Global Conflict Resolution in Automated Service-Based Protocol Synthesis

M. Kapus-Kolar

Jožef Stefan Institute, Ljubljana, Slovenia
monika.kapus-kolar@ijs.si

Abstract

A transformation is proposed which, given a specification of the required external behaviour of a server consisting of two synchronously communicating components and a partitioning of the specified service actions among the server components, derives behaviour of individual components, i.e. a protocol implementing the service. The adopted specification language is an abstraction of E-LOTOS. The transformation accepts service specifications written in its Basic-LOTOS-like sublanguage. The stress is on demonstrating that distributed decision-making can be implemented without preventing the derived protocol specifications from reflecting the structure of the service specifications.
Keywords: *Distributed service implementation, Automated protocol synthesis, LOTOS, E-LOTOS*
Computing Review Categories: *C.2.2, C.2.4*

1 Introduction

The present work is an enhancement of [2], where Brinksma and Langerak propose a compositional correctness preserving transformation for conducting functionality decomposition based on specifications written in LOTOS [1], a standard algebraic language primarily intended for specification of concurrent and reactive systems. The problem they address is usually referred to as *protocol derivation* (service-based protocol synthesis) and defined as follows:

A *distributed server* (that might be a material or a logical, a technical or a social system) is characterised by its external behaviour, i.e. the *service*, and by the external behaviours of its co-operating components, i.e. the *protocol* implementing the service. Given a specification of the expected service and a partitioning of the specified service actions among the server components, the task is to derive a suitable protocol.

Transformations for protocol derivation are becoming increasingly important, as they facilitate rapid prototyping of new distributed services for specific user needs, that is an imperative particularly in modern intelligent telecommunications networks and global computer networks. If a transformation is implemented in a tool, it can even be employed by non-specialists.

When trying to automate protocol derivation, it is desirable to simultaneously pursue four often conflicting goals:

1) to stay within the limits of the adopted formal description technique (FDT),

2) to minimise the restrictions on the distributed server architecture, on the service specification structure and on the partitioning of service actions among the server components,

3) to reflect the structure of service specifications in the derived protocol specifications, and

4) to minimise the inter-component communication.

Since the middle eighties, protocol synthesis has been subject to intensive research. Numerous methods are discussed in [2], and an even more exhaustive survey can be found in [13], thus in the present paper we provide no systematic overview of the existing methods and refer to them only where necessary for evaluation of the proposed solutions. Many of the methods are based on LOTOS-like languages.

Like [2], we assume that a server consists of *two components* exchanging the necessary protocol messages *synchronously* through a special gate. Synchronous communication means that an interaction is always an atomic common action executed when both partners are ready for it. Communication between the server and its users is also supposed to be synchronous, consisting of *service primitives* (SPs), i.e. atomic interactions between a server component and a service user it supports. The motivation for limiting the discussion to two-party servers with synchronous communication has been to present a solution to a difficult service-implementation problem in the simplest possible setting.

The adopted FDT is a syntactically simplified sublanguage of E-LOTOS [3], an enhanced version of LOTOS currently approaching its standardisation, though we study distributed implementation only for those behaviour types that E-LOTOS has inherited (with slight semantic changes) from Basic LOTOS, i.e. from LOTOS with parametrisation and value-

passing ignored. Judging correctness of a distributed service implementation, we require that the external behaviour of the server is *observation equivalent* (i.e. $\approx$) [11] to the specified service.

Our enhancement over [2] is handling of *global conflicts*. Two SPs are conflicting if one of them is *disruptive* for the other (i.e. its execution makes subsequent occurrence of the other SP illegal). If the conflict is *local* to a server component, it can be resolved by the component itself [2]. If the SPs belong to different server components, the conflict is *global* and requires *distributed decision-making*.

Example 1 *Suppose that there are server components* 1 *and* 2 *respectively controlling execution of SPs* a^1 *and* d^2*, and expected to offer them for execution as alternatives. In the absence of a proper decision-making procedure, it might happen that* d^2 *is executed before component* 1 *manages to report to component* 2 *that* a^1 *has been executed, implying that* d^2 *must be immediately disabled.*

The paper is organised as follows: In Section 2 we introduce the adopted specification language. Section 3 explains how to identify the global conflicts that arise when a given service is executed in a distributed manner. Section 4 shows how the components of a distributed server can resolve global conflicts by synchronising on a virtual token. Section 5 proposes a transformation for deriving protocol specifications from service specifications, but doesn't tell how to specify token management. The missing issue is handled in Section 6. Section 7 brings a discussion and conclusions.

2 Specification Language and Formal Problem Definition

The employed language, defined in Table 1 in a Backus-Naur-like form, is an abstract representation of some E-LOTOS constructs, in the exclusive setting of the protocol derivation problem. Not shown in the table is that parentheses may be used to control parsing.

In the following, if c is one of the server components, c' refers to the other one and $\mathcal{C} = \{c, c'\}$. Likewise, if n is an element of $\{1, 2\}$, n' is the other one.

A b denotes a behaviour, i.e. a process exhibiting it, for instance the server as a whole, an individual server component or some other partial server behaviour. $Init(b)$ lists its possible initial actions and exceptions.

The basic behaviour types are successful termination δ, an individual action a (implicitly followed by δ), and an exception x. An action is basically denoted by the interaction gate on which it occurs, but might also carry a parameter. By an "**any** : V" we specify for an action parameter that any value from the set V is acceptable for it. For an "**any**:$\{v\}$", v is defined as a shorthand.

Name of the construct	Syntax definition
Specification	$::= d^+$
Process definition	$d ::= p := b$
Process name	$p ::= <identifier>$
Behaviour	$b ::=$
Ordinary action	a
Successful termination	$\mid \delta$
Exception raising	$\mid x$
Sequential composition	$\mid b_1; b_2$
Iteration	$\mid$ **loop** b_1
Choice	$\mid b_1 [] b_2$ where $\delta \notin (Init(b_1) \cup Init(b_2))$
Suspend/resume	$\mid b_1 [> b_2$ where $((Init(b_2) \cap \{\delta, x\}) = \emptyset)$
Disabling	$\mid b_1 [> b_2$ where $\delta \notin Init(b_2)$
Parallel composition	$\mid b_1 \|[g^*]\| b_2$
Gate hiding	$\mid$ **hide** g^* **in** b_1
Action renaming	$\mid$ **ren** r^* **in** b_1
Process instantiation	$\mid p$
Ordinary action	$a ::= g \mid h(\ldots)$
Interaction gate	$g ::= s \mid h$
Service primitive	$s ::= u^c$
Service-primitive type	$u ::= <identifier>$
Server component	$c ::= <identifier>$
Auxiliary gate	$h ::=$ **sync**|**tok**|**it**|**ct**|**ist**|**st**|**itt**|**tt**|**lt**
Exception	$x ::= <identifier>$
Renaming an action	$r ::= a \rightarrow a'$

Table 1: The adopted specification language

We introduce actions of three basic types.

- An SP u^c is an interaction of type u between component c and a service user.
- Gate **sync** serves for inter-component communication. For specific purposes, the gate will sometimes be called **tok**, **it** or **ct**.
- Gates **st**, **tt** and **lt** serve for synchronisation of processes within an individual component. For specific purposes, gates **st** and **tt** will sometimes be called **ist** and **itt**, respectively.

For the behaviour composition operators informally described below, a formal definition of semantics can be found in [3].

"$b_1; b_2$", "$b_1 [] b_2$" and "$b_1 |[G]| b_2$" respectively specify execution of b_1 and b_2 in sequence, as alternatives, or in parallel and synchronised on (actions on) the gates listed in G (and on δ). For our purposes it is crucial that $(b; \delta) \approx^c (\delta; b) \approx^c b$, where $\approx^c$ is the relation of observation congruence [11]. Iteration "**loop** b_1" denotes an infinite sequence of behaviours b_1. "$b_1 ||| b_2$" is a shorthand for independent parallelism "$b_1 |[]| b_2$". Operators "[]" and "|||" may also be employed in the prefix form, combining an arbitrary number of behaviours.

"$b_1 [> b_2$" denotes a process with behaviour b_1 repeatedly interrupted by behaviour b_2. b_1 is resumed upon a x in b_2, while b_2 is at that point restarted. While the b_1 part is active, the process might internally decide to terminate by enabling a δ in b_1. "$b_1 [> b_2$" denotes the special case with no resump-

tion, i.e. a LOTOS-like disabling (except that in LOTOS, the process in principle makes the decision to terminate by terminating b_1 in co-operation with its environment).

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

"**hide** G **in** b_1" hides the gates of b_1 listed in G, i.e. makes actions on the gates internal to b_1.

Explicit processes can be defined and instantiated without parameters.

No.	Syntax definition	
(1)	::=	$p := b$
(2)	b::=	s
(3)	b::=	$b_1; b_2$
(4)	b::=	$b_1 [] b_2$
(5)	b::=	$b_1 [> b_2$
(6)	b::=	$b_1 \|[s^*]\| b_2$

Table 2: The service specification sublanguage

Many of the constructs are only allowed in the derived protocol specifications. A service must be defined as an individual explicitly specified process (Table 2). The constructs legal within its behaviour specification are SPs, sequential composition, choice, pure disabling and parallel composition. In addition, we reasonably expect that a service is a non-blocking behaviour, i.e. that none of its parts b is ever blocked by non-co-operation of another service part b' running in parallel and synchronised with b on some common gates. All rows in Table 2 are numbered, so that the corresponding rows in Tables 3–8 and 10 can refer to them.

We want to implement services in a compositional way. Formally, we want our protocol derivation algorithm to establish for every service part b

Property 1 *Let b^c and $b^{c'}$ be the implementations of b at the server components c and c', respectively. Then* **hide sync in** $(b^c \|[\mathbf{sync}]\| b^{c'}) \approx b$.

In the derived protocol specifications, we shall extensively use the *constraint-oriented specification style* [14]. This is the style in which two or more parallel processes synchronise on the actions that they collectively control, and each process imposes its own constraints on the execution of the actions, so that they are enabled only when all the processes allow it. The constraint-oriented style is characterised by intensive inter-process communication, but as long as the constraining processes belong to the same server component, that is not problematic, for internal communication is supposed to be cheap.

3 Identification of Global Conflicts

In service specifications, the sources of conflicts are operators of choice ([]) and disabling ([>). In a $(b_1 [] b_2)$, the starting SPs of b_1 are in conflict with the starting SPs of b_2, and vice versa. In a $(b_1 [> b_2)$, the starting SPs of b_2 are in conflict with all the SPs of b_1. To identify the conflicts that are global, we must identify the executors of the conflicting SPs.

By computing service attributes ES and EA (Table 3), we identify the executors of individual SPs. $ES_c(b)$ is true for a service part b if c is the executor of some of the starting SPs of b. $EA_c(b)$ is true for a service part b if c is the executor of some SP among all the SPs of b.

No.	$ES_c(b)$	$EA_c(b)$
(2)	$\exists u : (s = u^c)$	$\exists u : (s = u^c)$
(3)	$ES_c(b_1)$	$EA_c(b_1) \vee EA_c(b_2)$
(4-6)	$ES_c(b_1) \vee ES_c(b_2)$	$EA_c(b_1) \vee EA_c(b_2)$

Table 3: Executors of service primitives

By computing service attributes CS and CA (Table 4), we identify for each individual SP the executors of the SPs conflicting with it. $CS_c(b)$ is true for a service part b if c is the executor of an SP outside b that is in conflict only with the starting SPs of b. $CA_c(b)$ is true for a service part b if c is the executor of an SP outside b that is in conflict with all the SPs of b.

