\beta!1;\delta)\|\|\|(\mathrm{d}^\alpha;\mathbf{s}^\alpha_\beta!2;\delta)$ **endspec**
$\mathbf{T}_\beta(w,\varepsilon)\approx$ **spec** $(\mathrm{b}^\beta;\mathbf{r}^\beta_\alpha!1;\mathrm{c}^\beta;\delta)\|\|\|(\mathbf{r}^\beta_\alpha!2;\mathrm{e}^\beta;\delta)$ **endspec**

Table 15: An example of parallel composition requiring buffered reception

$w =$ **spec** $(\delta^\alpha[>(\mathrm{a}^\alpha;\mathrm{b}^\beta;\delta^\alpha))\|[\mathrm{a}^\alpha]\|(\delta^\alpha[](\mathbf{i}^\alpha;\mathrm{a}^\alpha;\delta^\alpha))$ **endspec**
$\mathbf{T}_\alpha(w,1)\approx$ **spec** $((\delta[>(\mathrm{a}^\alpha;\mathbf{s}^\alpha_\beta!11;\mathbf{r}^\alpha_\beta!11;\delta))$ $\|[\mathrm{a}^\alpha]\|(\delta[](\mathbf{i};\mathrm{a}^\alpha;\delta)))$ $\gg\mathbf{s}^\alpha_\beta!1;\delta$ **endspec**
$\mathbf{T}_\beta(w,1)\approx$ **spec** $((\mathbf{r}^\beta_\alpha!11;\mathrm{b}^\beta;\mathbf{s}^\beta_\alpha!11;$ **stop**$)[>\delta)$ $\|\|(\mathbf{r}^\beta_\alpha!1;\delta)$ **endspec**

Table 16: An example of decisive and synchronized termination

lock, because message 2 is not the first in the channel.

In the third part, we no longer require that the two concurrent parts are synchronized on b^β*. We also rename the second* b^β *into* e^β*, to distinguish it from the first one. The above scenario no longer leads to a deadlock, but its destination state erroneously requires that* b^β *is executed before* e^β*. Again, reception buffers would help.*

For a b specified as "$b_1|[S]|b_2$", successful termination of $\mathbf{T}(b,z)$ requires successful termination of $\mathbf{Term}(b_1,z)$ and $\mathbf{Term}(b_2,z)$. If such termination is decisive for one or both of the component protocols, i.e. represents a δ in a decisive position within b_1 or b_2, formally $DT(b)$, its implementation is problematic [14, 15]. It has been suggested that such a δ should be put under control of a single server component, its pre-assigned executor, responsible both for its decisive role and for its synchronization role [14]. If successful termination of $\mathbf{T}(b,z)$ is to be a matter of a single component, the latter must be the only member of $TC(b)$, and consequently the only member of $EC(b)$, $TC^+(b_1)$, $TC^+(b_2)$, $EC(b_1)$ and $EC(b_2)$.

Example 4 *An example of decisive and synchronized termination is given in Table 16. Termination of* b *has been put under exclusive control of component* α*, while component* β *receives only a report of it.*

3.10 Implementation of choice

For a b specified as "$b_1[]b_2$", we assume that there are service actions (at least a δ) in both alternatives, so that both are relevant. The operator introduces distributed conflicts, formally $DC(b)$, if b has more than one starting component.

Protocol $\mathbf{T}(b,z)$ combines protocols $\mathbf{Term}(b_1,z)$ and $\mathbf{Term}(b_2,z)$. b_2 is the higher-priority alternative, so $\mathbf{Term}(b_2,z)$ upon its start always quickly disables $\mathbf{Term}(b_1,z)$, even if $\mathbf{Term}(b_1,z)$ has already started. On the other hand, when a component detects the start of $\mathbf{Term}(b_1,z)$, it tries to prevent starting of $\mathbf{Term}(b_2,z)$, but might be unsuccessful.

Until one of the alternatives is abandoned, protocols $\mathbf{Term}(b_1,z)$ and $\mathbf{Term}(b_2,z)$ run in parallel, so we require that their protocol message sets are disjoint.

Within $\mathbf{Term}(b_1,z)$, any starting action must be promptly reported to any starting component c of b_2, formally $SR_c(b_1)$, to inform it that execution of b_2 should not start unless it already has. Analogously, we require $SR_c(b_2)$ for any starting component c of b_1. If $DC(b)$, any component might already be executing b_1 when $\mathbf{Term}(b_2,z)$ starts, so we require $SR_c(b_2)$ also for the non-starting participants of b_1, to make them quickly abandon execution of b_1. Note that the executor of an action is informed of the action by the action itself.

If not earlier, a participant c abandons $\mathbf{Term}_c(b_2,z)$ upon successful termination of $\mathbf{Term}_c(b_1,z)$, if any. At that moment, it must already be obvious that $\mathbf{Term}(b_2,z)$ will never start, i.e. every starting component of b_2 must have already executed an action within $\mathbf{Term}(b_1,z)$, thereby refusing to be an initiator of $\mathbf{Term}(b_2,z)$. In other words, such a starting component c' must guard the termination at c, formally $GT^+_{c,c'}(b_1)$.

If not earlier, a participant c abandons $\mathbf{Term}_c(b_1,z)$ upon successful termination of $\mathbf{Term}_c(b_2,z)$, if any. At that moment, c must already have detected the start of $\mathbf{Term}(b_2,z)$, and that is true if and only if c is a participating component of b_2.

(8) $\mathbf{T'}_c(b,z) :=$ if $\neg DC(b)$ then $(\mathbf{Term}_c(b_1,z)[]\mathbf{Term}_c(b_2,z))$
else **ren** $\cup_{n=1,2}$ $(\{(u^c/u^c_n)|(u^c \in AS_c(b_n))\} + \{(\mathbf{r}^c_{c'}/\mathbf{a}^n_{c'})|CH^+_{c',c}(b_n)\})$ **in hide t in**
$((\mathbf{Const}_{c,1}|[\mathbf{StGt}_{c,2} + \mathbf{RecGt}_{c,2} + \{\mathbf{t}\}]|\mathbf{Const}_{c,2})$
$|[\mathbf{StGt}_{c,1} + \mathbf{RecGt}_{c,1} + \mathbf{StGt}_{c,2} + \mathbf{RecGt}_{c,2}]|\mathbf{Const}_{c,3})$
$|[\mathbf{RecGt}_{c,1} + \{\mathbf{a}^2_{c'}|CH^+_{c',c}(b_1)\}]|\mathbf{Const}_{c,4}$
endhide endren
where $\mathbf{Const}_{c,1} := (((\mathbf{Task}_{c,1} \gg \mathbf{t}; \mathbf{stop})[> (\mathbf{OneStRec}_{c,2} \gg (\mathbf{AllStRec}_{c,2}|||\mathbf{AllRec}_{c,1})))[> \delta)$
where $\mathbf{Task}_{c,1} :=$ **ren** $\{(u^c_1/u^c)|(u^c \in AS_c(b_1))\} + \{(\mathbf{a}^1_{c'}/\mathbf{r}^c_{c'})|CH^+_{c',c}(b_1)\}$ **in** $\mathbf{Par}_{c,1}$ **endren**
where $\mathbf{Par}_{c,1} :=$ see Table 14
$\mathbf{Const}_{c,2} := (\mathbf{Task}_{c,2}[](\mathbf{t}; \delta))$
where $\mathbf{Task}_{c,2} :=$ **ren** $\{(u^c_2/u^c)|(u^c \in AS_c(b_2))\} + \{(\mathbf{a}^2_{c'}/\mathbf{r}^c_{c'})|CH^+_{c',c}(b_2)\}$
in $\mathbf{Term}_c(b_2,z)$ **endren**
$\mathbf{Const}_{c,3} := (((\mathbf{OneStRec}_{c,2} \gg (\mathbf{AllStRec}_{c,2}|||\mathbf{AllRec}_{c,1}))[]$
$(\mathbf{OneStRec}_{c,1} \gg (\mathbf{AllStRec}_{c,1}$
$[> (\mathbf{OneRec}_{c,2} \gg (\mathbf{AllStRec}_{c,2}|||\mathbf{AllRec}_{c,1})))))$
$[> \delta)$
$\mathbf{Const}_{c,4} := ((|||_{CH^+_{c',c}(b_1)}(\mathrm{Loop}(\mathbf{a}^1_{c'}?z{\cdot}CI^+(b_1))[> \mathrm{Loop}(\mathbf{a}^2_{c'}?z{\cdot}CI^+(b_2))))[> \delta)$
$\mathbf{StGt}_{c,n} := \{u^c_n|(u^c \in SS_c(b_n))\}$
$\mathbf{RecGt}_{c,n} := \{\mathbf{a}^n_{c'}|CH^+_{c',c}(b_n)\}$
$\mathbf{OneRec}_{c,n} := ([]_{g \in \mathbf{RecGt}_{c,n}}(g?z{\cdot}CI^+(b_n); \delta))$
$\mathbf{OneStRec}_{c,n} := (([]_{g \in \mathbf{StGt}_{c,n}}(g; \delta))[]\mathbf{OneRec}_{c,n})$
$\mathbf{AllRec}_{c,n} := (\mathbf{stop}|||(|||_{g \in \mathbf{RecGt}_{c,n}}\mathrm{Loop}(g?z{\cdot}CI^+(b_n))))$
$\mathbf{AllStRec}_{c,n} := ((|||_{g \in \mathbf{StGt}_{c,n}}\mathrm{Loop}(g))|||\mathbf{AllRec}_{c,n})$ endif

Table 17: Mapping $\mathbf{T'}$ for choice

A participant c combines $\mathbf{Term}_c(b_1,z)$ and $\mathbf{Term}_c(b_2,z)$ as specified in Table 17. If $\neg DC(b)$, $\mathbf{Term}(b_1,z)$ is known to be the selected alternative as soon as it starts, so every c is allowed to execute $\mathbf{Term}_c(b_1,z)$ and $\mathbf{Term}_c(b_2,z)$ as alternatives.

