Basic E-LOTOS and Its Application to Protocol Synthesis

Osnovni E-LOTOS in njegova uporaba v sintezi protokolov

Monika Kapus-Kolar

Jožef Stefan Institute, POB 3000, SI-1001 Ljubljana, Slovenia

Phone: +386 61 1773 531, Fax: +386 61 1262 102, Email: monika.kapus-kolar@ijs.si

Abstract: A transformation is described which, given a specification of the expected service of a distributed system and a partitioning of the specified service actions among the system components, derives behaviour of individual components implementing the service. There may be an arbitrary finite number of the components, pairwise communicating over reliable, unbounded first-in-first-out channels. The adopted specification language is an abstraction of basic E-LOTOS. Compared to the earlier proposal on which it is based, the method is substantially simplified.

Povzetek: Članek opisuje transformacijo za izpeljavo obnašanja komponent porazdeljenega sistema, ki implementira podano pričakovano storitev sistema, pri čemer za vsako posamezno storitveno akcijo že vemo, katera komponenta naj bi jo izvajala. Dovoljeno je poljubno končno število komponent, ki paroma komunicirajo preko zanesljivih kanalov, ki ohranjajo vrstni red sporočil. Uporabljeni specifikacijski jezik je abstrakcija osnovnega dela jezika E-LOTOS. Predlagana metoda je bistveno enostavnejša od starejše, iz katere izhaja.

Keywords: distributed service implementation, automated protocol derivation, E-LOTOS.

Ključne besede: porazdeljena implementacija storitev, avtomatska izpeljava protokolov, E-LOTOS.

1 Introduction

The paper addresses the following problem: Given a specification of the required external behaviour (the expected service) of a distributed system and a partitioning of the service actions among the service access points, derive a protocol implementing the service. It is assumed that each service access point is supported by a system component, where the components coordinate their activities by communicating via reliable, unbounded first-in-first-out channels.

Transformations for automated protocol derivation (functionality decomposition) are becoming more and more important, as they facilitate rapid prototyping of new services for specific user needs, that is an imperative in modern intelligent telecommunications networks and global computer networks. If the transformations are implemented in a tool, they can even be employed by non-specialists. Their applicability is as wide as that of the specification language on which they are based. We base our transformation on E-LOTOS [2], an enhanced version of LOTOS [1], a standard formal language intended for specification of reactive and concurrent systems. The applicability of the languages ranges over all kinds of discrete systems: technical and social, material and logical [7].

During the last decade, protocol synthesis has been subject to intensive research. [6], an exhaustive survey of the existing methods, also points to several such methods for LOTOS-like languages, among them [3] being probably the most important one. Unfortunately [3] contains some errors, so we have proposed an improved method [4], that has subsequently been extended [5] to support implementation of the basic behaviour types newly introduced by E-LOTOS.

The solution in [5] is quite complicated because it facilitates, like [4] for LOTOS, protocol optimization through supporting for distributed implementation of individual service parts a multitude of termination types. As it is in many cases difficult to select the best one, one might prefer to rely on a single termination type that usually works quite well. Such is the termination type employed in [3]. In the present paper, we show how to employ it for E-LOTOS. To further simplify the protocol derivation transformation [5], we pose additional restrictions on the service specification structure, that can easily be satisfied by insertion of dummy service actions. Anyhow, the proposed solution is just a special case of [5] (except for some additional optimizations), thus no thorough discussion on its correctness is given. Instead we concentrate on explaining the transformation to a practitioner. The paper is also intended for promotion of E-LOTOS.

The paper is organized as follows. Section 2 introduces the adopted specification language, i.e. makes the reader familiar with the basic constructs of E-LOTOS. In Section 3 we explain the protocol derivation transformation. Section 4 brings a discussion and conclusions.

2 Specification Language

The language defined in Table 1 has been conceived with an aim to include in an abstract and

concise way all the main constructs of the basic behaviour sublanguage of E-LOTOS [2], in the exclusive setting of the protocol derivation problem. Time is not considered, there are no variables and no structuring into explicit processes, and actions are only allowed to have parameters that are constants, i.e. parts of their names.

The following typographical convention has been adopted: If $\mathcal{X}$ is some universe of elements where 'x' denotes its name and stands for any letter, then variables x, x',..., x_0,... range over elements of $\mathcal{X}$ and variables X, X',..., X_0,... over subsets of $\mathcal{X}$, if not stated otherwise. The cases where an X (or a $\{...\}$) identifies a multi-set rather than a set are marked with an *. In particular, a b stands for a behaviour (for a process exhibiting it), a c for a system component, a t for a trappable action, a g for an interaction gate or an action on it, a u or a w for a user-defined action type, an x for an exception, an m for a protocol message, a h for an action handler, an s for a synchronization, an n for a natural number, and an r for a renaming. $Init(b)$ denotes the initial actions of b.