No.	CS_c
(1)	$CS_c(b) = false$
(3)	$CS_c(b_1) = CS_c(b)$
	$CS_c(b_2) = false$
(4)	$CS_c(b_1) = (CS_c(b) \vee ES_c(b_2))$
	$CS_c(b_2) = (CS_c(b) \vee ES_c(b_1))$
(5)	$CS_c(b_1) = CS_c(b)$
	$CS_c(b_2) = (CS_c(b) \vee EA_c(b_1))$
(6)	$CS_c(b_1) = CS_c(b_2) = CS_c(b)$
No.	CA_c
(1)	$CA_c(b) = false$
(3-4,6)	$CA_c(b_1) = CA_c(b_2) = CA_c(b)$
(5)	$CA_c(b_1) = (CA_c(b) \vee ES_c(b_2))$
	$CA_c(b_2) = CA_c(b)$

Table 4: Identification of conflicts

By computing service attributes GA and GS (Table 5), we identify the SPs that are globally conflicting at least in some service runs at least some part of their life time, i.e. the GCSPs. Attribute $GA(b)$ is true if there are some GCSPs among all the SPs of b. Attribute $GS(b)$ is true if there are some GCSPs among the starting SPs of b. For a b of a form $(b_1 [] b_2)$ or $(b_1 [> b_2)$, implementation can be much simpler if $\neg GS(b)$.

No.	$GA(b)$	$GS(b)$
(2)	$\exists c : (EA_c(b) \wedge (CS_{c'}(b) \vee CA_{c'}(b)))$	
(3)	$GA(b_1) \vee GA(b_2)$	$GS(b_1)$
(4-6)	$GA(b_1) \vee GA(b_2)$	$GS(b_1) \vee GS(b_2)$

Table 5: Identification of global conflicts

4 Resolving Global Conflicts with a Token

If two SPs are in global conflict, the server must never enable them concurrently, for that would allow service users to invoke them both, in any order. The simplest solution is to introduce an internal server action localising the conflict.

Example 2 *For the conflict in Example 1 (illustrated in Figure 1(a)), an internal action* $\mathbf{i}^1$*, guarded by component* 1*, could be inserted before* d^2*, so that the conflict would be between* a^1 *and* $\mathbf{i}^1$*, i.e. local to component* 1 *(Figure 1(b)).*

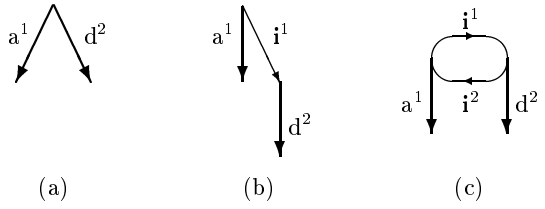

Figure 1: Localising a conflict

However, upon the dummy SP, one of the conflicting SPs is permanently disabled. In other words, the choice is made within the server, and the users deprived of the right to choose. The components should rather enable the SPs by turns [10]. That can be achieved by introducing a loop of internal server actions, for it is known that relation $\approx$ is insensitive to internal loops that are fair with their exits [10].

Example 3 *For the conflict in Figure 1(a), the transformation in Figure 1(c) allows the components* 1 *and* 2 *to repeatedly enable local choice between executing an SP and passing control to the peer component instead. If the components exchange the right to execute an SP sufficiently often, it will seem to the users that the SPs are continuously available for execution and subject to users' choice.*

In [10], it is explained how to use internal loops for services specified entirely in the *action-prefix form.* In that form, we describe a process as choice between a set of alternatives that are guarded by their initial actions. If a service is specified entirely in that form, i.e. in the so called *monolithic style* [14], all conflicts between its SPs are explicitly represented as conflicts between starting SPs of alternatives. For each point of choice in the service, there are some alternatives guarded by SPs belonging to a component c and others guarded by SPs belonging to c'. All we need is a loop in which each individual component repeatedly receives and passes the right to enable the alternatives that it is guarding.

Example 4 *A service behaviour* b *in the monolithic style is* $((\mathrm{a}^1;(\mathrm{d}^2[]\mathrm{g}^2))[](\mathrm{d}^2;\mathrm{a}^1)[](\mathrm{g}^2;\mathrm{a}^1))$*. In its initial step, either component* 1 *executes* a^1 *or component* 2 *executes* d^2 *or* g^2*.*

It is always possible to transform a service specification into the monolithic style [14], but information on which were the service parts running in parallel or composed by the disabling operator is lost. That is unacceptable, because (like [2]), we don't want to start protocol derivation by confluence analysis of a service behaviour (as for example in [4]) and we rely solely on the explicitly specified parallelism.

Example 5 *Suppose that the original form* b' *of the service behaviour* b *presented in Example 4 has been* $(\mathrm{a}^1|||(\mathrm{d}^2[]\mathrm{g}^2))$*. From that form we see that there are no two SPs either in global conflict or ordered, i.e. no interaction between components* 1 *and* 2 *is necessary.*

Although $b \approx b'$*,* b *suggests existence of a global conflict upon its first step (Example 4). By its sequential composition operators,* b *also suggests that* a^1 *must never be enabled concurrently to* d^2 *or* g^2*.*

Thus in the following, we generalise the ideas of [10] to structured service specifications. In [10], the right to enable GCSPs is some kind of a *token* repeatedly passed between server components. A component may enable a GCSP only when owning the token pertaining to the SP. The following example indicates that there should be a single token per server, not just to minimise the protocol traffic, but also to prevent deadlocks.

Example 6 *Consider a service*
$((\mathrm{a}^1[]\mathrm{d}^2)|[\mathrm{a}^1,\mathrm{d}^2]|(\mathrm{a}^1[]\mathrm{d}^2))$
of the form $(b_1|[\mathrm{a}^1,\mathrm{d}^2]|b_2)$*. If we introduce a token per each choice operator that gives rise to a global conflict, the distributed implementation of* b_1 *will by turns enable* a^1 *and* d^2*, and so will the distributed implementation of* b_2*. With badly chosen relative speeds of the two implementations, it might happen that they never enable* a^1 *or* d^2 *simultaneously. So the two parts would never succeed to synchronise and execute an SP. With a single token,* a^1 *or* d^2 *would always be enabled in both parts simultaneously, i.e. token management would not interfere with synchronisation of the service parts. Although the example is rather strange, it clearly shows that the solution with two (or more) independent tokens is not correct in a general case.*

To implement token management in a compositional way, we require that it is correct for every service part b, namely:

Property 2 *If* $GA(b)$*, both components know the initial location of the token.*

Property 3 *Whenever there is no GCSP pending, there is no token exchange (TE).*

Property 4 *Whenever there is a GCSP pending at a component* c*,* c *repeatedly receives the token until the SP is executed or disrupted.*

Property 5 *No GCSP is ever executed when its executor is not the current token owner.*

Property 6 *While a GCSP is pending at a component* c*, there is only a finite number of such token receptions at* c *that don't enable the SP.*

That is (in the presence of Property 4) the simplest way to ensure for any two service parts synchronised on a pending GCSP, that they repeatedly enable the SP simultaneously. In the absence of the property, the following situation would be possible:

Example 7 *Consider again the service from Example 6. Whenever component* 1 *receives the token, it may enable* a^1 *both in the implementation of* b_1 *and in the implementation of* b_2*, but with a permission to miss the opportunity infinitely often for* b_1 *and* b_2 *individually, it could e.g. enable* a^1 *for* b_1 *only on odd token receptions and for* b_2 *only on even receptions, thereby preventing execution of* a^1 *for ever.*

Property 7 *After a component c executes a GCSP disruptive for some SPs at c', c' doesn't receive the token before detecting that the GCSP has been executed, so that it can disrupt the SPs before the token would facilitate their execution.*

Let **tk** be the protocol message serving for token passing, i.e. the token always passed upon an interaction **sync**(**tk**).

5 Specifying a Protocol with Implicit Token Management

In this section, we start describing our protocol derivation algorithm by proposing how a c should behave with respect to SPs and the **sync** actions reporting their execution, for these are the types of actions known from earlier protocol derivation algorithms. For a service part b, the behaviour of a component c is specified by the mapping $\mathbf{H}_c(b)$ defined in Table 6. The mapping is based on the assumption that there is an implicit token management properly resolving global conflicts, i.e. for every service part b:

Assumption 1 *If in b, an SP s at a c is disruptive for an SP s' at c', then in the distributed implementation of b, c' never enables s' after c has executed s.*

In several places, mapping **H** uses service attribute $i(b)$, the identifier of a service part b. By a **sync**$(i(b))$, the components synchronise upon completion of b, thus $i(b)$ must be different from **tk**, that is also intended to be carried in **sync** actions. As communication on gate **sync** is non-buffered, exchange of **tk** messages never interferes with exchange of $i(b)$ messages.

If reporting of completion of a service part b is not specified, it is not necessary to define an $i(b)$. Otherwise the exact nature of $i(b)$ is irrelevant, except that it must be dynamically unique, i.e. when the components synchronise upon completion of b, they must both know that the particular protocol message $i(b)$ refers to the particular b. In Table 7, simple sufficient conditions for satisfying the requirement are expressed with the help of service attribute $I(b)$, that lists the identifiers of a b and of its sub-behaviours.

No.	b	$\mathbf{H'}_c(b)$
(2)	s	(if $EA_c(b)$ then s; endif **sync**$(i(b))$)
(3)	$b_1; b_2$	**Seq**$_c(\mathbf{H}, b_1, b_2)$
(4)	$b_1 [] b_2$	**Cho**$_c(\mathbf{H}, b_1, b_2)$
(5)	$b_1 [> b_2$	**Dis**$_c(\mathbf{H}, b, b_1, b_2)$
(6)	$b_1 \|[S]\| b_2$	**Par**$_c(\mathbf{H}, b_1, b_2, S)$

$\mathbf{H}_c(b) =$ if $(PC_c(b) \wedge PC_{c'}(b))$ then $\mathbf{H'}_c(b)$
else if $EA_c(b)$ then b else δ endif endif

Seq$_c(M, b_1, b_2) = (M_c(b_1);$
if $\exists c'' : (SC_{c''}(b_2) \wedge \neg DT_{c''}(b_1))$
then **sync**$(i(b_1))$; endif
$M_c(b_2))$

Cho$_c(M, b_1, b_2) =$ (**Alt**$_c(M, b_1)$[]**Alt**$_c(M, b_2)$)

Alt$_c(M, b) = (M_c(b)$ if $(\neg PC_c(b) \vee \neg PC_{c'}(b))$
then ; **sync**$(i(b))$ endif)

Dis$_c(M, b, b_1, b_2) =$(**hide lt in**
(**Dsb**$_c(M, b, b_1, b_2)$
$\|[\{u^c | (u^c \in SS(b_2))\} \cup \{\mathbf{sync}, \mathbf{lt}\})]\|$
Disr$_c(b, b_1, b_2)$)

Dsb$_c(M, b, b_1, b_2) = ((M_c(b_1); \mathbf{sync}(i(b)))$
$[> (M_c(b_2); \mathbf{lt}))$

Disr$_c(b, b_1, b_2) =$ (**Any**$(($
$\{u^c | (u^c \in SS(b_2))\} \cup$
$\{\mathbf{sync}(\mathbf{any} : (I(b_1) \cup I(b_2)))\}))$
$[> (\mathbf{sync}(i(b))[]\mathbf{lt}))$

Any$(A) = (|||(\{(\mathbf{loop}\ a) | a \in A\}))$

Par$_c(M, b_1, b_2, S) = (M_c(b_1) |[\{u^c | (u^c \in S)\}]| M_c(b_2))$

Table 6: Mapping **H**

No.	$I(b)$
(2)	$\{i(b)\}$
(3-6)	$I(b_1) \cup I(b_2) \cup \{i(b)\}$
(5)	$i(b) \notin (I(b_1) \cup I(b_2))$
(4-6)	$(I(b_1) \cap I(b_2)) = \emptyset$

Table 7: Identifiers of service parts

If a c must participate in implementation of a service part b with actions other than the final δ, we say that c is a participating component of b. $PC_c(b)$ is true iff c executes some SPs in b or b requires TE. Formally,

$PC_c(b) = (EA_c(b) \vee GA(b))$

5.1 Implementation of Local Service Behaviours

If $\neg PC_c(b)$, then c' implements b as it is, and c implements only a δ. c' locally executes b up to δ, and finally both components execute δ as their common action.