If $DC(b)$, $\mathbf{Term}_c(b_1,z)$ and $\mathbf{Term}_c(b_2,z)$ must be combined in such a complicated way that no LOTOS operator can express it directly. So we resort to the so called *constraint-oriented specification style* [28]. This is the style in which two or more parallel processes synchronize on the actions they collectively control, and each process imposes its own constraints on the execution of the actions, so that they are enabled only when so allowed by all the processes referring to them.

A $\mathbf{T'}_c(b,z)$ consists of four constraints. $\mathbf{Const}_{c,1}$ and $\mathbf{Const}_{c,2}$ are respectively responsible for execution of $\mathbf{Term}_c(b_1,z)$ and $\mathbf{Term}_c(b_2,z)$, while $\mathbf{Const}_{c,3}$ and $\mathbf{Const}_{c,4}$ serve for their additional co-ordination.

In the first place, we must be aware that in the case of $DC(b)$, protocols $\mathbf{Term}(b_1,z)$ and $\mathbf{Term}(b_2,z)$ are actually executed in parallel for some time, so every shared incoming channel in principle requires an input buffer for $\mathbf{Term}_c(b_1,z)$ and an input buffer for $\mathbf{Term}_c(b_2,z)$ (see Section 3.9). But as no c' ever transmits to c within $\mathbf{Term}_{c'}(b_1,z)$ after it has transmitted to c within $\mathbf{Term}_{c'}(b_2,z)$, input buffers for prompt reception are necessary only for $\mathbf{Term}_c(b_1,z)$. So we enhance $\mathbf{Term}_c(b_1,z)$ into $\mathbf{Par}_{c,1}$, as described in Table 14, though the buffers are usually redundant.

Internally to $\mathbf{T'}_c(b,z)$, we rename every service primitive u^c in $\mathbf{Term}_c(b_1,z)$ (i.e. in $\mathbf{Par}_{c,1}$) into u^c_1. Likewise, we internally rename every service primitive u^c in $\mathbf{Term}_c(b_2,z)$ into u^c_2. Besides, we internally to $\mathbf{T'}_c(b,z)$ split every reception gate $\mathbf{r}^c_{c'}$ into gates $\mathbf{a}^1_{c'}$ and $\mathbf{a}^2_{c'}$, where messages for $\mathbf{Term}_c(b_1,z)$ are, according to their contents, routed to the first gate, and messages for $\mathbf{Term}_c(b_2,z)$ to the second gate. The renamings are guided by service attributes $AS_c(b_n)$ (lists all the service actions of b_n at c) and $CH^+_{c',c}(b_n)$ (true if the channel from c' to c is employed within $\mathbf{Term}(b_n,z)$).

Applying all the above renamings to $\mathbf{Par}_{c,1}$ and $\mathbf{Term}_c(b_2,z)$, we obtain processes $\mathbf{Task}_{c,1}$ and $\mathbf{Task}_{c,2}$, respectively, that have disjoint sets of service primitives and reception gates. Every action within $\mathbf{T'}_c(b,z)$ is an action of $\mathbf{Task}_{c,1}$ or an action of $\mathbf{Task}_{c,2}$, except that there is also an action on a hidden gate **t** serving for synchronization of $\mathbf{Const}_{c,1}$ and $\mathbf{Const}_{c,2}$ upon successful termination of $\mathbf{Task}_{c,1}$.

The critical actions of $\mathbf{Task}_{c,1}$ are its starting actions. They must influence execution of $\mathbf{Task}_{c,2}$, so they are subject to synchronization between $\mathbf{Const}_{c,1}$ and $\mathbf{Const}_{c,3}$. A starting action of $\mathbf{Task}_{c,1}$ is a starting service action of b_1 at c, i.e. a member of $SS_c(b_1)$, or a reception. If it is a member of $SS_c(b_1)$, it might also be an **i** or a δ, i.e. not suitable for synchronization, so we in principle require that every member of $SS_c(b_1)$ is a service primitive. If c is not a starting component of b_2, $\mathbf{Const}_{c,3}$ is redundant, hence the requirement is not necessary.

The critical actions of $\mathbf{Task}_{c,2}$ are its starting actions. They must in principle influence execution of $\mathbf{Task}_{c,1}$, so they are subject to synchronization between $\mathbf{Const}_{c,1}$ and $\mathbf{Const}_{c,2}$. A starting action of $\mathbf{Task}_{c,2}$ is a member of $SS_c(b_2)$ or a reception. If disruption of $\mathbf{Task}_{c,1}$ is necessary, i.e. if $PC_c(b_1)$, we require that every member of

$w = \mathbf{spec}\ ((\mathrm{a}^{\alpha};\delta)|||(\mathrm{b}^{\beta};\delta))[]((\mathrm{c}^{\gamma};\delta)|||(\mathrm{b}^{\beta};\delta))\ \mathbf{endspec}$
$w_1 = \mathbf{spec}\ ((\mathrm{a}^{\alpha};(\delta^{\beta}|||\delta^{\gamma}))||||(\mathrm{b}^{\beta};\delta^{\gamma}))[]((\mathrm{c}^{\gamma};(\delta^{\alpha}|||\delta^{\beta}))||||(\mathrm{b}^{\beta};(\delta^{\alpha}|||\delta^{\gamma})))\ \mathbf{endspec}$
$w_2 = \mathbf{spec}\ ((\mathrm{a}^{\alpha};(\delta^{\beta}|||\delta^{\gamma}))||||(\mathrm{b}^{\beta};\mathbf{i}^{\gamma};(\delta^{\alpha}|||\delta^{\beta})))[]((\mathrm{c}^{\gamma};(\delta^{\alpha}|||\delta^{\beta}))||||(\mathrm{b}^{\beta};(\delta^{\alpha}|||\delta^{\gamma})))\ \mathbf{endspec}$
$w_3 = \mathbf{spec}\ ((\mathrm{a}^{\alpha};(\delta^{\beta}|||\delta^{\gamma}))||||(\mathrm{b}^{\beta};\mathbf{i}^{\gamma};(\delta^{\alpha}|||\delta^{\beta})))[]((\mathrm{c}^{\gamma};((\mathbf{i}^{\alpha};\delta^{\gamma})|||\delta^{\beta}))||||(\mathrm{b}^{\beta};((\mathbf{i}^{\alpha};\delta^{\beta})|||\delta^{\gamma})))\ \mathbf{endspec}$

$\mathbf{T}_{\alpha}(w_3,\varepsilon) \approx \mathbf{spec\ ren}\ \mathbf{r}^{\alpha}_{\gamma}/\mathbf{a}^{1}_{\gamma}, \mathbf{r}^{\alpha}_{\beta}/\mathbf{a}^{2}_{\beta}, \mathbf{r}^{\alpha}_{\gamma}/\mathbf{a}^{2}_{\gamma}\ \mathbf{in\ hide\ t\ in}$
$(((\boxed{(\mathbf{hide}\ \mathbf{b}_{\gamma}\ \mathbf{in}\ ((\mathrm{a}^{\alpha};((\mathbf{s}^{\alpha}_{\beta}!1;\delta)|||(\mathbf{s}^{\alpha}_{\gamma}!1;\delta)))||||(\mathbf{b}_{\gamma}!1;\delta))|[\mathbf{b}_{\gamma}]|\mathrm{FIFO}(\mathbf{a}^{1}_{\gamma},\mathbf{b}_{\gamma},1)\ \mathbf{endhide}\gg\mathbf{t};\mathbf{stop})}$
$[>(((\mathbf{a}^{2}_{\beta}?2;\delta)[](\mathbf{a}^{2}_{\gamma}?2;\delta))\gg(\mathrm{Loop}(\mathbf{a}^{2}_{\beta}?2)|||\mathrm{Loop}(\mathbf{a}^{2}_{\gamma}?2)|||\mathrm{Loop}(\mathbf{a}^{1}_{\gamma}?1))))[>\delta)$
$|[\mathbf{a}^{2}_{\beta},\mathbf{a}^{2}_{\gamma},\mathbf{t}]|(\boxed{((\mathbf{a}^{2}_{\gamma}!2;\mathbf{s}^{\alpha}_{\gamma}!2;\delta)||||(\mathbf{a}^{2}_{\beta}!2;\mathbf{s}^{\alpha}_{\beta}!2;\delta))}\ [](\mathbf{t};\delta))$
$|[\mathbf{a}^{1}_{\gamma},\mathbf{a}^{2}_{\gamma}]|((\mathrm{Loop}(\mathbf{a}^{1}_{\gamma}?1)[>\mathrm{Loop}(\mathbf{a}^{2}_{\gamma}?2))[>\delta)$
$\mathbf{endhide\ endren\ endspec}$