For a b with two participants, mapping **H** reduces to mapping $\mathbf{H'}$ (Table 6), discussed in the rest of the section.

5.2 Implementation of Individual Service Primitives

Mapping $\mathbf{H'}$ is applied to an SP s (2) iff it is a GCSP. In that case, execution of s by its pre-assigned executor is reported to the peer component by a **sync**$(i(b))$,

and finally the components synchronise on δ. The reporting of s is necessary to stop TE for the needs of b, to support Property 3. If s is a disruptive SP, its reporting is also necessary to support Property 7.

Implementing a compound service behaviour, we assume that under Assumption 1, its individual constituent parts are properly implemented. It is important that we have for every service part b Property 8, so that implementations of partially or totally concurrent service parts use different protocol-message sets (see Table 7).

Property 8 *If a* $\mathbf{sync}(m)$ *is an action in* $\mathbf{H}_c(b)$*, then* $(m \in I(b))$.

5.3 Implementation of Sequential Composition

For mapping of sequential composition (3), Table 8 introduces a new attribute $DT_c(b)$. It is true only if $\mathbf{H}_c(b)$ is known to always enable c to detect, before the components synchronise on the final δ, that execution of SPs in b has terminated. c receives the information by executing either the final SP in b or a synchronisation with c' known to occur after the final SP. All such cases have been deduced from Table 6.

No.	$DT_c(b)$
(2)	$PC_c(b)$
(3,5)	$DT_c(b_2)$
(4)	$(DT_c(b_1) \vee (PC_c(b) \wedge \neg PC_c(b_1))) \wedge$ $(DT_c(b_2) \vee (PC_c(b) \wedge \neg PC_c(b_2)))$
(6)	$DT_c(b_1) \wedge DT_c(b_2)$

Table 8: Components detecting termination

Executing a $(b_1; b_2)$, both components first execute their b_1 parts, i.e. b_1. If necessary, they subsequently synchronise upon the termination of b_1 by a $\mathbf{sync}(i(b_1))$. Finally they execute their b_2 parts, i.e. b_2.

According to Table 7, we don't require that the b_1 part and the b_2 part of the distributed implementation use different protocol messages. That is possible thanks to the following property of every service part b:

Property 9 *When a* c *executing* $\mathbf{H}_c(b)$ *enables* δ*,* c' *executing* $\mathbf{H}_{c'}(b)$ *has no* **sync** *action enabled.*

The property prevents a component c executing a $(b_1; b_2)$ from synchronising the **sync** actions in its b_1 part with the $\mathbf{sync}(i(b_1))$ (if) offered by c' and also with the **sync** actions in the b_2 part of c'.

The $\mathbf{sync}(i(b_1))$ is necessary if there is a component c that must detect termination of b_1 for proper start of b_2, but $\neg DT_c(b_1)$. A c is responsible for proper start of a service part b if there is a starting SP of b for which c is the executor or must participate in the TE for its needs. In that case we say that c is a starting component of b, formally $SC_c(b)$, where

$SC_c(b) = (ES_c(b) \vee GS(b))$

Example 8 *Let a service be* $((a^1; ((d^1; g^2)[> e^1)) ||| (a^2; ((d^2; g^1)[> e^2)))$

Its GCSPs are g^2, e^1, g^1 *and* e^2*, all because of the disabling operators.*

Suppose that we take the policy of [5] that for a $(b_1; b_2)$*, protocol synchronisation upon the sequential composition operator is necessary only if* b_1 *has an ending SP at a* c *and* b_2 *a starting SP at* c'*. Hence there would be such a synchronisation only in front of* g^2 *and* g^1*, but none after* a^1 *and* a^2*. Consequently, component* 2 *would be able to immediately start TE for the needs of* e^1*, as would component* 1 *for* e^2*. With both components ready to start TE immediately, it could erroneously start before a GCSP became pending, thereby violating Property 3.*

5.4 Implementation of Choice

Implementing a $(b_1 [] b_2)$ (4), both components execute their b_1 and b_2 parts as local alternatives. Mapping $\mathbf{H}$ is such that we have for every service part b

Property 10 *Every starting action of a* $\mathbf{H}_c(b)$ *is either a starting SP of* b *for which* c *is the executor, or a* **sync** *action guarded by a starting SP of* b *for which* c' *is the executor. It can also be a* δ*, but only if* $\neg PC_c(b)$.

Hence with b_1 and b_2 properly implemented, the starting actions of the implementation of a b specified as $(b_1 [] b_2)$ are exactly the starting actions of b. Suppose that execution of b starts by a c executing a starting SP s of an alternative b_n, i.e. of $\mathbf{H}_c(b_n)$. Upon the action, c abandons its $b_{n'}$ part. Assumption 1 prevents c' from starting $\mathbf{H}_{c'}(b_{n'})$ by an SP. Neither can c' start it $b_{n'}$ part by a **sync** action, because that would only be possible with co-operation of the b_n part of c, but such co-operation doesn't exist, for the two parts use different protocol messages. So c' remains ready to enter its b_n part.

It remains to ensure that c' detects that b_n has been selected for execution [5]. If $PC_{c'}(b_n)$, that is not a problem, as Property 10 implies that there will be an SP or a **sync** action executed within $\mathbf{H}_{c'}(b_n)$. If $\neg PC_{c'}(b_n)$, c must conclude execution of its b_n part by reporting to c'. In that way, the b_n part of c', that is originally equivalent to δ, becomes $\mathbf{sync}(i(b_n))$, i.e. provides c' with the information necessary for proper termination of b.

Example 9 *Table 9 gives an example of choice that is local, i.e. no TE is necessary, so that the derived protocol is already complete. Observe that there is an alternative in which originally only one of the components participates. The derived behaviours are already simplified modulo* $\approx$*, like in the rest of examples.*

5.5 Implementation of Disabling

Executing a b specified as $(b_1 [> b_2)$ (5), each component c basically executes $\mathbf{Dsb}_c(\mathbf{H}, b, b_1, b_2)$ consisting

$b = (\mathrm{a}^1[](\mathrm{d}^1; \mathrm{g}^2))$
$\mathbf{H}_1(b) \approx ((\mathrm{a}^1; \mathbf{sync}(1))[](\mathrm{d}^1; \mathbf{sync}(2)))$
$\mathbf{H}_2(b) \approx (\mathbf{sync}(1)[](\mathbf{sync}(2); \mathrm{g}^2))$

Table 9: Specifications for Example 9

of its b_1 and b_2 parts locally composed by $[>$. It is important to note that $PC_c(b)$ implies $PC_c(b_2)$.

The components usually start by executing their b_1 parts. As the SPs of b_1 are not disruptive for the SPs of b_2, no special measures are necessary for their execution. However, the δ of b_1 is disruptive for b_2. Therefore the components conclude execution of their b_1 parts by synchronising on $\mathbf{sync}(i(b))$, that indicates termination of b without activation of b_2.

At an individual c, $\mathbf{sync}(i(b))$ must prevent execution of all further non-δ actions in the **Dsb**. With $PC_c(b_2)$ and Property 10 for b_2, it suffices to prevent the starting SPs of b_2 for which c is the executor (the starting SPs of a b are listed in service attribute $SS(b)$ (Table 10)). For that purpose, we add in parallel to **Dsb** an additional constraint $\mathbf{Disr}_c(b, b_1, b_2)$. The two constraints are synchronised on a set of gates G consisting of the SPs that $\mathbf{sync}(i(b))$ should prevent and the gate **sync**. Why an auxiliary internal action **lt** is also in G, is explained below.

No.	$SS(b)$
(2)	$\{s\}$
(3)	$SS(b_1)$
(4-6)	$SS(b_1) \cup SS(b_2)$

Table 10: Starting SPs of service parts

The **Disr** starts by permitting free execution of all SPs and **sync** actions within $\mathbf{H}_c(b_1)$ and $\mathbf{H}_c(b_2)$, including disruption of b_1 (hence the name). Upon a $\mathbf{sync}(i(b))$ or **lt** issued by **Dsb**, **Disr** cancels the permission for actions on the gates in G, and is ready to terminate.

Hence **Disr** allows c to execute its b_1 part up to $\mathbf{sync}(i(b))$. After the action, **Disr** doesn't allow **Dsb** to start its b_2 part, so that the only option for **Dsb** is a δ in co-operation with **Disr**. The same holds for c' after the $\mathbf{sync}(i(b))$.

Alternatively, a starting SP s of b_2 might be executed, possibly after some SPs of b_1. Let c be its executor. s makes c immediately abandon its b_1 part. As for c', we know that it can execute no further actions in its b_1 part, for the SPs in the part are disabled according to Assumption 1, while for the **sync** actions, the necessary co-operation of the b_1 part of c no longer exists and the b_2 part of c uses different protocol messages. Hence the only option for c' is to proceed with its b_2 part.

With $PC_{c'}(b_2)$ and Property 10 for b_2 we know that c' will actually execute a non-δ action in its b_2 part. Hence the two components enter their b_2 parts in a co-ordinated manner and are thus able to complete execution of b_2. The only task remaining for a c before synchronising with c' on the δ of b upon completion of b_2 is to terminate **Disr**. That is achieved by an action **lt** indicating local termination of b at c. By hiding, the action is made internal to $\mathbf{H}_c(b)$. The scheme is an optimisation over [2], where execution of b concludes by a $\mathbf{sync}(i(b))$ even if b terminates by terminating b_2.

Example 10 *Table 11 gives an example of disabling that is local, i.e. no TE is necessary. For comparison, a protocol for it is derived first by our mapping* **H**, *and then another one in the manner of [2].*

$b = (\mathrm{a}^1[>(\mathrm{d}^1; \mathrm{a}^1; \mathrm{g}^2))$
$\mathbf{H}_1(b) \approx (\mathbf{hide\ lt\ in}$
$\quad (((\mathrm{a}^1; \mathbf{sync}(1))[>(\mathrm{d}^1; \mathrm{a}^1; \mathbf{sync}(2); \mathbf{lt}))$
$\quad\ \|[\mathrm{d}^1, \mathbf{sync}, \mathbf{lt}]\|$
$\quad\ (\mathbf{Any}(\{\mathrm{d}^1, \mathbf{sync}(2)\})[>(\mathbf{sync}(1)[]\mathbf{lt}))))$
$\mathbf{H}_2(b) \approx (\mathbf{sync}(1)[](\mathbf{sync}(2); \mathrm{g}^2))$
$\mathbf{H}_1(b) \approx ((\mathrm{a}^1[>(\mathrm{d}^1; \mathrm{a}^1; \mathbf{sync}(2))); \mathbf{sync}(1))$
$\mathbf{H}_2(b) \approx (\mathbf{sync}(1)[](\mathbf{sync}(2); \mathrm{g}^2; \mathbf{sync}(1)))$

Table 11: Specifications for Example 10

5.6 Implementation of Parallel Composition

Implementing a $(b_1|[S]|b_2)$ (6), it suffices that each component c executes $\mathbf{H}_c(b_1)$ and $\mathbf{H}_c(b_2)$ in parallel and synchronised on the SPs in S for which c is the executor [2]. That is because the protocol channel is synchronous and because the implementations of b_1 and b_2 use different protocol messages. Consequently, one may pretend that the two implementations use two different protocol channels, and are hence independent. As for their synchronisation on the SPs in S, it is a local matter.

6 Implementation of Token Management

In this section, we integrate mapping **H** with actions serving for token management, to obtain a complete protocol derivation mapping **T**. If a service part b requires no TE, its mapping **T** reduces to mapping **H** (i.e. execution of b doesn't depend on the token ownership), otherwise we shall call it mapping $\mathbf{T}'$. Formally,

$\mathbf{T}_c(b) =$ if $GA(b)$ then $\mathbf{T}'_c(b)$ else $\mathbf{H}_c(b)$ endif

6.1 The Virtual Token Manager

The behaviour of a server component c virtually consists of two concurrent processes. The first one is the *virtual service executor* (VSE), that executes the actions specified by mapping **H**. The other one is the *virtual token manager* (VTM), that manages the token for the needs of VSE. Implementing a service in

a compositional way, VTM consists of a set of coordinated processes managing the token for individual VSE parts – the VTMs belonging to the parts.

As VTM should perform TE only when there is a VSE part needing it (Property 3), we introduce auxiliary actions **st** and **tt**. By a **st**, a VSE part requires starting of TE for its needs. By a **tt**, a VSE part requires termination of TE for its needs. Between issuing a **st** and a subsequent **tt**, a VSE part is said to be *connected to the token*. It is quite possible that a VSE part connects to the token only occasionally. After a VSE part has disconnected from the token, it may issue another **tt** with no harm.

In Section 4 it has been decided that the action type serving for TE is **sync**(**tk**). Any **sync**(**tk**) should be executed as a common action of the VTMs of all the VSE parts currently connected to the token, because all VSE parts use the same token (see Example 6). That scheme also supports implementation of Property 7. For suppose that a GCSP has been executed within a VSE part, but not yet reported to c'. By remaining connected to the token and not allowing its VTM to enable another **sync**(**tk**) until the components have synchronised on a report on the GCSP, the VSE part can prevent the VTMs of the other VSE parts from premature passing of the token.

To synchronise on a **sync**(**tk**), concurrent parts of c have to synchronise on gate **sync**. That might be inconvenient, because their other actions on the gate (those introduced by mapping **H**) are typically intended for independent execution. For that reason we internally to c pretend that **sync**(**tk**) actions belong to a special gate **tok**. The token is passed from a c to c' upon a **tok**(c').

The basic task of VTM is to keep record of the token ownership. For that purpose, we introduce into the behaviour of a c that might require TE a constraint **Tok** securing that c executes actions **tok**(c) and **tok**(c') by turns (the *token-ownership constraint*). Integration of the constraint into the specification of c is specified in Table 12. In the following, the implementation of a service part b at a c denotes the behaviour of $\mathbf{T}_c(b)$ under the additional constraint.

$\mathbf{T}_c(p := b) =$
 if $GA(b)$ then
  $p_c :=$ **ren** $\forall c \in \mathcal{C} : (\mathbf{tok}(c) \rightarrow \mathbf{sync}(\mathbf{tk}))$ **in**
   hide st, tt, lt in
    $((\mathbf{T}_c(b); \mathbf{lt})$
     $|[\mathbf{tok}, \mathbf{lt}]|$
     $(\mathbf{Tok}($if SO_c then c' else c endif$)[>\mathbf{lt}))$
  TokC:=... see Table 17
 else $p_c := \mathbf{T}_c(b)$ endif
Tok$(c) = (\mathbf{loop}\ (\mathbf{tok}(c); \mathbf{tok}(c')))$

Table 12: Mapping **T** for a service specification

A service is specified as a process p with a behaviour b. If b doesn't require TE, specification of a component c reduces to specification of a process p_c with behaviour $\mathbf{T}_c(b)$. If $GA(b)$, the behaviour of p_c is more complicated, and it might be necessary to also specify an auxiliary process **TokC** (described in Section 6.4).

If $GA(b)$, $\mathbf{T}_c(b)$ executes **tok** actions and must thus run in parallel to the token-ownership constraint. If c is the token owner upon the start, i.e. SO_c, the **Tok** is a **Tok**(c'), i.e. starts by passing the token to c', and a **Tok**(c) otherwise. After $\mathbf{T}_c(b)$ terminates, a **lt** indicating local termination of b is executed to disable **Tok** and allow p_c to terminate. Hiding indicates that **lt** is an internal action of p_c. The **st** and **tt** actions delimiting the intervals of TE for the needs of b must also be hidden. For proper synchronisation with c', **tok** actions are externally renamed into **sync**(**tk**) actions.

In the rest of the section, we discuss token management for service parts that require TE, i.e. mapping $\mathbf{T}'$.

6.2 Token Management for Individual Service Primitives

Mapping $\mathbf{T}'$ for a b specified as a GCSP s (2) is given in Table 13. Each c starts $\mathbf{T}'_c(b)$ by connecting to the token by a **st** and terminates it by disconnecting from the token by a **tt**. In between, server components execute TE for the needs of s. The executor of s is also responsible for its execution and reporting.

$\mathbf{T}'_c(b)$ where $(b = s) =$
 (**st**;
  if $EA_c(b)$ then
   (**tok**(c)[](**tok**(c'); **tok**(c)));
   (($s[x>$(**tok**(c'); **tok**(c); $x)$)
    $|[s, \mathbf{tok}]|$
    ((**loop tok**(**any** : $\mathcal{C}$))$[>(s; \mathbf{sync}(i(b)))))$
  else ((**Tok**(c)[]**Tok**(c'))$[>\mathbf{sync}(i(b)))$ endif
 ;**tt**)

Table 13: Mapping $\mathbf{T}'$ for a service primitive

The token location upon the point when a $\mathbf{T}'_c(s)$ connects to the token can never be determined in advance. For the fact that s is a GCSP implies that it is potentially concurrent to a GCSP s', i.e. $\mathbf{T}'_c(s')$ connects to the token concurrently to $\mathbf{T}'_c(s)$. So it might happen that $\mathbf{T}'_c(s')$ connects to the token and executes an unpredictable number of **tok** actions before $\mathbf{T}'_c(s)$ manages to connect. Hence a $\mathbf{T}'_c(s)$ must be ready to start TE both by a **tok**(c) or a **tok**(c'). The decision is made in co-operation with the token-ownership constraint.

If c is the executor of s, the central part of $\mathbf{T}'_c(b)$ is $(s[x>(\mathbf{tok}(c'); \mathbf{tok}(c); x))$, i.e. s is disabled upon every token transmission **tok**(c') and re-enabled upon every token reception **tok**(c). The exact name of the resumption exception x is irrelevant.

Upon execution of s, c immediately stops TE and issues a report $\mathbf{sync}(i(b))$, to terminate $\mathbf{T'}_{c'}(b)$ and, if s is disruptive, to implement Property 7. That is specified by constraint

$((\mathbf{loop\ tok(any} : \mathcal{C}))[> (s; \mathbf{sync}(i(b))))$

running in parallel to the central part of $\mathbf{T'}_c(b)$. After the components synchronise on $\mathbf{sync}(i(b))$ and execute $\mathbf{tt}$, TE can be resumed for the needs of other GCPSs.

Before $\mathbf{T'}_c(b)$ first enables s, i.e. its central part, it must check the current token location, by trying both a $\mathbf{tok}(c)$ and a $\mathbf{tok}(c')$. If $\mathbf{T'}_c(b)$ is successful on $\mathbf{tok}(c')$, it must before its central part execute a $\mathbf{tok}(c)$, to regain the token.

If a c is not the executor of s, it simply co-operates in all $\mathbf{tok}$ actions initiated by c'. The loop is disrupted upon the $\mathbf{sync}(i(b))$ by which c' reports execution of s.

Example 11 *Let a service behaviour b be specified as* a^1, *where* $GA(b)$, $(\mathcal{C} = \{1,2\})$ *and* $(i(b) = 3)$. *Mappings* $\mathbf{H}$ *and* $\mathbf{T}$ *for b are given in Table 14. Observe how the token is passed between the components, how it repeatedly enables execution of the SP, and how TE is suspended upon the SP. Remember that every* $\mathbf{tok}(c)$ *is actually a* $\mathbf{sync}(\mathbf{tk})$.

$b = \mathrm{a}^1$

$\mathbf{H}_1(b) = (\mathrm{a}^1; \mathbf{sync}(3))$

$\mathbf{H}_2(b) = \mathbf{sync}(3)$

$\mathbf{T}_1(b) = (\mathbf{st}; (\mathbf{tok}(1)[](\mathbf{tok}(2); \mathbf{tok}(1)));$
$((\mathrm{a}^1[x>(\mathbf{tok}(2); \mathbf{tok}(1); x))|[\mathrm{a}^1, \mathbf{tok}]|$
$((\mathbf{loop\ tok(any} : \{1,2\}))[>(\mathrm{a}^1; \mathbf{sync}(3))))$
$; \mathbf{tt})$

$\mathbf{T}_2(b) = (\mathbf{st};$
$(((\mathbf{loop}\ (\mathbf{tok}(1); \mathbf{tok}(2)))[]$
$(\mathbf{loop}\ (\mathbf{tok}(2); \mathbf{tok}(1))))$
$[>\mathbf{sync}(3))$
$; \mathbf{tt})$

Table 14: Specifications for Example 11

Before proposing in the rest of the section how to implement TE for compound behaviours, we make an important observation that the implementation of each individual SP implements TE for its own needs. Hence implementing token management for a compound service behaviour b, it suffices to locally synchronise the $\mathbf{tok}$ actions belonging to the implementations of the SPs currently connected to the token. The restructuring of $\mathbf{H}_c(b)$ into $\mathbf{T'}_c(b)$ is then correct if

1) the behaviour of c with respect to SPs and $\mathbf{sync}$ actions is preserved, and

2) the disruptive effects of the remote GCSPs on the SPs at c are properly implemented (for mapping $\mathbf{H}$ properly implements b only under Assumption 1).

Again we will proceed in a compositional way, assuming correct implementation of service parts and specifying implementation of composition operators.

6.3 Token Management for Sequential Composition

Implementing a b specified as $(b_1; b_2)$ (3), a component c combines its b_1 part and its b_2 part in the same manner (function $\mathbf{Seq}$ in Table 15) as in the mapping $\mathbf{H}$ (Table 6(3)), hence the behaviour of c with respect to SPs and $\mathbf{sync}$ actions remains that of $\mathbf{H}_c(b)$. There is no concurrency between b_1 and b_2, hence no synchronisation of $\mathbf{T}_c(b_1)$ and $\mathbf{T}_c(b_2)$ on $\mathbf{tok}$ actions is necessary. The sequential composition operator introduces no additional SP disruptions to implement.