$\mathbf{T}_{\beta}(w_3,\varepsilon) \approx \mathbf{spec\ ren}\ \mathrm{b}^{\beta}/\mathrm{b}^{\beta}_{1}, \mathbf{r}^{\beta}_{\alpha}/\mathbf{a}^{1}_{\alpha}, \mathbf{r}^{\beta}_{\gamma}/\mathbf{a}^{1}_{\gamma}, \mathrm{b}^{\beta}/\mathrm{b}^{\beta}_{2}, \mathbf{r}^{\beta}_{\alpha}/\mathbf{a}^{2}_{\alpha}, \mathbf{r}^{\beta}_{\gamma}/\mathbf{a}^{2}_{\gamma}\ \mathbf{in\ hide\ t\ in}$
$((((\boxed{(\mathbf{hide}\ \mathbf{b}_{\alpha},\mathbf{b}_{\gamma}\ \mathbf{in}\ ((\mathbf{b}_{\alpha}!1;\delta)||||(\mathrm{b}^{\beta}_{1};\mathbf{s}^{\beta}_{\gamma}!1;\mathbf{b}_{\gamma}!1;\delta))$
$|[\mathbf{b}_{\alpha},\mathbf{b}_{\gamma}]|(\mathrm{FIFO}(\mathbf{a}^{1}_{\alpha},\mathbf{b}_{\alpha},1)|||\mathrm{FIFO}(\mathbf{a}^{1}_{\gamma},\mathbf{b}_{\gamma},1))\ \mathbf{endhide}\gg\mathbf{t};\mathbf{stop})}$
$[>(((\mathrm{b}^{\beta}_{2};\delta)[](\mathbf{a}^{2}_{\gamma}?2;\delta))\gg(\mathrm{Loop}(\mathrm{b}^{\beta}_{2})|||\mathrm{Loop}(\mathbf{a}^{2}_{\gamma}?2)|||\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\gamma}?1))))[>\delta)$
$|[\mathrm{b}^{\beta}_{2},\mathbf{a}^{2}_{\gamma},\mathbf{t}]|(\boxed{((\mathbf{a}^{2}_{\gamma}!2;\delta)||||(\mathrm{b}^{\beta}_{2};((\mathbf{s}^{\beta}_{\alpha}!2;\delta)||||(\mathbf{s}^{\beta}_{\gamma}!2;\delta))\gg\mathbf{a}^{2}_{\alpha}!2;\delta))}\ [](\mathbf{t};\delta)))$
$|[\mathrm{b}^{\beta}_{1},\mathbf{a}^{1}_{\alpha},\mathbf{a}^{1}_{\gamma},\mathrm{b}^{\beta}_{2},\mathbf{a}^{2}_{\gamma}]|(((((\mathrm{b}^{\beta}_{2};\delta)[](\mathbf{a}^{2}_{\gamma}?2;\delta))\gg(\mathrm{Loop}(\mathrm{b}^{\beta}_{2})|||\mathrm{Loop}(\mathbf{a}^{2}_{\gamma}?2)|||\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\gamma}?1)))[]$
$(((\mathrm{b}^{\beta}_{1};\delta)[](\mathbf{a}^{1}_{\alpha}?1;\delta)[](\mathbf{a}^{1}_{\gamma}?1;\delta))\gg$
$((\mathrm{Loop}(\mathrm{b}^{\beta}_{1})|||\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\gamma}?1))$
$[>(\mathbf{a}^{2}_{\gamma}?2;(\mathrm{Loop}(\mathrm{b}^{\beta}_{2})|||\mathrm{Loop}(\mathbf{a}^{2}_{\gamma}?2)|||\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\gamma}?1))))))$
$[>\delta))$
$|[\mathbf{a}^{1}_{\alpha},\mathbf{a}^{1}_{\gamma},\mathbf{a}^{2}_{\alpha},\mathbf{a}^{2}_{\gamma}]|(((\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)[>\mathrm{Loop}(\mathbf{a}^{2}_{\alpha}?2))||||(\mathrm{Loop}(\mathbf{a}^{1}_{\gamma}?1)[>\mathrm{Loop}(\mathbf{a}^{2}_{\gamma}?2)))[>\delta)$
$\mathbf{endhide\ endren\ endspec}$

$\mathbf{T}_{\gamma}(w_3,\varepsilon) \approx \mathbf{spec\ ren}\ \mathbf{r}^{\gamma}_{\alpha}/\mathbf{a}^{1}_{\alpha}, \mathbf{r}^{\gamma}_{\beta}/\mathbf{a}^{1}_{\beta}, \mathbf{r}^{\gamma}_{\alpha}/\mathbf{a}^{2}_{\alpha}, \mathbf{r}^{\gamma}_{\beta}/\mathbf{a}^{2}_{\beta}\ \mathbf{in\ hide\ t\ in}$
$((((\boxed{(\mathbf{hide}\ \mathbf{b}_{\alpha},\mathbf{b}_{\beta}\ \mathbf{in}\ ((\mathbf{b}_{\alpha}!1;\delta)||||(\mathbf{b}_{\beta}!1;((\mathbf{s}^{\gamma}_{\alpha}!1;\delta)||||(\mathbf{s}^{\gamma}_{\beta}!1;\delta))))$
$|[\mathbf{b}_{\alpha},\mathbf{b}_{\beta}]|(\mathrm{FIFO}(\mathbf{a}^{1}_{\alpha},\mathbf{b}_{\alpha},1)|||\mathrm{FIFO}(\mathbf{a}^{1}_{\beta},\mathbf{b}_{\beta},1))\ \mathbf{endhide}\gg\mathbf{t};\mathbf{stop})}$
$[>(((\mathrm{c}^{\gamma};\delta)[](\mathbf{a}^{2}_{\beta}?2;\delta))\gg(\mathrm{Loop}(\mathrm{c}^{\gamma})|||\mathrm{Loop}(\mathbf{a}^{2}_{\beta}?2)|||\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\beta}?1))))[>\delta)$
$|[\mathrm{c}^{\gamma},\mathbf{a}^{2}_{\beta},\mathbf{t}]|(\boxed{((\mathrm{c}^{\gamma};((\mathbf{s}^{\gamma}_{\alpha}!2;\delta)||||(\mathbf{s}^{\gamma}_{\beta}!2;\delta))\gg\mathbf{a}^{2}_{\alpha}!2;\delta)||||(\mathbf{a}^{2}_{\beta}!2;\delta))}\ [](\mathbf{t};\delta)))$
$|[\mathbf{a}^{1}_{\alpha},\mathbf{a}^{1}_{\beta},\mathrm{c}^{\gamma},\mathbf{a}^{2}_{\beta}]|(((((\mathrm{c}^{\gamma};\delta)[](\mathbf{a}^{2}_{\beta}?2;\delta))\gg(\mathrm{Loop}(\mathrm{c}^{\gamma})|||\mathrm{Loop}(\mathbf{a}^{2}_{\beta}?2)|||\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\beta}?1)))[]$
$(((\mathbf{a}^{1}_{\alpha}?1;\delta)[](\mathbf{a}^{1}_{\beta}?1;\delta))\gg$
$((\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\beta}?1))$
$[>(\mathbf{a}^{2}_{\beta}?2;(\mathrm{Loop}(\mathrm{c}^{\gamma})|||\mathrm{Loop}(\mathbf{a}^{2}_{\beta}?2)|||\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)|||\mathrm{Loop}(\mathbf{a}^{1}_{\beta}?1))))))$
$[>\delta))$
$|[\mathbf{a}^{1}_{\alpha},\mathbf{a}^{1}_{\beta},\mathbf{a}^{2}_{\alpha},\mathbf{a}^{2}_{\beta}]|(((\mathrm{Loop}(\mathbf{a}^{1}_{\alpha}?1)[>\mathrm{Loop}(\mathbf{a}^{2}_{\alpha}?2))||||(\mathrm{Loop}(\mathbf{a}^{1}_{\beta}?1)[>\mathrm{Loop}(\mathbf{a}^{2}_{\beta}?2)))[>\delta)$
$\mathbf{endhide\ endren\ endspec}$

Table 18: An example of distributed choice

(9) $\mathbf{T'}_{c}(b,z) :=$ <u>if</u> $\neg DC(b)$ <u>then</u> $(\mathbf{Term}_{c}(b_1,z)[>\mathbf{Term}_{c}(b_2,z))$
<u>else</u> $\mathbf{ren}\ \cup_{n=1,2}(\{(u^{c}/u^{c}_{n})|(u^{c}\in AS_{c}(b_n))\}+\{(\mathbf{r}^{c}_{c'}/\mathbf{a}^{n}_{c'})|CH^{+}_{c',c}(b_n)\})\ \mathbf{in\ hide\ t\ in}$
$((\mathbf{Const}_{c,1}|[\mathbf{StGt}_{c,2}+\mathbf{RecGt}_{c,2}+\{\mathbf{t}\}]|\mathbf{Const}_{c,2})$
$|[\mathbf{StGt}_{c,2}+\mathbf{RecGt}_{c,2}+\{\mathbf{p}^{c}_{1}\}]|\mathbf{Const}_{c,3})$
$|[\mathbf{RecGt}_{c,1}+\{\mathbf{a}^{2}_{c'}|CH^{+}_{c',c}(b_1)\}]|\mathbf{Const}_{c,4}$
$\mathbf{endhide\ endren}$
<u>where</u> $\mathbf{Const}_{c,3} := ((\mathbf{AllStRec}_{c,2}[](\mathbf{p}^{c}_{1};\mathbf{OneRec}_{c,2}\gg\mathbf{AllStRec}_{c,2}))[>\delta)$
the rest of definitions as in Table 17 <u>endif</u>

Table 19: Mapping $\mathbf{T'}$ for disabling

$SS_{c}(b_2)$ is a service primitive.