$\mathbf{T'}_c(b)$ where $(b = (b_1; b_2)) = \mathbf{Seq}_c(\mathbf{T}, b_1, b_2)$

Table 15: Mapping $\mathbf{T'}$ for sequential composition

6.4 Token Management for Parallel Composition

Implementing a b specified as $(b_1|[S]|b_2)$ (6), a component c basically combines its b_1 and its b_2 part in the same manner (function $\mathbf{Par}$ in Table 16) as in the mapping $\mathbf{H}$ (Table 6(6)), hence the behaviour of c with respect to SPs and $\mathbf{sync}$ actions remains that of $\mathbf{H}_c(b)$. The parallel composition operator introduces no additional SP disruptions to implement. However, b_1 and b_2 are concurrent. Hence if $\mathbf{T}_c(b_1)$ and $\mathbf{T}_c(b_2)$ both execute TE, formally $(GA(b_1) \wedge GA(b_2))$, synchronisation of their $\mathbf{tok}$ actions is necessary.

$\mathbf{T'}_c(b)$ where $(b = (b_1|[S]|b_2)) =$
if $(GA(b_1) \wedge GA(b_2))$ then
$\mathbf{ExtTok}((\mathbf{T\_Par}_c(b_1, b_2, S)|[\mathbf{it}, \mathbf{ct}, \mathbf{ist}, \mathbf{itt}, \mathbf{lt}]|\mathbf{TokC}))$
else $\mathbf{Par}_c(\mathbf{T}, b_1, b_2, S)$ endif

$\mathbf{T\_Par}_c(b_1, b_2, S) = (\mathbf{IntTok}((\mathbf{T}_c(b_1); \mathbf{lt}), 1)$
$||[(\{u^c|(u^c \in S)\} \cup \{\mathbf{ct}, \mathbf{lt}\})]||$
$\mathbf{IntTok}((\mathbf{T}_c(b_2); \mathbf{lt}), 2))$

$\mathbf{IntTok}(b, n) = (\mathbf{ren}\ \forall c \in \mathcal{C} : (\mathbf{tok}(c) \rightarrow \mathbf{it}(c)),$
$\forall c \in \mathcal{C} : (\mathbf{tok}(c) \rightarrow \mathbf{ct}(c)),$
$\mathbf{st} \rightarrow \mathbf{ist}(n),\ \mathbf{tt} \rightarrow \mathbf{itt}(n)$
$\mathbf{in}\ b)$

$\mathbf{ExtTok}(b) = (\mathbf{ren}\ \forall c \in \mathcal{C} : (\mathbf{it}(c) \rightarrow \mathbf{tok}(c)),$
$\forall c \in \mathcal{C} : (\mathbf{ct}(c) \rightarrow \mathbf{tok}(c))$
$\mathbf{in\ hide\ ist,itt,lt\ in}\ b)$

Table 16: Mapping $\mathbf{T'}$ for parallel composition

With $\mathbf{T}_c(b_1)$ and $\mathbf{T}_c(b_2)$ pursuing TE concurrently, a $\mathbf{T}_c(b_n)$ must be ready to execute a $\mathbf{tok}(c'')$ either as an individual action, i.e. as an $\mathbf{it}(c'')$, or as an action common with the concurrent part $\mathbf{T}_c(b_{n'})$, i.e. as a $\mathbf{ct}(c'')$. The renaming of the token management actions internally to $\mathbf{T}_c(b)$ is specified by function $\mathbf{IntTok}$ (Table 16). Actions $\mathbf{st}$ and $\mathbf{tt}$ of each $\mathbf{T}_c(b_n)$ are also given internal names $\mathbf{ist}(n)$ and $\mathbf{itt}(n)$, respectively, by the function $\mathbf{IntTok}$.

For proper execution of $\mathbf{ct}$ actions, we synchronise the two concurrent parts on gate $\mathbf{ct}$, as specified by

function **T_Par** in Table 16, a modified version of function **Par**. The function also specifies that the two parts terminate by synchronising on a **lt** indicating local termination of b.

The **lt** issued by $\mathbf{T\_Par}_c(b_1, b_2, S)$ serves for termination of the token exchange co-ordinator **TokC** running in parallel as an additional constraint. The auxiliary process is specified in Table 17.

$$
\begin{array}{l}
\mathbf{TokC} := ((\mathbf{ist}(1); \mathbf{st}; \mathbf{TokC1}) [] (\mathbf{ist}(2); \mathbf{st}; \mathbf{TokC2}) [] \\
\qquad (\mathbf{itt}(1); \mathbf{TokC}) [] (\mathbf{itt}(2); \mathbf{TokC}) [] \mathbf{lt}) \\
\mathbf{TokC1} := ((\mathbf{loop\ it}(\mathbf{any} : \mathcal{C})) \\
\qquad [> ((\mathbf{ist}(2); \mathbf{TokC12}) [] \\
\qquad\quad (\mathbf{itt}(1); \mathbf{tt}; \mathbf{TokC}) [] (\mathbf{itt}(2); \mathbf{TokC1}))) \\
\mathbf{TokC2} := ((\mathbf{loop\ it}(\mathbf{any} : \mathcal{C})) \\
\qquad [> ((\mathbf{ist}(1); \mathbf{TokC12}) [] \\
\qquad\quad (\mathbf{itt}(1); \mathbf{TokC2}) [] (\mathbf{itt}(2); \mathbf{tt}; \mathbf{TokC}))) \\
\mathbf{TokC12} := ((\mathbf{loop\ ct}(\mathbf{any} : \mathcal{C})) \\
\qquad [> ((\mathbf{itt}(1); \mathbf{TokC2}) [] (\mathbf{itt}(2); \mathbf{TokC1})))
\end{array}
$$

Table 17: Token exchange co-ordinator

The co-ordinator switches between four different modes. Its basic mode **TokC** is active when neither the b_1 part nor the b_2 part is connected to the token. If just a b_n part is connected, the mode of **TokC** is **TokC**n. If both parts are connected, the mode is **TokC12**. The initial mode is **TokC**. When a b_n part indicates its connection to (or disconnection from) the token by an $\mathbf{ist}(n)$ ($\mathbf{itt}(n)$, respectively), the co-ordinator switches to the appropriate mode. In a mode **TokC**n, the co-ordinator allows unlimited execution of **it** actions. In the mode **TokC12**, it allows unlimited execution of **ct** actions.

When switching from the mode **TokC** to a mode **TokC**n, the co-ordinator issues a **st** indicating that $\mathbf{T}_c(b)$ is connecting to the token. When switching from a mode **TokC**n to **TokC**, it issues a **tt** indicating that $\mathbf{T}_c(b)$ is disconnecting from the token. The co-ordinator can terminate only in its basic mode, for $\mathbf{T}_c(b)$ is not allowed to terminate while connected to the token.

As **ist**, **itt** and **lt** actions serve only for synchronisation within $\mathbf{T}_c(b)$, they must be hidden. On the other hand, **it** and **ct** actions must be externally renamed back into **tok** actions. The proper external appearance of the token management actions is established by function **ExtTok** (Table 16).

Example 12 *In Table 18, mappings* **H** *and* **T** *are given for a service part b that consists of concurrent parts b_1 and b_2 synchronised on SP* d^2*. The GCSPs are the SPs executed by component* 2*, e.g. because there is a service part b' disabling for b and with component* 1 *executing all its starting SPs.*

For each component, only its b_1 part initially connects to the token. After execution and reporting of g^1*, the b_1 and the b_2 part are both connected. After execution and reporting of* d^2*, there is no TE at all. Observe how* **TokC** *(specified in Table 17) switches between individual and common execution of* **tok** *actions. Observe also how protocol message types* 1 *and* 2 *are re-used within the b_1 and the b_2 part, respectively.*

$$
\begin{array}{l}
b = ((\mathrm{a}^2; \mathrm{d}^2) |[\mathrm{d}^2]| (\mathrm{g}^1; \mathrm{d}^2; \mathrm{e}^1)) \\
\mathbf{H}_1(b) \approx ((\mathbf{sync}(1); \mathbf{sync}(1)) ||| \\
\qquad (\mathrm{g}^1; \mathbf{sync}(2); \mathbf{sync}(2); \mathrm{e}^1)) \\
\mathbf{H}_2(b) \approx ((\mathrm{a}^2; \mathbf{sync}(1); \mathrm{d}^2; \mathbf{sync}(1)) |[\mathrm{d}^2]| \\
\qquad (\mathbf{sync}(2); \mathrm{d}^2; \mathbf{sync}(2))) \\
\mathbf{T}_1(b) \approx \\
(\mathbf{ren\ it}(1) \to \mathbf{tok}(1), \mathbf{it}(2) \to \mathbf{tok}(2), \\
\quad \mathbf{ct}(1) \to \mathbf{tok}(1), \mathbf{ct}(2) \to \mathbf{tok}(2)\ \mathbf{in} \\
\ \mathbf{hide\ ist, itt, lt\ in} \\
\ (((\mathbf{ren\ tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2), \\
\qquad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in} \\
\quad (\mathbf{ist}(1); \\
\quad\ (((\mathbf{loop}\ (\mathbf{tok}(1); \mathbf{tok}(2))) [] (\mathbf{loop}\ (\mathbf{tok}(2); \mathbf{tok}(1)))) \\
\quad\ [> \mathbf{sync}(1)); \\
\quad\ \mathbf{itt}(1); \mathbf{ist}(1); \\
\quad\ (((\mathbf{loop}\ (\mathbf{tok}(1); \mathbf{tok}(2))) [] (\mathbf{loop}\ (\mathbf{tok}(2); \mathbf{tok}(1)))) \\
\quad\ [> \mathbf{sync}(1)); \\
\quad\ \mathbf{itt}(1))) \\
\quad |[\mathbf{ct}, \mathbf{lt}]| \\
\quad (\mathbf{ren\ tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2), \\
\qquad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in} \\
\quad\ (\mathrm{g}^1; \mathbf{sync}(2); \mathbf{ist}(2); \\
\quad\ (((\mathbf{loop}\ (\mathbf{tok}(1); \mathbf{tok}(2))) [] (\mathbf{loop}\ (\mathbf{tok}(2); \mathbf{tok}(1)))) \\
\quad\ [> \mathbf{sync}(2)); \\
\quad\ \mathbf{itt}(2); \mathrm{e}^1))) \\
\ |[\mathbf{it}, \mathbf{ct}, \mathbf{ist}, \mathbf{itt}, \mathbf{lt}]| \mathbf{TokC})) \\
\mathbf{T}_2(b) \approx \\
(\mathbf{ren\ it}(1) \to \mathbf{tok}(1), \mathbf{it}(2) \to \mathbf{tok}(2), \\
\quad \mathbf{ct}(1) \to \mathbf{tok}(1), \mathbf{ct}(2) \to \mathbf{tok}(2)\ \mathbf{in} \\
\ \mathbf{hide\ ist, itt, lt\ in} \\
\ (((\mathbf{ren\ tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2), \\
\qquad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in} \\
\quad (\mathbf{ist}(1); (\mathbf{tok}(2) [] (\mathbf{tok}(1); \mathbf{tok}(2))); \\
\quad\ ((\mathrm{a}^2[x > (\mathbf{tok}(1); \mathbf{tok}(2); x)) |[\mathrm{a}^2, \mathbf{tok}]| \\
\quad\ ((\mathbf{loop\ tok}(\mathbf{any} : \{1, 2\}))[> (\mathrm{a}^2; \mathbf{sync}(1)))); \\
\quad\ \mathbf{itt}(1); \mathbf{ist}(1); (\mathbf{tok}(2) [] (\mathbf{tok}(1); \mathbf{tok}(2))); \\
\quad\ ((\mathrm{d}^2[x > (\mathbf{tok}(1); \mathbf{tok}(2); x)) |[\mathrm{d}^2, \mathbf{tok}]| \\
\quad\ ((\mathbf{loop\ tok}(\mathbf{any} : \{1, 2\}))[> (\mathrm{d}^2; \mathbf{sync}(1)))); \\
\quad\ \mathbf{itt}(1))) \\
\quad |[\mathrm{d}^2, \mathbf{ct}, \mathbf{lt}]| \\
\quad (\mathbf{ren\ tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2), \\
\qquad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in} \\
\quad\ (\mathbf{sync}(2); \\
\quad\ \mathbf{ist}(2); (\mathbf{tok}(2) [] (\mathbf{tok}(1); \mathbf{tok}(2))); \\
\quad\ ((\mathrm{d}^2[x > (\mathbf{tok}(1); \mathbf{tok}(2); x)) |[\mathrm{d}^2, \mathbf{tok}]| \\
\quad\ ((\mathbf{loop\ tok}(\mathbf{any} : \{1, 2\}))[> (\mathrm{d}^2; \mathbf{sync}(2)))); \\
\quad\ \mathbf{itt}(2)))) \\
\ |[\mathbf{it}, \mathbf{ct}, \mathbf{ist}, \mathbf{itt}, \mathbf{lt}]| \mathbf{TokC}))
\end{array}
$$

Table 18: Specifications for Example 12

6.5 Token Management for Choice

In a b specified as $(b_1 [] b_2)$ (4), the choice operator introduces concurrency and conflicts between the starting SPs of b_1 and those of b_2.

If $\neg GS(b)$, all the starting SPs have the same executor and don't pursue TE. Hence for every alternative

b_n and component c, Property 10, originally addressing mapping **H**, also applies to $\mathbf{T}_c(b_n)$. Consequently, function **Cho** used in the mapping **H** (Table 6(4)) is also suitable for combining $\mathbf{T}_c(b_1)$ and $\mathbf{T}_c(b_2)$ (Table 19).

$\mathbf{T'}_c(b)$ where $(b = (b_1 [] b_2)) =$
 if $GS(b)$ then
  $\mathbf{ExtTok}((\mathbf{T\_Cho}_c(b_1, b_2) | [\mathbf{it}, \mathbf{ct}, \mathbf{ist}, \mathbf{itt}, \mathbf{lt}] | \mathbf{TokC}))$
 else $\mathbf{Cho}_c(\mathbf{T}, b_1, b_2)$ endif
$\mathbf{T\_Cho}_c(b_1, b_2) =$
 $\mathbf{ExtSP}_c((\mathbf{IntTok}(\mathbf{T\_Alt}_c(b_1, b_2, 1, 2), 1)$
  $|[(\{u1^c | (u^c \in SS(b_1))\} \cup \{u2^c | (u^c \in SS(b_2))\} \cup$
  $\{\mathbf{sync}, \mathbf{ct}, \mathbf{lt}\})]|$
  $\mathbf{IntTok}(\mathbf{T\_Alt}_c(b_2, b_1, 2, 1), 2)),$
  $b_1, b_2)$
$\mathbf{IntSP}_c(b, b_n, n) = (\mathbf{ren}\ \forall u^c \in SA(b_n) : (u^c \rightarrow un^c)\ \mathbf{in}\ b)$
$\mathbf{ExtSP}_c(b, b_1, b_2) = (\mathbf{ren}\ \forall u^c \in SA(b_1) : (u1^c \rightarrow u^c),$
  $\forall u^c \in SA(b_2) : (u2^c \rightarrow u^c)$
  $\mathbf{in}\ b)$
$\mathbf{T\_Alt}_c(b, b', n, n') = ((\mathbf{IntSP}_c(\mathbf{T}_c(b), b, n); \mathbf{lt})$
  $[> \mathbf{Det}_c(b', n'))$
$\mathbf{Det}_c(b, n) = (([](\mathbf{Act}_c(b, n))); \mathbf{tt};$
  $(\mathbf{Any}(\mathbf{Act}_c(b, n))[> \mathbf{lt}))$
$\mathbf{Act}_c(b, n) = (\{un^c | (u^c \in SS(b))\} \cup \{\mathbf{sync}(\mathbf{any} : I(b))\})$

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

If $GS(b)$, it is convenient that both components originally participate in the implementations of both alternatives, but we have problems with the TE actions belonging to the starting SPs of b.

First, an implementation $\mathbf{T}_c(s)$ of a starting GCSP s in an alternative b_n has a token management action as its initial action. Hence the action is also among the initial actions of $\mathbf{T}_c(b_n)$, and would be able to resolve the choice, if $\mathbf{T}_c(b_1)$ and $\mathbf{T}_c(b_2)$ were combined as specified by function **Cho**, while mapping **H** indicates that resolution of the choice is only allowed upon an SP or a **sync** action.

Second, $GS(b)$ implies that there are starting GCSPs in both b_1 and b_2. The GCSPs pursue TE concurrently, so we must synchronise the **tok** actions of $\mathbf{T}_c(b_1)$ with those of $\mathbf{T}_c(b_2)$. To implement the synchronisation in the manner proposed in the previous section, we specify $\mathbf{T}_c(b_1)$ and $\mathbf{T}_c(b_2)$ in parallel composition (function **T_Cho** in Table 19) and concurrent to a TE co-ordinator **TokC**.

$\mathbf{T\_Cho}_c(b_1, b_2)$ in Table 19 is an analogue of $\mathbf{T\_Par}_c(b_1, b_2, S)$ in Table 16. As suggested in Section 6.4, it executes TE actions with the names that are valid internally to the implementation of b. The names are obtained by applying function **IntTok** to the parallel constituents of the **T_Cho**. The proper external appearance of the TE actions is established by function **ExtTok**.

In **T_Cho**, the part analogous to a $(\mathbf{T}_c(b_n); \mathbf{lt})$ in the **T_Par** is $\mathbf{T\_Alt}_c(b_n, b_{n'}, n, n')$, responsible for the alternative b_n. As in **T_Par**, the b_1 part and the b_2 part are synchronised on actions **ct** (their common **tok** actions) and **lt** (the local termination). Besides, the two parts must be synchronised on their starting SPs and on their starting **sync** actions (i.e. on gate **sync**), to help make **T_Cho** an analogue of $\mathbf{Cho}_c(\mathbf{T}, b_1, b_2)$, like synchronisation of its two parts on the local SPs in S makes **T_Par** an analogue of $\mathbf{Par}(b_1, b_2, S)$. For don't forget that the two parts of **T_Cho** are originally intended for execution as alternatives as far as SPs and **sync** actions are concerned. Thus upon the start of a $\mathbf{H}_c(b_n)$ in the incorporated $\mathbf{H}_c(b)$, the $b_{n'}$ part must be able to detect the first of such actions, to properly abort itself as the abandoned alternative.

Obviously we must distinguish between the starting SPs of b_1 and those of b_2, even if they have identical names. That is achieved by function **IntSP**, that internally to **T_Cho** extends the name of each local SP in a b_n with n. The proper external appearance of the SPs is established by function **ExtSP**.

In a $\mathbf{T\_Alt}_c(b_n, b_{n'}, n, n')$, the central part is $(\mathbf{T}_c(b_n); \mathbf{lt})$, that is the same as the b_n part in **T_Par**. In addition, there is a $\mathbf{Det}_c(b_{n'}, n')$, a process detecting the start of SPs and **sync** actions in $(\mathbf{T}_c(b_{n'}); \mathbf{lt})$, the central part of the $b_{n'}$ part. The actions disruptive for $\mathbf{T}_c(b_n)$ are specified in $\mathbf{Act}_c(b_{n'}, n')$.

The **Det** is activated upon the first such action. It immediately disables $\mathbf{T}_c(b_n)$ and by a **tt** informs **TokC** that the b_n part no longer needs TE, for $\mathbf{T}_c(b_n)$ might have been disabled while connected to the token. Afterwards, the **Det** supports free execution of the actions listed in $\mathbf{Act}_c(b_{n'}, n')$, until local termination of $\mathbf{T}_c(b)$ is indicated by a **lt** issued by the $b_{n'}$ part. For note that the b_1 and the b_2 part of **T_Cho** are synchronised on the actions in $\mathbf{Act}_c(b_{n'}, n')$. So if the activated $\mathbf{T}_c(b_{n'})$ decides to enable further such actions, that can not be without co-operation of the b_n part, i.e. of its **Det**.

If the **Det** is not activated, $\mathbf{T}_c(b_n)$ and $\mathbf{Det}_c(b_n, n)$ in the $b_{n'}$ part are executed and their termination followed by a common **lt**.

The **T_Cho** is with respect to SPs and **sync** actions equivalent to $\mathbf{H'}_c(b)$, and so is $\mathbf{T}_c(b)$. It remains to prove that $\mathbf{T}_c(b)$ properly implements the disruptive effects of the starting SPs of b at c'.

A starting SP s' of a $b_{n'}$ at c' disruptive for a starting SP s of b_n at c is, like s, a GCSP. It can only be executed when c' is the token owner. c can not execute s after s' until it regains the token. But c never receives the token before c' has reported s' by a **sync** action (Property 7). The **sync** action will occur in $\mathbf{T}_c(b_{n'})$ before s, and disrupt $\mathbf{T}_c(b_n)$, i.e. s, with the help of $\mathbf{Det}_c(b_{n'}, n')$.

Example 13 *In Table 20, mappings* **H** *and* **T** *are given for a service part b specified as choice between a* b_1 *and a* b_2*. Its GCSPs are* a^1 *and the* d^2 *in* b_2*, hence in each* $\mathbf{T}_c(b)$*, the* b_1 *and the* b_2 *part both connect to the token upon the start. After the choice is made (e.g.*

when b_1 proceeds to the execution of d^2), the token is no longer used.