The gates on which the starting service primitives and receptions within a $\mathbf{Task}_{c,n}$ occur are listed in $\mathbf{StGt}_{c,n}$ and $\mathbf{RecGt}_{c,n}$, respectively. $\mathbf{OneStRec}_{c,n}$ specifies a process ready to synchronize on one action of $\mathbf{Task}_{c,n}$ on gates from $\mathbf{StGt}_{c,n}$ and $\mathbf{RecGt}_{c,n}$. $\mathbf{AllStRec}_{c,n}$ specifies a processes ready to synchronize on all such actions. Processes $\mathbf{OneRec}_{c,n}$ and $\mathbf{AllRec}_{c,n}$ are analogous to $\mathbf{OneStRec}_{c,n}$ and $\mathbf{AllStRec}_{c,n}$, respectively, except that they refer only to receptions.

$\mathbf{Const}_{c,1}$ prescribes the following: 1) Basically, execute $\mathbf{Task}_{c,1}$ and indicate its successful termination by a $\mathbf{t}$. 2) If $\mathbf{Task}_{c,2}$ starts in the meantime (that will always be before $\mathbf{t}$), stop the basic activity, but remain ready for reception of protocol messages sent to $\mathbf{Task}_{c,1}$. 3) Always be ready to terminate, though $\mathbf{Const}_{c,2}$ will ensure that that

$w = \mathbf{spec}\ ((\mathrm{a}^\alpha;\delta)|||(\mathrm{b}^\beta;\delta))[>((\mathrm{c}^\alpha;\delta)|||(\mathrm{b}^\beta;\delta))\ \mathbf{endspec}$

$w_1 = \mathbf{spec\ hide}\ \mathbf{p}^\alpha,\mathbf{p}^\beta\ \mathbf{in}\ (((\mathrm{a}^\alpha;\delta^\alpha)|||(\mathrm{b}^\beta;\delta^\beta))\gg((\mathbf{p}^\alpha;\delta^\alpha)|||(\mathbf{p}^\beta;\delta^\beta))\gg(\delta^\alpha|||\delta^\beta))[>((\mathrm{c}^\alpha;\delta^\beta)|||(\mathrm{b}^\beta;\delta^\alpha))\ \mathbf{endhide}\ \mathbf{endspec}$

$\mathbf{T}_\alpha(w_1,\varepsilon) \approx \mathbf{spec\ hide}\ \mathbf{p}^\alpha,\mathbf{t}\ \mathbf{in\ ren}\ \mathbf{r}^\alpha_\beta/\mathbf{a}^1_\beta,\mathbf{r}^\alpha_\beta/\mathbf{a}^2_\beta\ \mathbf{in}$

$((((\boxed{(\mathbf{hide}\ \mathbf{b}_\beta\ \mathbf{in}\ (\mathrm{a}^\alpha;((\mathbf{s}^\alpha_\beta!1;\delta)|||(\mathbf{b}_\beta!1;\delta))\gg\mathbf{p}^\alpha;((\mathbf{s}^\alpha_\beta!1;\delta)|||(\mathbf{b}_\beta!1;\delta)))\ |[\mathbf{b}_\beta]|\mathrm{FIFO}(\mathbf{a}^1_\beta,\mathbf{b}_\beta,1)\ \mathbf{endhide}\gg\mathbf{t};\mathbf{stop})})$

$[>(((\mathrm{c}^\alpha;\delta)[](\mathbf{a}^2_\beta?2;\delta))\gg(\mathrm{Loop}(\mathrm{c}^\alpha)|||\mathrm{Loop}(\mathbf{a}^2_\beta?2)|||\mathrm{Loop}(\mathbf{a}^1_\beta?1))))[>\delta)$

$|[\mathrm{c}^\alpha,\mathbf{a}^2_\beta,\mathbf{t}]|(\boxed{((\mathrm{c}^\alpha;\mathbf{s}^\alpha_\beta!2;\delta)|||(\mathbf{a}^2_\beta!2;\delta))}\ [](\mathbf{t};\delta)))$

$|[\mathbf{p}^\alpha,\mathrm{c}^\alpha,\mathbf{a}^2_\beta,\mathbf{t}]|(((\mathrm{Loop}(\mathrm{c}^\alpha)|||\mathrm{Loop}(\mathbf{a}^2_\beta?2))[](\mathbf{p}^\alpha;\mathbf{a}^2_\beta?2;(\mathrm{Loop}(\mathrm{c}^\alpha)|||\mathrm{Loop}(\mathbf{a}^2_\beta?2))))[>\delta))$

$|[\mathbf{a}^1_\beta,\mathbf{a}^2_\beta]|((\mathrm{Loop}(\mathbf{a}^1_\beta?1)[>\mathrm{Loop}(\mathbf{a}^2_\beta?2))[>\delta)\ \mathbf{endren\ endhide\ endspec}$

$\mathbf{T}_\beta(w_1,\varepsilon) \approx \mathbf{spec\ hide}\ \mathbf{p}^\beta,\mathbf{t}\ \mathbf{in\ ren}\ \mathrm{b}^\beta/\mathrm{b}^\beta_1,\mathbf{r}^\beta_\alpha/\mathbf{a}^1_\alpha,\mathrm{b}^\beta/\mathrm{b}^\beta_2,\mathbf{r}^\beta_\beta/\mathbf{a}^2_\alpha\ \mathbf{in}$

$((((\boxed{(\mathbf{hide}\ \mathbf{b}_\alpha\ \mathbf{in}\ (\mathrm{b}^\beta_1;((\mathbf{s}^\beta_\alpha!1;\delta)|||(\mathbf{b}_\alpha!1;\delta))\gg\mathbf{p}^\beta;((\mathbf{s}^\beta_\alpha!1;\delta)|||(\mathbf{b}_\alpha!1;\delta)))\ |[\mathbf{b}_\alpha]|\mathrm{FIFO}(\mathbf{a}^1_\alpha,\mathbf{b}_\alpha,1)\ \mathbf{endhide}\gg\mathbf{t};\mathbf{stop})})$

$[>(((\mathrm{b}^\beta_2;\delta)[](\mathbf{a}^2_\alpha?2;\delta))\gg(\mathrm{Loop}(\mathrm{b}^\beta_2)|||\mathrm{Loop}(\mathbf{a}^2_\alpha?2)|||\mathrm{Loop}(\mathbf{a}^1_\alpha?1))))[>\delta)$

$|[\mathrm{b}^\beta_2,\mathbf{a}^2_\alpha,\mathbf{t}]|(\boxed{((\mathbf{a}^2_\alpha!2;\delta)|||(\mathrm{b}^\beta_2;\mathbf{s}^\beta_\alpha!2;\delta))}\ [](\mathbf{t};\delta)))$

$|[\mathbf{p}^\beta,\mathrm{b}^\beta_2,\mathbf{a}^2_\alpha,\mathbf{t}]|(((\mathrm{Loop}(\mathrm{b}^\beta_2)|||\mathrm{Loop}(\mathbf{a}^2_\alpha?2))[](\mathbf{p}^\beta;\mathbf{a}^2_\alpha?2;(\mathrm{Loop}(\mathrm{b}^\beta_2)|||\mathrm{Loop}(\mathbf{a}^2_\alpha?2))))[>\delta))$

$|[\mathbf{a}^1_\alpha,\mathbf{a}^2_\alpha]|((\mathrm{Loop}(\mathbf{a}^1_\alpha?1)[>\mathrm{Loop}(\mathbf{a}^2_\alpha?2))[>\delta)\ \mathbf{endren\ endhide\ endspec}$

Table 20: An example of distributed disabling

will happen only after successful termination of $\mathbf{Task}_{c,1}$ or $\mathbf{Task}_{c,2}$.

$\mathbf{Const}_{c,2}$ prescribes the following: Execute $\mathbf{Task}_{c,2}$ or terminate upon a $\mathbf{t}$ indicating that $\mathbf{Task}_{c,1}$ has successfully terminated.