Observe that in the results of mapping $\mathbf{T}$*, there are many possibilities for further simplification modulo* $\approx$*. For example, since there are* d^2 *both in* b_1 *and* b_2*, they have been internally to* $\mathbf{T}_2(b)$ *renamed into* $\mathrm{d1}^2$ *and* $\mathrm{d2}^2$*, respectively. That is superfluous. Second, it would be legal to simplify all the* **Det** *parts, e.g. to replace* $\mathbf{Det}_1(b_1, 1)$ *specified in lines 22–23 of the table with* $(\mathrm{a}^1; \mathbf{itt}(2); \mathbf{sync}(1); \mathbf{lt})$*. For we know that in* $\mathbf{H}_1(b_1)$ *that the* **Det** *observes, the first action can only be* a^1*, while the rest of the* $\mathbf{H}$ *is always* $\mathbf{sync}(1)$*.*

$b = ((\mathrm{a}^1; \mathrm{d}^2)[]\mathrm{d}^2)$
$\mathbf{H}_1(b) \approx ((\mathrm{a}^1; \mathbf{sync}(1))[]\mathbf{sync}(2))$
$\mathbf{H}_2(b) \approx ((\mathbf{sync}(1); \mathrm{d}^2)[](\mathrm{d}^2; \mathbf{sync}(2)))$
$\mathbf{T}_1(b) \approx$
$(\mathbf{ren}\ \mathbf{it}(1) \to \mathbf{tok}(1), \mathbf{it}(2) \to \mathbf{tok}(2),$
$\quad \mathbf{ct}(1) \to \mathbf{tok}(1), \mathbf{ct}(2) \to \mathbf{tok}(2)\ \mathbf{in}$
$\ \mathbf{hide}\ \mathbf{ist}, \mathbf{itt}, \mathbf{lt}\ \mathbf{in}$
$\ (((\mathbf{ren}\ \mathbf{tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2),$
$\quad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in}$
$\quad ((\mathbf{ist}(1); (\mathbf{tok}(1)[](\mathbf{tok}(2); \mathbf{tok}(1)));$
$\quad\ ((\mathrm{a}^1[x>(\mathbf{tok}(2); \mathbf{tok}(1); x))|[\mathrm{a}^1, \mathbf{tok}]|$
$\quad\ ((\mathbf{loop}\ \mathbf{tok}(\mathbf{any} : \{1, 2\}))[>(\mathrm{a}^1; \mathbf{sync}(1))));$
$\quad\ \mathbf{itt}(1); \mathbf{lt})$
$\quad [>(\mathbf{sync}(2); \mathbf{itt}(1); ((\mathbf{loop}\ \mathbf{sync}(2))[>\mathbf{lt}))))$
$\ |[\mathrm{a}^1, \mathbf{sync}, \mathbf{ct}, \mathbf{lt}]|$
$\ (\mathbf{ren}\ \mathbf{tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2),$
$\quad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in}$
$\ ((\mathbf{ist}(2);$
$\quad (((\mathbf{loop}\ (\mathbf{tok}(1); \mathbf{tok}(2)))[](\mathbf{loop}\ (\mathbf{tok}(2); \mathbf{tok}(1))))$
$\quad\ [>\mathbf{sync}(2));$
$\quad\ \mathbf{itt}(2); \mathbf{lt})$
$\quad [>((\mathrm{a}^1[]\mathbf{sync}(1)); \mathbf{itt}(2);$
$\quad\ (((\mathbf{loop}\ \mathrm{a}^1)|||(\mathbf{loop}\ \mathbf{sync}(1)))[>\mathbf{lt})))))$
$\ |[\mathbf{it}, \mathbf{ct}, \mathbf{ist}, \mathbf{itt}, \mathbf{lt}]|\mathbf{TokC}))$
$\mathbf{T}_2(b) \approx$
$(\mathbf{ren}\ \mathbf{it}(1) \to \mathbf{tok}(1), \mathbf{it}(2) \to \mathbf{tok}(2),$
$\quad \mathbf{ct}(1) \to \mathbf{tok}(1), \mathbf{ct}(2) \to \mathbf{tok}(2),$
$\quad \mathrm{d1}^2 \to \mathrm{d}^2, \mathrm{d2}^2 \to \mathrm{d}^2\ \mathbf{in}$
$\ \mathbf{hide}\ \mathbf{ist}, \mathbf{itt}, \mathbf{lt}\ \mathbf{in}$
$\ (((\mathbf{ren}\ \mathbf{tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2),$
$\quad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in}$
$\quad ((\mathbf{ist}(1);$
$\quad\ (((\mathbf{loop}\ (\mathbf{tok}(1); \mathbf{tok}(2)))[](\mathbf{loop}\ (\mathbf{tok}(2); \mathbf{tok}(1))))$
$\quad\ [>\mathbf{sync}(1));$
$\quad\ \mathbf{itt}(1); \mathrm{d1}^2; \mathbf{lt})$
$\quad [>((\mathrm{d2}^2[]\mathbf{sync}(2)); \mathbf{itt}(1);$
$\quad\ (((\mathbf{loop}\ \mathrm{d2}^2)|||(\mathbf{loop}\ \mathbf{sync}(2)))[>\mathbf{lt}))))$
$\ |[\mathrm{d2}^2, \mathbf{sync}, \mathbf{ct}, \mathbf{lt}]|$
$\ (\mathbf{ren}\ \mathbf{tok}(1) \to \mathbf{it}(1), \mathbf{tok}(2) \to \mathbf{it}(2),$
$\quad \mathbf{tok}(1) \to \mathbf{ct}(1), \mathbf{tok}(2) \to \mathbf{ct}(2)\ \mathbf{in}$
$\ ((\mathbf{ist}(2); (\mathbf{tok}(2)[](\mathbf{tok}(1); \mathbf{tok}(2)));$
$\quad ((\mathrm{d2}^2[x>(\mathbf{tok}(1); \mathbf{tok}(2); x))|[\mathrm{d2}^2, \mathbf{tok}]|$
$\quad\ ((\mathbf{loop}\ \mathbf{tok}(\mathbf{any} : \{1, 2\}))[>(\mathrm{d2}^2; \mathbf{sync}(2))));$
$\quad\ \mathbf{itt}(2); \mathbf{lt})$
$\quad [>(\mathbf{sync}(1); \mathbf{itt}(2); ((\mathbf{loop}\ \mathbf{sync}(1))[>\mathbf{lt})))))$
$\ |[\mathbf{it}, \mathbf{ct}, \mathbf{ist}, \mathbf{itt}, \mathbf{lt}]|\mathbf{TokC}))$

Table 20: Specifications for Example 13

6.6 Token Management for Disabling

In a b specified as $(b_1[>b_2)$ (5), the disabling operator introduces concurrency and conflicts between the SPs and the termination of b_1 on one side, and the starting SPs of b_2 on the other. Hence we can implement b in a similar manner as a $(b_1[]b_2)$, except that $\mathbf{T}_c(b_1)$ and $\mathbf{T}_c(b_2)$ now disrupt each other on a different set of actions. The corresponding analogues from Tables 19 and 21 are $\mathbf{Cho}(\mathbf{T}, b_1, b_2)$ and $\mathbf{Dis}(\mathbf{T}, b, b_1, b_2)$, $\mathbf{T\_Cho}_c(b_1, b_2)$ and $\mathbf{T\_Dis}_c(b, b_1, b_2)$, $\mathbf{T\_Alt}_c(b_1, b_2, 1, 2)$ and $\mathbf{Norm}_c(b, b_1, b_2)$, and $\mathbf{T\_Alt}_c(b_2, b_1, 2, 1)$ and $\mathbf{Intr}_c(b, b_1, b_2)$, respectively. The function name **Norm** indicates that b_1 is the normally executed part of $(b_1[> b_2)$, while function name **Intr** indicates that b_2 in the b interrupts b_1. In the **T_Dis**, the b_1 part and the b_2 part are no longer synchronised on the starting SPs of b_1, as they are in the **T_Cho**, because the SPs are no longer disruptive for the starting SPs of b_2.

$\mathbf{T'}_c(b)$ where $(b = (b_1[>b_2)) =$
$\ \mathbf{if}\ GS(b)\ \mathbf{then}$
$\quad \mathbf{ExtTok}((\mathbf{T\_Dis}_c(b, b_1, b_2)|[\mathbf{it}, \mathbf{ct}, \mathbf{ist}, \mathbf{itt}, \mathbf{lt}]|\mathbf{TokC}))$
$\ \mathbf{else}\ \mathbf{Dis}_c(\mathbf{T}, b, b_1, b_2)\ \mathbf{endif}$
$\mathbf{T\_Dis}_c(b, b_1, b_2) =$
$\ \mathbf{ExtSP}_c((\mathbf{IntTok}(\mathbf{Norm}_c(b, b_1, b_2), 1)$
$\quad\ |[(\{u2^c|(u^c \in SS(b_2))\} \cup \{\mathbf{sync}, \mathbf{ct}, \mathbf{lt}\})]|$
$\quad\ \mathbf{IntTok}(\mathbf{Intr}_c(b, b_1, b_2), 2)),$
$\quad\ b_1, b_2)$
$\mathbf{Norm}_c(b, b_1, b_2) = ((\mathbf{IntSP}_c(\mathbf{T}_c(b_1), b_1, 1); \mathbf{sync}(i(b)); \mathbf{lt})$
$\quad\ [>\mathbf{Det}_c(b_2, 2))$
$\mathbf{Intr}_c(b, b_1, b_2) = (((\mathbf{IntSP}_c(\mathbf{T}_c(b_2), b_2, 2); \mathbf{lt})$
$\quad\ |[\mathbf{lt}]|((\mathbf{loop}\ \mathbf{sync}(\mathbf{any} : I(b_1)))[>\mathbf{lt}))$
$\quad [>(\mathbf{sync}(i(b)); \mathbf{tt}; \mathbf{lt}))$

Table 21: Mapping $\mathbf{T}'$ for disabling

Like $\mathbf{T\_Alt}_c(b_1, b_2, 1, 2)$, the **Norm** implements b_1 disrupted upon the start of b_2. The only difference is that in **Norm**, $\mathbf{T}_c(b_1)$ is followed by a $\mathbf{sync}(i(b))$ upon which the components corporately decide to terminate b by terminating b_1.

In the **Intr**, $\mathbf{T}_c(b_2)$ for which it is basically responsible is not disrupted upon the start of b_1, like in $\mathbf{T\_Alt}_c(b_2, b_1, 2, 1)$, but upon the $\mathbf{sync}(i(b))$. Afterwards $(\mathbf{sync}(i(b)); \mathbf{tt}; \mathbf{lt})$, the analogue of $\mathbf{Det}_c(b_1, 1)$, informs **TokC** that the b_2 part is disconnecting from the token, and terminates $\mathbf{T}_c(b)$ by synchronising with the b_1 part upon its **lt** following the $\mathbf{sync}(i(b))$. While b_1 is stile active, **Intr** must also support unlimited execution of **sync** actions in $\mathbf{T}_c(b_1)$, because **Norm** and **Intr** are synchronised on gate **sync**.

The **T_Dis** is with respect to SPs and **sync** actions equivalent to $\mathbf{H'}_c(b)$, and so is $\mathbf{T}_c(b)$. It remains to prove that the starting SPs of b_2 properly disrupt the SPs of b_1. With the disruptions implemented as in the case of $(b_1[]b_2)$, we know that the requirement is satisfied.

Example 14 *In Table 22, mappings* $\mathbf{H}$ *and* $\mathbf{T}$ *are*

given for a service part b specified as a b_1 potentially disabled by a b_2. Observe that b is the same as in Example 13, except that the composition operator is changed. Consequently, implementations specified in Tables 22 and 20, respectively, are very similar, particularly in their b_1 parts, as expected from Tables 21 and 19, respectively. Again the GCSPs are a^1 *and the* d^2 *in b_2. The reader is encouraged to observe the developments following the disabling of b_1 in its various stages, and the termination of b upon termination of b_1.*

7 Discussion and Conclusions

7.1 Correctness

A detailed correctness proof for the proposed mapping **T** is given in [8]. There we define a set of properties that the implementation of each individual service part is expected to possess. Semantically, the properties are similar to those proposed in the paper. We prove them for the implementation of individual SPs. Then we prove for every type of composition operators, that the implementation of the composite behaviour inherits the properties from the implementations of its constituents. So we have been able to prove that, by induction on the service specification structure, **hide sync in** $(p_c|[\mathbf{sync}]|p_{c'}) \approx p$ for any service p and the derived server component behaviours p_c and $p_{c'}$.

7.2 Computation of Service Attributes

Since there are no cycles in the dependence between attributes of various types, it is possible to compute them one after another, in the order in which they have been introduced in the paper. For an individual attribute type, computation of the attribute for individual service parts proceeds either from composite behaviours to their parts, or vice versa, depending on the nature of the attribute.

The only attribute for which computation has not been precisely specified is $i(b)$, but one can always resort to the simple solution in which every service part b is assigned a different $i(b)$. However, when the protocol derivation method is generalised to infinite service behaviours and buffered protocol channels, a solution supporting re-use of protocol messages, i.e. minimising the number (and thereby the length) of different $i(b)$, will be required.