$\mathbf{Const}_{c,3}$ in addition prescribes that in the case that the first action belongs to $\mathbf{Task}_{c,1}$, $\mathbf{Task}_{c,2}$ may start only upon a reception, i.e. upon detecting that $\mathbf{Term}(b_2,z)$ has already started at a remote site.

With the described measures for prompt start reporting and for prevention of premature local termination, $\mathbf{T'}_c(b,z)$ will progress towards completion of $\mathbf{Task}_{c,1}$ or $\mathbf{Task}_{c,2}$ as appropriate.

There is, however, still a problem to solve. $\mathbf{Task}_{c,2}$ must not terminate while c may still expect messages sent to $\mathbf{Task}_{c,1}$. So we require that $\mathbf{Task}_{c,2}$ (i.e. $\mathbf{Term}_c(b_2,z)$) never successfully terminates without receiving on each of the channels on which $\mathbf{Term}_c(b_1,z)$ receives. Upon a reception within $\mathbf{Term}_c(b_2,z)$, c knows that on the channel, there will be no more messages for $\mathbf{Term}_c(b_1,z)$. For some channels, the requirement might be redundant.

It is convenient if c indeed promptly becomes unwilling to receive on gates in $\mathbf{RecGt}_{c,1}$, to improve the possibility of re-use of protocol messages belonging to $\mathbf{Term}_c(b_1,z)$. Therefore we introduce $\mathbf{Const}_{c,4}$. An analogous constraint for protocol messages belonging to $\mathbf{Term}_c(b_2,z)$ would also be desirable, but we have found its automatic specification too difficult.

Example 5 *An example of distributed choice is given in Table 18. The original service specification w is gradually transformed into a well-formed specification, following suggestions from Section 4.2. w_1 secures prompt reporting of each individual starting service action. w_2 in addition secures that no component terminates the first alternative until it is selected by components β and γ, the starting components of the second alternative. w_3 in addition secures that every channel employed for the first alternative is also employed for the second one.*

In each individual component specification, the first and the second alternative are highlighted by a box. When divergence occurs, components execute the first alternative, but gradually switch to the other. We see that every protocol message of the first alternative is a 1, and every message of the second one is a 2. All the specified FIFO buffers are redundant.

3.11 Implementation of disabling

For a b specified as "$b_1[> b_2$", we assume that there are service actions (at least a δ) in both parts, so that both are relevant. The operator does not introduce distributed conflicts, formally $\neg DC(b)$, if there is a c which is the only participating component of b_1 and also the only starting component of b_2.

Protocols $\mathbf{Term}(b_1,z)$ and $\mathbf{Term}(b_2,z)$ are combined as for "$b_1[]b_2$", except that $\mathbf{Term}(b_2,z)$ is allowed to start as long as there is a starting component c of b_2 which has not yet detected that b_1 is successfully terminating and confirmed this knowledge by executing a special-purpose service primitive $\mathbf{p}^c$ in b_1.

A participant c combines $\mathbf{Term}_c(b_1,z)$ and $\mathbf{Term}_c(b_2,z)$ as specified in Table 19. If $\neg DC(b)$, activation of $\mathbf{Term}(b_2,z)$ is a local matter of the starting component of b_2. For any other c, $\mathbf{Term}_c(b_1,z)$ is equivalent to **stop**, i.e. the component just waits for an eventual start of $\mathbf{Term}_c(b_2,z)$.

If $DC(b)$, we require that b_1 consists of a regular part b_3 followed by a dummy part b_4 indicating its successful termination (if $\neg TM(b_1)$, b_4 is never activated, and as such not specified), i.e. we pretend that the service we are implementing is actually "$b_3[> b_2$". More precisely, we require

$b_4 = ((|||_{SC_c(b_2)}(\mathbf{p}^c;\delta^c)) \gg (|||_{TC_c^+(b_1)}\delta^c))$

where **p** primitives are supposed to be hidden on a higher service level and not among the visible primitives of b_3. Note that we also prescribe the executor of each individual δ. Since $DC(b)$ and $TM(b_1)$ imply that b in no way synchronizes with concurrent service parts, any $\mathbf{p}^c$ may be regarded entirely as an internal action of $\mathbf{T'}_c(b,z)$.

For such a b_1, protocol $\mathbf{Term}(b_1,z)$ consists of two phases. The first phase is $\mathbf{Term}(b_3,z)$ followed by reporting of successful termination to all the starting components of b_4, i.e. exactly to the starting components of b_2. In other words, the components are, as required, promptly informed when starting of $\mathbf{Term}(b_2,z)$ becomes undesirable. If the first phase successfully terminates before $\mathbf{Term}(b_2,z)$ starts, $\mathbf{T}(b,z)$ starts executing the usual distributed implementation of a well-formed "$b_4[]b_2$". If the start of $\mathbf{Term}(b_2,z)$ is sufficiently delayed, the executed alternative is b_4, i.e. b_1 is not disrupted by b_2. In any case, no participant abandons $\mathbf{Term}(b_2,z)$ until every starting component c of b_2 has executed a $\mathbf{p}^c$, i.e. refused to be an initiator of $\mathbf{Term}(b_2,z)$.

Comparing $\mathbf{T'}_c(b_1[> \ b_2,z)$ with $\mathbf{T'}_c(b_1[]b_2,z)$, we see that, instead of waiting for the starting actions of $\mathbf{Term}_c(b_1,z)$, $\mathbf{Const}_{c,3}$ now waits for the only $\mathbf{p}^c$ in $\mathbf{Term}_c(b_1,z)$, if any. Consequently, instead of synchronizing on the gates in $\mathbf{StGt}_{c,1}$ and $\mathbf{RecGt}_{c,1}$, $\mathbf{Const}_{c,1}$ and $\mathbf{Const}_{c,3}$ have to synchronize just on $\mathbf{p}_1^c$, hence $\mathbf{Const}_{c,3}$ is much easier to specify.

Example 6 *An example of distributed disabling is given in Table 20. To obtain a well-formed service specification, we furnish the first part with the required hidden* **p** *actions, and make sure that the starting actions of the second part are promptly reported and that both protocol channels are used for the part.*

4 Computation and tuning of service attributes

4.1 Attribute evaluation rules

The attributes in Table 21 provide information on service actions and their executors. SS_c and AS_c respectively list for an a, b or p its starting service actions and all its service actions at c. SC_c and PC_c respectively indicate for an a or b that c is its starting component or its participating component.

The attributes in Table 22 provide information on successful terminations. TM, IT and DT respectively indicate for a b or p that it might successfully terminate, that it might terminate initially, or that the termination might be decisive.

The attributes in Table 23 provide information on distributed conflicts. DC indicates for a b that distributed conflicts are introduced by its top-level composition operator. AD and TD respectively indicate for a b or p whether there are any distributed conflicts in it and whether there are distributed conflicts involving its successful termination.

The attribute SR_c in Table 24 indicates for a b or p that its start must be promptly reported to c.

The attribute EC_c in Table 25 indicates for a b or p that c is its ending component for mapping **T**. EC_c^+ is the analogue for mapping **Term**.

No.	SS_c	No.	SS_c
(2)	$SS_c(p) = SS_c(b)$	(4)	$SS_c(b) = \{\delta \mid PC_c(b)\}$
(3)	$SS_c(b) = \emptyset$	(6)	$SS_c(b) = SS_c(a)$
(12)	$SS_c(b) = SS_c(p)$	(13)	$SS_c(a) = \{a \mid PC_c(a)\}$
(7)	$SS_c(b) = ((SS_c(b_1) \setminus S) \cup (SS_c(b_2) \setminus S) \cup (SS_c(b_1) \cap SS_c(b_2) \cap S))$		
(8,9)	$SS_c(b) = (SS_c(b_1) \cup SS_c(b_2))$		
(5)	$SS_c(b) = ((SS_c(b_1) \setminus \{\delta\}) \cup \{\mathbf{i} \mid (\delta \in SS_c(b_1))\})$		
(10)	$SS_c(b) = ((SS_c(b_1) \setminus S) \cup \{\mathbf{i} \mid ((SS_c(b_1) \cap S) \neq \emptyset)\})$		
(11)	$SS_c(b) = ((SS_c(b_1) \setminus \{s \mid \exists (s'/s) \in R\}) \cup \{s' \mid \exists s \in SS_c(b_1) : ((s'/s) \in R)\})$		
No.	AS_c	No.	AS_c
(2)	$AS_c(p) = AS_c(b)$	(4)	$AS_c(b) = SS_c(b)$
(3)	$AS_c(b) = \emptyset$	(12)	$AS_c(b) = AS_c(p)$
(5)	$AS_c(b) = ((AS_c(b_1) \setminus \{\delta\}) \cup \{\mathbf{i}\} \cup AS_c(b_2))$		
(6)	$AS_c(b) = (SS_c(a) \cup AS_c(b_2))$		
(7–9)	$AS_c(b) = (AS_c(b_1) \cup AS_c(b_2))$		
(10)	$AS_c(b) = ((AS_c(b_1) \setminus S) \cup \{\mathbf{i} \mid ((AS_c(b_1) \cap S) \neq \emptyset)\})$		
(11)	$AS_c(b) = ((AS_c(b_1) \setminus \{s \mid \exists (s'/s) \in R\}) \cup \{s' \mid \exists s \in AS_c(b_1) : ((s'/s) \in R)\})$		
(3–12)	$(SC_c(b) = (SS_c(b) \neq \emptyset)) \wedge (PC_c(b) = (AS_c(b) \neq \emptyset))$		
(4)	$\exists c : (PC(b) = \{c\})$		
(13)	$(\exists c : (PC(a) = \{c\})) \wedge ((\exists u : (a = u^c)) \Rightarrow PC_c(a))$		

Table 21: Service actions and their executors

No.	DT	No.	DT
(2)	$DT(p) = DT(b)$	(10,11)	$DT(b) = DT(b_1)$
(3,4)	$DT(b) = false$	(12)	$DT(b) = DT(p)$
(5,6)	$DT(b) = DT(b_2)$	(7)	$DT(b) = (DT(b_1) \vee DT(b_2))$
(8)	$DT(b) = (DT(b_1) \vee DT(b_2) \vee IT(b))$		
(9)	$DT(b) = (TM(b_1) \vee IT(b_2) \vee DT(b_2))$		
(3–12)	$TM(b) = \exists c : (\delta \in AS_c(b))$		
(3–12)	$IT(b) = \exists c : (\delta \in SS_c(b))$		

Table 22: Successful terminations

No.	AD	No.	AD
(2)	$AD(p) = AD(b)$	(3,4)	$AD(b) = false$
(6)	$AD(b) = AD(b_2)$	(5,7)	$AD(b) = (AD(b_1) \vee AD(b_2))$
(12)	$AD(b) = AD(p)$	(10,11)	$AD(b) = AD(b_1)$
(8,9)	$AD(b) = (AD(b_1) \vee AD(b_2) \vee DC(b))$		
No.	TD	No.	TD
(2)	$TD(p) = TD(b)$	(10,11)	$TD(b) = TD(b_1)$
(3,4)	$TD(b) = false$	(12)	$TD(b) = TD(p)$
(5,6)	$TD(b) = TD(b_2)$	(7)	$TD(b) = (TD(b_1) \vee TD(b_2))$
(8)	$TD(b) = (TD(b_1) \vee TD(b_2) \vee (DC(b) \wedge IT(b)))$		
(9)	$TD(b) = (TD(b_2) \vee (DC(b) \wedge (TM(b_1) \vee IT(b_2))))$		
(8)	$DC(b) := (\lvert SC(b) \rvert > 1)$		
(9)	$DC(b) := (\lvert PC(b_1) \cup SC(b_2) \rvert > 1)$		

Table 23: Distributed conflicts

No. SR_c	No.	SR_c
(1) $SR_c(b) = false$	(5,7,9–11)	$SR_c(b_1) = SR_c(b)$
(2) $SR_c(b) = SR_c(p)$	(5,6)	$SR_c(b_2) = false$
(7) $SR_c(b_2) = SR_c(b)$	(12)	$SR_c(p) = (SR_c(p) \vee SR_c(b))$
(8) $SR_c(b_1) = (SR_c(b) \vee SC_c(b_2))$		
(8) $SR_c(b_2) = (SR_c(b) \vee SC_c(b_1) \vee (DC(b) \wedge PC_c(b_1)))$		
(9) $SR_c(b_2) = (SR_c(b) \vee PC_c(b_1))$		

Table 24: Start reporting

No.	EC_c	No.	EC_c
(2)	$EC_c(p) = EC_c^+(b)$	(5,6)	$EC_c(b) = EC_c^+(b_2)$
(3)	$EC_c(b) = false$	(10,11)	$EC_c(b) = EC_c^+(b_1)$
(4)	$EC_c(b) = PC_c(b)$	(12)	$EC_c(b) = EC_c(p)$
(7–9)	$EC_c(b) = (EC_c^+(b_1) \vee EC_c^+(b_2))$		
(3–12)	$EC_c^+(b) = ((EC_c(b) \wedge \nexists c' : RT_{c'}(b)) \vee RT_c(b))$		

Table 25: Ending components

No.	TC_c^+	No.	TC_c^+
(1)	$TC_c^+(b) = TM(b)$	(2)	$TC_c^+(b) = TC_c^+(p)$
(5)	$TC_c^+(b_1) = (EC_c(b_1) \vee PC_c(b_2) \vee TC_c(b))$		
(5–9)	$TC_c^+(b_2) = (TC_c(b) \wedge PC_c(b) \wedge TM(b_2))$		
(7–11)	$TC_c^+(b_1) = (TC_c(b) \wedge PC_c(b) \wedge TM(b_1))$		
(12)	$TC_c^+(p) = (TC_c^+(p) \wedge TC_c(b))$		
No.	TC_c		
(3–6,10,11)	$TC_c(b) = TC_c^+(b)$		
(7)	$TC_c(b) = (TC_c^+(b) \wedge (EC_c(b) \vee \neg PC_c(b) \vee (\neg DT(b) \wedge \nexists c' : SH_{c',c}(b))))$		
(8,9)	$TC_c(b) = (TC_c^+(b) \wedge (((\forall c' \in SC(b_2) : GT_{c,c'}^+(b_1)) \wedge (\neg DC(b) \vee ((EC_c(b) \vee \neg TM(b_1)) \wedge \nexists c' : (CH_{c',c}^+(b_1) \wedge \neg CT_{c',c}^+(b_2)))) \wedge (\neg TM(b_2) \vee PC_c(b_2))) \vee \neg PC_c(b)))$		
(12)	$TC_c(b) = (TC_c^+(b) \wedge (TC_c^+(p) \vee \neg PC_c(b)))$		
(3–12)	$RT_c(b) = (TC_c^+(b) \wedge \neg TC_c(b))$		

Table 26: Termination types

The attributes in Table 26 provide information on termination types. TC_c and TC_c^+ respectively indicate for a b or p that c is its terminating component for mapping **T** or **Term**. RT_c indicates for a b that c detects its termination upon receiving a special report on it.

The attributes in Table 27 provide information on utilization of protocol channels. $CH_{c,c'}$ and $CH_{c,c'}^+$ respectively indicate for a b or p that mapping **T** or **Term** introduces protocol messages on the channel from c to c'. $CT_{c,c'}$ and $CT_{c,c'}^+$ respectively indicate that the channel is used in every successfully terminating run. For a b consisting of two competing parts, $SH_{c,c'}$ indicates if the channel is shared.

The attributes $GT_{c,c'}$ and $GT_{c,c'}^+$ in Table 28 respectively indicate for a b or p that in mapping **T** or **Term**, its successful termination at c is guarded by c'.

By the rules in Table 29, we choose for a b such identifiers CI and CI^+ that all protocol messages introduced by mapping **T** or **Term**, respectively, are dynamically unique.

No. $CH_{c,c'}$	No.	$CH_{c,c'}$
(2) $CH_{c,c'}(p) = CH_{c,c'}^+(b)$	(10,11)	$CH_{c,c'}(b) = CH_{c,c'}^+(b_1)$
(3,4) $CH_{c,c'}(b) = false$	(12)	$CH_{c,c'}(b) = CH_{c,c'}(p)$
(5) $CH_{c,c'}(b) = (CH_{c,c'}^+(b_1) \vee CH_{c,c'}^+(b_2) \vee ((c \neq c') \wedge EC_c^+(b_1) \wedge SC_{c'}(b_2)))$		
(6) $CH_{c,c'}(b) = (CH_{c,c'}^+(b_2) \vee ((c \neq c') \wedge PC_c(a) \wedge SC_{c'}(b_2)))$		
(7–9) $CH_{c,c'}(b) = (CH_{c,c'}^+(b_1) \vee CH_{c,c'}^+(b_2))$		
No. $CT_{c,c'}$	No.	$CT_{c,c'}$
(2) $CT_{c,c'}(p) = CT_{c,c'}^+(b)$	(10,11)	$CT_{c,c'}(b) = CT_{c,c'}^+(b_1)$
(3) $CT_{c,c'}(b) = true$	(12)	$CT_{c,c'}(b) = CT_{c,c'}(p)$
(4) $CT_{c,c'}(b) = false$		
(5) $CT_{c,c'}(b) = (CT_{c,c'}^+(b_1) \vee CT_{c,c'}^+(b_2) \vee ((c \neq c') \wedge EC_c^+(b_1) \wedge SC_{c'}(b_2)))$		
(6) $CT_{c,c'}(b) = (CT_{c,c'}^+(b_2) \vee ((c \neq c') \wedge PC_c(a) \wedge SC_{c'}(b_2)))$		
(7) $CT_{c,c'}(b) = (CT_{c,c'}^+(b_1) \vee CT_{c,c'}^+(b_2))$		
(8,9) $CT_{c,c'}(b) = (CT_{c,c'}^+(b_1) \wedge CT_{c,c'}^+(b_2))$		
(3–12) $CH_{c,c'}^+(b) = (CH_{c,c'}(b) \vee (EC_c(b) \wedge RT_{c'}(b)))$		
(3–12) $CT_{c,c'}^+(b) = (CT_{c,c'}(b) \vee (EC_c(b) \wedge RT_{c'}(b)))$		
(7–9) $SH_{c,c'}(b) = (CH_{c,c'}^+(b_1) \wedge CH_{c,c'}^+(b_2))$		