7.3 Service Specification Language

Currently, not all behaviour types that one would expect in a Basic-LOTOS-like sublanguage of E-LOTOS are allowed in service specifications. In comparison to [2], the missing types are inaction **stop**, successful termination δ, anonymous internal action **i**, hiding, renaming, and process instantiation.

For a realistic service, the only place where it might be useful to specify a **stop** would be to indicate that in a $(b_1[>b_2)$, it might happen that b_1 doesn't terminate, so that b_2 must be activated. Inaction can be

$b = ((\mathrm{a}^1; \mathrm{d}^2)[>\mathrm{d}^2)$
$\mathbf{H}_1(b) \approx$ (**hide lt in**
(((a^1; **sync**(1); **sync**(3))[>(**sync**(2); **lt**))
|[**sync**, **lt**]|
((**loop sync**(**any** : {1, 2}))[>(**sync**(3)[]**lt**))))
$\mathbf{H}_2(b) \approx$ (**hide lt in**
(((**sync**(1); d^2; **sync**(3))[>(d^2; **sync**(2); **lt**))
|[d^2, **sync**, **lt**]|
(((**loop** d^2)|||(**loop sync**(**any** : {1, 2})))
[>(**sync**(3)[]**lt**))))
$\mathbf{T}_1(b) \approx$
(**ren it**(1) → **tok**(1), **it**(2) → **tok**(2),
ct(1) → **tok**(1), **ct**(2) → **tok**(2) **in**
hide ist, **itt**, **lt in**
(((**ren tok**(1) → **it**(1), **tok**(2) → **it**(2),
tok(1) → **ct**(1), **tok**(2) → **ct**(2) **in**
((**ist**(1); (**tok**(1)[](**tok**(2); **tok**(1)));
((a^1[x>(**tok**(2); **tok**(1); x))|[a^1, **tok**]|
((**loop tok**(**any** : {1, 2}))[>(a^1; **sync**(1))));
itt(1); **sync**(3); **lt**)
[>(**sync**(2); **itt**(1); ((**loop sync**(2))[>**lt**))))
|[**sync**, **ct**, **lt**]|
(**ren tok**(1) → **it**(1), **tok**(2) → **it**(2),
tok(1) → **ct**(1), **tok**(2) → **ct**(2) **in**
(((**ist**(2);
(((**loop** (**tok**(1); **tok**(2)))[](**loop** (**tok**(2); **tok**(1))))
[>**sync**(2));
itt(2); **lt**)
|[**lt**]|((**loop sync**(1))[>**lt**))
[>(**sync**(3); **itt**(2); **lt**))))
|[**it**, **ct**, **ist**, **itt**, **lt**]|**TokC**))
$\mathbf{T}_2(b) \approx$
(**ren it**(1) → **tok**(1), **it**(2) → **tok**(2),
ct(1) → **tok**(1), **ct**(2) → **tok**(2),
d1^2 → d^2, d2^2 → d^2 **in**
hide ist, **itt**, **lt in**
(((**ren tok**(1) → **it**(1), **tok**(2) → **it**(2),
tok(1) → **ct**(1), **tok**(2) → **ct**(2) **in**
((**ist**(1);
(((**loop** (**tok**(1); **tok**(2)))[](**loop** (**tok**(2); **tok**(1))))
[>**sync**(1));
itt(1); d1^2; **sync**(3); **lt**)
[>((d2^2[]**sync**(2)); **itt**(1);
(((**loop** d2^2)|||(**loop sync**(2)))[>**lt**))))
|[d2^2, **sync**, **ct**, **lt**]|
(**ren tok**(1) → **it**(1), **tok**(2) → **it**(2),
tok(1) → **ct**(1), **tok**(2) → **ct**(2) **in**
(((**ist**(2); (**tok**(2)[](**tok**(1); **tok**(2)));
((d2^2[x>(**tok**(1); **tok**(2); x))|[d2^2, **tok**]|
((**loop tok**(**any** : {1, 2}))[>(d2^2; **sync**(2))));
itt(2); **lt**)
|[**lt**]|((**loop sync**(1))[>**lt**))
[>(**sync**(3); **itt**(2); **lt**))))
|[**it**, **ct**, **ist**, **itt**, **lt**]|**TokC**))

Table 22: Specifications for Example 14

easily implemented, for both components as a **stop** [2]. The only remaining task would then be to identify the parts of the derived specifications that the **stop**s introduced in the protocol make non-executable or redundant.

With every SP by definition followed by an implicit δ, explicit specification of δ would serve some purpose only in a decision-making position. But there is no such situation, since δ is by definition never decision-making in E-LOTOS.

If we introduce into a service specification actions that are inherently internal or explicitly hidden from the service users, that can strongly simplify the derived protocol. For remember (Section 4) that internal actions indicate cases where the server is allowed to resolve a conflict internally, without users' co-operation. Full exploitation of such possibilities is for further study.

According to [2], action renaming should be allowed only without changing the location of the SPs. In that case, its implementation would be simple, because renaming commutes with mapping **T**. For note that even if we are implementing a b that combines a b_1 and a b_2 by an operator that introduces additional conflicts, the only properties of the involved SPs that influence the protocol are their position within b_1 or b_2 and the location of their execution. Hence if some renaming of the SPs in b is required, we may as well specify it on the component behaviours derived for b.

Process instantiation is easy to implement [2]. The only real problem is that a service part b within a dynamically activated process instance must, in principle, be assigned its $i(b)$ dynamically. If the number of the active process instances grows with time, more and more new identifiers for their parts are introduced, i.e. the protocol messages for reporting their completion are longer and longer. Hence processes should only be instantiated in positions where re-use of service part identifiers is possible. One also has to be careful with service behaviours combining finite and infinite (i.e. non-terminating) alternatives [7].

In a more distant future, it would of course be nice to have a method for deriving protocols for services specified in full E-LOTOS and with no restrictions on SPs partitioning.

7.4 Legibility of the Derived Specifications

In Table 22, for example, we see that token management makes the derived specifications very complex, even for the simplest services. However, if a presentation tool is instructed to show specifications one hierarchical level at a time, and if the presentation is in an abstract form, like for example the one in Table 21, specifications might become quite readable. The important property is that they always reflect the structure of the service specification.

7.5 Adapting the Method to Asynchronous Protocol Channels

As demonstrated in [6], asynchronous channels bring at least two problems, even if they are reliable, first-in-first-out and with an unlimited buffer capacity.

1) When implementing a $(b_1|[S]|b_2)$ with $(S \neq \emptyset)$, the implementations of the two parts might interfere on the protocol channels they share.

2) When a $\mathbf{T}_c(b)$ is disrupted, that prevents c from properly receiving the incoming protocol messages belonging to the implementation of b. That would be particularly problematic in implementation of disabling, as evident from **Norm** in Table 21.

Because our mapping **T** introduces TE as an additional activity with its own protocol messages, the above problems would be even more acute.

7.6 Adapting the Method to Multi-Party Servers

On a multi-party server, the set of the server components with SPs conflicting with a particular SP at a c might be strongly changing with time.

Example 15 *As an illustration, consider the service behaviour* $((\mathrm{a}^1|||\mathrm{d}^1)[]\mathrm{g}^2)$. *Initially, all the three SPs are GCSPs. Suppose that* a^1 *is executed and the* g^2 *alternative consequently disrupted. After that, the remaining SP* d^1 *is no longer a GCSP, but our current protocol derivation method does nothing to stop the TE for its needs.*

So one would really like to be able to derive protocols that detect and handle conflicts dynamically. Our method is not very good at it. Its straightforward generalisation to multi-party servers would be a token travelling between components in a pre-defined order, i.e. pretending that for every GCSP, its executor is by definition in conflict with all the other components, which is seldom true. Even if conflicts are determined statically, TE should be more flexible.

Ideally, there would be a separate token for each pair of components. To prevent execution of a GCSP, it would suffice to deprive its executor of one of the tokens required for the SP. Based on the minimal requirement, an efficient TE subprotocol should be developed.

7.7 Alternative Solutions

Virtual Global Variables

In [9], it is indeed assumed that a server for which a service-implementation protocol is to be derived runs a subprotocol for mutual exclusion of the conflicting SPs. Namely, SPs are allowed to concurrently access virtual global variables, where the subprotocol takes care that the only form of concurrency on a variable is concurrent reading. The basic idea is elegant, but

[9] is just an initial study on how it could be used for LOTOS-based protocol derivation.

1) In [9], the subprotocol is not integrated into the derived protocol specification. An SP is always represented in the specification as an individual action, even if it is known to be a complex transaction negotiating for access of the addressed variables. From Section 6.5 we know that in a LOTOS-like language (like the one adopted in [9]), replacing of an individual action with a more complex behaviour typically requires complicated restructuring of the specification, if the action is in a decision-making position. In [9], nothing is said of the necessary restructurings.

2) If a service specification consists of multiple hierarchical levels combining choice and/or disabling operators with synchronised parallel composition, it might be unavoidable that the enabling conditions of individual SPs are complicated functions of many global variables. In the presence of a multitude of such actions competing for the variables' access, it would be difficult to find conditions securing correctness of the implementation, not to mention the fairness issues. In [9], not much guidance is given for the task.

3) The method of [9] is based on [5], that is known to sometimes generate erroneous protocols [6].

Time Multiplex

Global conflicts can sometimes be efficiently resolved by additional real-time constraints [12]. E-LOTOS nicely supports specification of real-time properties. One could for example divide the global time into intervals allocated for GCSPs belonging to individual server components. In that way, the token-passing actions would be simply the selected ticks of the global clock, not contributing to the protocol traffic. Developing such a protocol derivation method, we have observed that the only real problem for its application is the requirement that the maximum transit delay of the protocol channels is short with respect to the time intervals, for otherwise Property 7 is violated.

7.8 Conclusions

Let us conclude by observing that prohibition of global conflicts is the most stubborn source of unrealistic restrictions on the services subjected to automated protocol derivation. In the present paper we have demonstrated that it is feasible to derive protocols in a compositional way even if there are global conflicts in the considered service.

Acknowledgement

The presentation style of the paper has been very much improved thanks to numerous kind suggestions of the anonymous referees.

References

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

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

[3] J. Quemada (ed.). Enhancements to LOTOS, July 2000. ISO/IEC FCD 15437 (E-LOTOS).

[4] M. Hulström. Structural decomposition. In *Protocol Specification, Testing and Verification XIV*, pages 201–216. Chapman&Hall, 1995.

[5] C. Kant, T. Higashino, and G. von Bochmann. Deriving protocol specifications from service specifications written in LOTOS. *Distributed Computing*, 10(1):29–47, 1996.

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

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

[8] M. Kapus-Kolar. Global conflict resolution in automated service-based protocol synthesis. Ljubljana, 2000. Jožef Stefan Institute Technical report 8221.

[9] A. Khoumsi and G. von Bochmann. Protocol synthesis using basic LOTOS and global variables. In *Proc. Int. Conf. on Networks and Protocols*, pages 126–133, Tokyo, 1995.

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

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

[12] A. Nakata, T. Higashino, and K. Taniguchi. Protocol synthesis from timed and structured specifications. In *Proc. Int. Conf. on Networks and Protocols*, pages 74–81, Tokyo, 1995.

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

[14] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. On the use of specification styles in the design of distributed systems. *Theoretical Computer Science*, 89(1):179–206, October 1991.