Table 27: Channel utilization

No. $GT_{c,c'}$	No.	$GT_{c,c'}$
(2) $GT_{c,c'}(p) = GT_{c,c'}^+(b)$	(10,11)	$GT_{c,c'}(b) = GT_{c,c'}^+(b_1)$
(3) $GT_{c,c'}(b) = true$	(12)	$GT_{c,c'}(b) = GT_{c,c'}(p)$
(4) $GT_{c,c'}(b) = (\neg TC_c(b) \vee ((c = c') \wedge PC_c(b)))$		
(5) $GT_{c,c'}(b) = (GT_{c,c'}^+(b_1) \vee GT_{c,c'}^+(b_2) \vee (PC_c(b_2) \wedge \exists c'' : (EC_{c''}^+(b_1) \wedge GT_{c'',c'}^+(b_1))))$		
(6) $GT_{c,c'}(b) = ((PC_{c'}(a) \wedge ((c = c') \vee PC_c(b_2))) \vee GT_{c,c'}^+(b_2))$		
(7) $GT_{c,c'}(b) = (GT_{c,c'}^+(b_1) \vee GT_{c,c'}^+(b_2))$		
(8,9) $GT_{c,c'}(b) = (GT_{c,c'}^+(b_1) \wedge GT_{c,c'}^+(b_2))$		
(3–12) $GT_{c,c'}^+(b) = (\neg TC_c^+(b) \vee (TC_c(b) \wedge GT_{c,c'}(b)) \vee (\neg TC_c(b) \wedge \exists c'' : (EC_{c''}(b) \wedge GT_{c'',c'}(b))))$		

Table 28: Termination guarding

No. CI^+	No.	CI^+
(1,2) $CI^+(b) = \varepsilon$	(5,6)	$CI^+(b_2) = CI(b)$
(5,10,11) $CI^+(b_1) = CI(b)$		
(7–9) if $\exists c, c' : SH_{c,c'}(b)$ then $CI^+(b_1) = CI(b) \cdot 1$, $CI^+(b_2) = CI(b) \cdot 2$ else $CI^+(b_1) = CI^+(b_2) = CI(b)$ endif		
No. CI		
(3–6,10,11) $CI(b) = CI^+(b)$		
(7–9) if $(((CI^+(b_1) \neq CI(b)) \wedge (CI^+(b_2) \neq CI(b))) \vee \nexists c, c' : (TC_c^+(b) \wedge RT_{c'}(b) \wedge CH_{c,c'}(b)))$ then $CI(b) = CI^+(b)$ else $CI(b) = CI^+(b) \cdot 1$ endif		
(12) if $\nexists c, c' : (TC_c^+(b) \wedge RT_{c'}(b) \wedge CH_{c,c'}(b))$ then $CI(b) = CI^+(b)$ else $CI(b) = CI^+(b) \cdot 1$ endif		

Table 29: Context identifiers

Attribute evaluation rules for a service specification con-

stitute a system of equations which might have more than one solution for the attributes of the explicitly defined processes. One should always maximize their attribute TC^+, while other attributes must be minimized.

4.2 Additional restrictions and their satisfaction

Table 30 summarizes the additional restrictions introduced so far for a well-formed service specification.

The first three restrictions state that no irrelevant service part may be specified. The restriction for parallel composition is actually more rigorous than its approximation in Table 30 (see Section 3.9).

The next two restrictions refer to the ending components of a b. Usually they can be satisfied simply by proper choice of executors for individual δ in b, but not always. It might be that a "$b_1[]b_2$" or a "$b_1[> b_2$" is terminating, but no c qualifies for its ending component, because a $GT^+_{c,c'}(b_1)$ or $PC_c(b_2)$ or a $CT^+_{c',c}(b_2)$ is not true as required. $GT^+_{c,c'}(b_1)$ can be satisfied by securing that in the terminating runs of b_1, the last (possibly dummy) action at c always comes after a (possibly dummy) action at c'. For $PC_c(b_2)$, it suffices to insert into b_2 a dummy action at c. For $CT^+_{c',c}(b_2)$, it helps to introduce into every terminating run of b_2 an action at c prefixed by an action at c'.

The next two restrictions require that there are hidden **p** primitives at certain places in the service specification. If **p** primitives are already used for other purposes, any other reserved service primitive type will do.

The next restriction states that a b with distributed conflicts must not synchronize with a concurrent service part, in order to avoid deadlock resulting from imprecise implementation of b. However, if the concurrent service part is sufficiently flexible (like, for example, a skilled user of an imprecisely implemented service), there will be no deadlock and the restriction may be ignored.

The next two restrictions secure prompt start reporting. An ordinary action a is always specified in a context "$a; b_2$". A report recipient c must be the executor of a or a starting component of b_2, so that the message will be generated to implement the action-prefix operator. If a c is a missing starting component of b_2, that can be solved by introducing into b_2 a dummy starting service action at c. For reporting of a δ, there is no such b_2 following, so we have only the first option.

In a general case, execution of a disruptive b might start by concurrent execution and reporting of several starting actions. To avoid as much as possible such multiple reporting of the start of b, it is advisable to rewrite the specification of b into the action-prefix form (as required in [10] for b_2 in a "$b_1[> b_2$"), i.e. make sure that $AP(b)$ (defined in Table 31).

The last two restrictions state that a service action in a particular position must not be an **i** or a δ. If it is an **i**, change it into a service primitive and hide it on a higher level. If it is a δ, prefix it with a subsequently hidden service primitive. For both cases, $DC(b)$ implies that b runs in such a context that the transformation is irrelevant.

(5)	$TM(b_1)$
(7)	$((\cup_{c\in\mathcal{C}} AS_c(b_1)) \cap (S + \{\delta\}))$
	$= ((\cup_{c\in\mathcal{C}} AS_c(b_2)) \cap (S + \{\delta\}))$
(8,9)	$(\lvert PC(b_1)\rvert > 0) \wedge (\lvert PC(b_2)\rvert > 0)$
(7)	$DT(b) \Rightarrow (\lvert EC(b)\rvert = 1)$
(3–12)	$EC_c(b) \Rightarrow TC_c(b)$
(1)	$\nexists c : (\mathbf{p}^c \in AS_c(b))$
(9)	$DC(b) \Rightarrow \exists b_3 :$
	$((b_1 = (b_3$ if $TM(b_1)$ then
	$\gg(\lvert\lvert\lvert_{SC_c(b_2)}(\mathbf{p}^c; \delta^c)) \gg (\lvert\lvert\lvert_{TC^+_c(b_1)} \delta^c)$
	endif)
	$\wedge \nexists c : (\mathbf{p}^c \in AS_c(b_3)))$
(7)	$((S \neq \emptyset) \Rightarrow \neg AD(b)) \wedge \neg TD(b)$
(4)	$SR_c(b) \Rightarrow PC_c(b)$
(6)	$SR_c(b) \Rightarrow (PC_c(a) \vee SC_c(b_2))$
(8)	$DC(b) \Rightarrow (SC_c(b_2) \Rightarrow (((\{\mathbf{i}, \delta\} \cap SS_c(b_1)) = \emptyset))$
(8,9)	$DC(b) \Rightarrow (PC_c(b_1) \Rightarrow (((\{\mathbf{i}, \delta\} \cap SS_c(b_2)) = \emptyset))$

Table 30: Restrictions

No.	AP	No.	AP
(2)	$AP(p) = AP(b)$	(3,4,6)	$AP(b) = true$
(7,9)	$AP(b) = false$	(8)	$AP(b) = (AP(b_1) \wedge AP(b_2))$
(12)	$AP(b) = AP(p)$	(5,10,11)	$AP(b) = AP(b_1)$

Table 31: Action-prefix form

5 Discussion and conclusions

5.1 Correctness

A formal proof of the protocol derivation method is given in [18], and briefly outlined below.

For every service part b, the only property that really matters is correctness of its $\mathbf{T}'$ and $\mathbf{Term}$ implementations for the context in which it is embedded, where a $\mathbf{T}'$ implementation consists of the members of $PC(b)$, while a $\mathbf{Term}$ implementation might also involve some other server components. However, when proving the property, we also assume over twenty auxiliary properties of the implementations.

All the properties are proven by induction on the service structure. Most of them are synthesized properties. We prove them for the $\mathbf{T}'$ implementations of **stop** and δ. For every composite b (i.e. for every service composition operator), we prove that if **Term** implementations of the constituent service parts possess the properties, the $\mathbf{T}'$ implementation of b possesses their analogues. In addition we prove that if the $\mathbf{T}'$ implementation of a b possesses the properties, its $\mathbf{Term}$ implementations possess their analogues. For the few inherited properties, the proof goes in the reverse direction. By proving the main property for the main service process, we prove that the entire service is properly implemented.

5.2 Message complexity

The operators potentially introducing protocol messages are the operators of sequence, choice and disabling. It is often possible to reduce the number of such operators by restructuring the service specification, i.e. by making its inherent parallelism more explicit. If such restyling of the service (and consequently of the protocol) is not unacceptable for readability reasons, it can greatly reduce the message complexity, and can even be automated [25]. One should also strive for optimal insertion of dummy service actions and optimal assignment of hidden service actions to server components.

Anyway, some of the messages introduced by our protocol derivation mapping are redundant.

- In some cases, it would be possible to omit a message based on the observation that for the service part b_1 to which it belongs, it sequences two service actions which are already sequenced for a concurrent service part b_2 synchronized on them [13].
- It would be possible to further optimize terminations of implementations of individual service parts, and their reporting in individual runs [14, 24].
- When implementing a "$b_1 [] b_2$", one could make better use of the fact that only the initial parts of b_1 and b_2 are concurrent.
- When implementing a "$b_1 [> b_2$", one could make better use of the fact that only the initial part of b_2 is concurrent to b_1.

With more extensive re-use of messages, their encodings could be shorter, but messages would no longer directly identify the service part to which they belong, leading to more complicated protocol specifications.

5.3 Comparison with similar methods

The popular formal technique for specifying self-stabilizing protocols have long been finite state machines (FSMs) [6, 27, 22]. With their explicit representation of states, they are very convenient for the purpose. Namely, when a process proceeds along a selected path in the transition graph representing its FSM, the fact that it ignores messages belonging to the abandoned paths can be specified simply by furnishing each state on the selected path with loops representing reception of such messages. In a process-algebraic language like LOTOS, there is no explicit notion of states, so specification of self-stabilization is a tricky task.

There are two basic approaches to deriving self-stabilizing protocols. In the older approach [6, 27], a protocol is first derived for the ideal case with no divergences and subsequently furnished with the reception-ignoring loops. The derivation algorithm in [22], like ours, handles the ideal and the non-ideal cases in an integrated manner, and is consequently much less complex. Moreover, the algorithm derives protocols in a compositional way, supporting implementation of sequence, choice and iteration. For those operators, the structure of services is quite well reflected in the derived protocols. Unfortunately, FSMs are less suited for explicit specification of more complex operators, particularly for such introducing concurrency. We have solved the problem by switching to the more expressive LOTOS.

We know no comparable LOTOS-based protocol derivation transformation. Some hidden divergence is allowed in [1], but it is resolved with the help of global controllers.

5.4 Handling of data

We intend to extend our method to service actions associated with data [5, 11], to approach the ideal that the service specification language should be the same as the protocol specification language. The strategy for flexible integrated handling of messages implementing proper ordering of actions and those carrying data is simple [11]: 1) In the service, identify the points where inter-component exchange of data would be desirable. 2) At each point, introduce a (possibly dummy) action of the data sender followed by a (possibly dummy) action of the data recipient, so that there will be an action-ordering message between the two components. 3) Let the message carry the data. In our case, data could also be carried in a message reporting termination of a b to a c with $RT_c(b)$.

Data exchange is also desirable as a powerful means for compositional service specification. Whenever the more specific operators (e.g. sequential composition, choice and disabling) do not suffice for describing a particular kind of composition of a set of service parts, one can still run the parts in parallel and let them exchange and process information on their respective states.

5.5 Handling of quantitative temporal constraints

Once being able to handle service actions with data, one can easily implement quantitative temporal constraints [12, 23]. Such a constraint specifies the allowed time gap between two service actions. So the time when the first action is executed is just another piece of data generated by the first action and needed for timely execution of the second one. Temporal constraints can also be employed for preventing distributed conflicts and for further optimization of protocol traffic [23].

5.6 The problem of co-ordinated self-stabilization

The most difficult challenge for future research seems to be implementation of self-stabilization after divergence in synchronized service parts. The problem is important because synchronized processes are the core of the constraint-

oriented specification style, that is indispensable for expressing more exotic forms of service composition. To solve it in a general case, one would need a protocol incorporating negotiation of implementations of concurrent service parts, so an enhancement along the lines of [29] could help.

5.7 Conclusions

Automatic implementation of self-stabilization after divergence is an important achievement in LOTOS-based protocol derivation, because many realistic services contain distributed conflicts (e.g. a connection establishment service with both parties as possible initiators). In the era of service integration, the problem is even more acute, because one often wishes to combine services which are not exactly compatible. Take for example feature interactions in telecommunications, which can be nicely detected and managed based on specifications in LOTOS [4]. With the possibility of compositional derivation of self-stabilizing protocols, it suffices to specify dynamic management of such interactions on the service level.

In our future work, we will focus on protocol derivation in E-LOTOS [8], the enhanced successor of LOTOS, because it supports specification of real-time aspects.

References

[1] Bista BB, Takahashi K, Shiratori N: A compositional approach for constructing communication services and protocols. IEICE Transactions on Fundamentals E82-A(11):2546–2557 (1999)

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

[3] Brinksma E, Langerak R: Functionality decomposition by compositional correctness preserving transformation. South African Computer Journal 13:2–13 (1995)

[4] Dietrich F, Hubaux J-P: Formal methods for communication services: meeting the industry expectations. Computer Networks 38(1):99–120 (2002)

[5] Gotzhein R, Bochmann Gv: Deriving protocol specifications from service specifications including parameters. ACM Transactions on Computer Systems 8(4):255–283 (1990)

[6] Gouda MG, Yu YT: Synthesis of communicating finite-state machines with guaranteed progress. IEEE Trans. on Communications COM-32(7):779–788 (1984)

[7] ISO/IEC: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, 1989

[8] ISO/IEC: Information Technology - Enhancements to LOTOS (E-LOTOS). IS 15473, 2001

[9] Kahlouche H, Girardot JJ: A stepwise refinement based approach for synthesizing protocol specifications in an interpreted Petri net model. Proceedings of IEEE INFOCOM'96, pp 1165–1173, 1996

[10] Kant C, Higashino T, Bochmann Gv: Deriving protocol specifications from service specifications written in LOTOS. Distributed Computing 10(1):29-47 (1996)

[11] Kapus-Kolar M: Deriving protocol specifications from service specifications including parameters. Microprocessing and Microprogramming 32:731–738 (1991)

[12] Kapus-Kolar M: Deriving protocol specifications from service specifications with heterogeneous timing requirements. Proceedings SERTS'91. IEE, London 1991, pp 266–270

[13] Kapus-Kolar M: On context-sensitive service-based protocol derivation. Proceedings of MELECON'96. IEEE Computer Society Press 1996, pp 955–958

[14] Kapus-Kolar M: More efficient functionality decomposition in LOTOS. Informatica (Ljubljana) 23(2):259–273 (1999)

[15] Kapus-Kolar M: Comments on deriving protocol specifications from service specifications written in LOTOS. Distributed Computing 12(4):175–177 (1999)

[16] Kapus-Kolar M: Service-based synthesis of two-party protocols. Elektrotehniški vestnik 67(3):153–161 (2000)

[17] Kapus-Kolar M: Global conflict resolution in automated service-based protocol synthesis. South African Computer Journal 27:34–48 (2001)

[18] Kapus-Kolar M: Deriving self-stabilizing protocols for services specified in LOTOS. Technical Report #8476, Jožef Stefan Institute, Ljubljana, 2003

[19] Khendek F, Bochmann Gv, Kant C: New results on deriving protocol specifications from service specifications. Proceedings of ACM SIGCOMM'89, pp 136–145, 1989

[20] Langerak R: Decomposition of functionality: A correctness-preserving LOTOS transformation. Protocol Specification, Testing and Verification X. North-Holland, Amsterdam 1990, pp 203–218

[21] Naik K, Cheng Z, Wei DSL: Distributed implementation of the disabling operator in LOTOS. Information and Software Technology 41(3):123–130 (1999)

[22] Nakamura M, Kakuda Y, Kikuno T: On constructing communication protocols from component - based service specifications. Computer Communications 19(14):1200–1215 (1996)

[23] Nakata A, Higashino T, Taniguchi K: Protocol synthesis from timed and structured specifications. Proceedings of ICNP'95. IEEE Computer Society Press 1995, pp 74–81

[24] Nakata A, Higashino T, Taniguchi K: Protocol synthesis from context-free processes using event structures. Proceedings RTCSA'98. IEEE Computer Society Press 1998, pp 173–180

[25] Pavón Gomez S, Hulström M, Quemada J, de Frutos D, Ortega Mallen Y: Inverse expansion. Formal Description Techniques IV. North-Holland, Amsterdam 1992, pp 297-312

[26] Saleh K: Synthesis of communication protocols: An annotated bibliography. Computer Communication Review 26(5):40–59 (1996)

[27] Saleh K, Probert RL: An extended service-based method for the synthesis of protocols. Proceedings of the Sixth Bilkent Intern. Symp. on Computer and Information Sciences. Elsevier, Amsterdam 1991, pp 547–557

[28] Vissers CA, Scollo G, Sinderen Mv: Specification styles in distributed systems design and verification. Theoretical Computer Science 89:179–206 (1991)

[29] Yasumoto K, Higashino T, Taniguchi K: A compiler to implement LOTOS specifications in distributed environments. Computer Networks 36(2–3):291–310 (2001)