Concentrate on the semantics of the language, informally overviewed below, for its syntax is intentionally kept simple, to simplify presentation of protocol derivation. Throughout the paper, the usual self-understood forms of "syntactic sugar" are used where convenient, e.g. switching between the infix and the prefix notation, parentheses, omission of implicitly implied

Name of the construct	Type(s)	Syntax
Inaction	b	**stop**
Successful termination	b,t	δ^c
Internal action	b	$\mathbf{i}^c$
Service primitive	b,g	u^c
Exception (raising)	b,t,x	w^c
Protocol message transmission	b,g	$\mathbf{s}_c(m)$
Protocol message reception	b,g	$\mathbf{r}_c(m)$
Action trapping	b	**trap** H **in** b_1
Handling a trapped action	h	$t \rightarrow b'$
Choice	b	$b_1[]b_2$ where $\delta \notin (Init(b_1) \cup Init(b_2))$
Suspend/resume	b	$b_1[x>b_2$ where $((Init(b_2) \cap \{\delta,x\})=\varnothing)$
Parallel composition	b	**par** $[S]$ $\{[G(b')]b'\mid(b' \in B^*)\}^*$
Synchronization	s	$g\#n$
Renaming	b	**ren** R **in** b_1
Renaming an action	r	$u_1^u \rightarrow u_2^u$ or $w_1^u \rightarrow w_2^u$
Hiding	b	**hide** G **in** b_1
Service specification		**serv**:=b
Component specification		$\mathbf{comp}_c$:=b

Table 1: The specification language abstract syntax

parts of the specification, etc. Illustrative service and protocol specifications can be found in Table 5 to Table 7.

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

Actions with reserved names δ and **i** respectively denote successful termination and an internal action of the specified process. In a service specification, they are furnished with a superscript c indicating the component controlling their execution, but the superscript doesn't belong to the action name - the selection of c influences the protocol derivation algorithm, but is irrelevant for the service itself. There is typically no need to have explicitly specified δ actions in a service specification, for any non-exceptional action is implicitly followed by a δ. By executing a δ, a process becomes inactive.

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

w^c denotes a type w exception originating from the system component c. Raising an exception within a process stops its execution, just like a δ.

A component can send (receive) a protocol message m to (from) another component c by an $\mathbf{s}_c(m)$ ($\mathbf{r}_c(m)$).

"**trap** H **in** b_1" denotes a process executing b_1 up to the point where a trappable action t with an unique handler b' listed in H is enabled. At that point, control is atomically transferred to b', unless some other alternative is selected for b_1. "$b_1;b_2$" denotes sequential composition, i.e. the special case of trapping where only δ is trapped in b_1, and b_2 is its handler. Iteration "**loop** b_1" is equivalent to "b_1;**loop** b_1".

"$b_1[]b_2$"denotes a process ready to behave as b_1 or as b_2. We define $[](\{\})$=**stop**.

"$b_1[x>b_2$" denotes a process with behaviour b_1 repeatedly interrupted by behaviour b_2. b_1 is resumed upon x trapped in b_2, while b_2 is at that point restarted. While b_1 is still active, the process might internally decide to terminate by enabling a δ in b_1. "$b_1[>b_2$"denotes the special case where the resumption exception x is never enabled in b_2, i.e. a LOTOS-like disabling.

"**par** $[S]$ $\{[G(b')]b'|(b'\in B^*)\}^*$" denotes parallel composition of processes in B^*. Each process b' listed in B^* has a list $G(b')$ of gates requiring synchronization with other members of B^*, while execution of the internal actions and exceptions of b' and the actions on gates not listed in $G(b')$ is entirely at discretion of b'. An action on a gate g in $G(b')$ can be executed as a common action of b' and some other members of B^* when they all ready for it. If g is not ex-

plicitly mentioned in the list of the legal synchronizations S, it requires co-operation of all members of B^* having it in their G lists, while a $g\#n$ in S specifies that it is appropriate if n of them synchronize. δ by definition requires co-operation of all the parallel processes. Composition operator "|||" shortly specifies independent parallel execution of processes. We define $|||(\{\})=\delta$.

"**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 listed in G, i.e. makes the gate actions internal to b_1.

The relation used throughout the paper for judging equivalence of behaviours is observational equivalence $\approx$ [1], i.e. we are interested only into the external behaviour of processes.

3 Protocol Derivation

3.1 General Considerations

The *protocol derivation problem* is precisely defined as follows. Given

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

derive such behaviour of individual components that, when the protocol interactions are hidden, the overall system behaviour is observationally

No.	b
1	**stop**
2	$\delta^{c'}$, $\mathbf{i}^{c'}$, $u^{c'}$
3	$w^{c'}$
4	$b_1;b_2$
5	**loop** b_1
6	**trap** H **in** b_1 where 1. $\neg\exists(\delta\rightarrow b_2)\in H$ 2. $\forall(x\rightarrow b')\in H:(SC(b')=PC(x))$ 3. $((EC(b_1)=\varnothing)\vee(PC(b_1)=PC(b)))\wedge$ $\forall(x\rightarrow b')\in H:((EC(b')=\varnothing)\vee(PC(b')=PC(b)))$
7	$b_1[]b_2$ where 1. $\exists c:(SC(b_1)=SC(b_2)=\{c\})$ 2. $((EC(b_1)=\varnothing)\vee(PC(b_1)=PC(b)))\wedge$ $((EC(b_2)=\varnothing)\vee(PC(b_2)=PC(b)))$
8	$b_1[x'>b_2$ where 1. $\exists c:(PC(b_1)=SC(b_2)=\{c\})$ 2. $EC(b_1)=\varnothing$
9	**par** $[S]$ $\{[G(b')]b''\mid(b''\in B^*)\}^*$ where 1. $\neg\exists[b',b'']\subseteq B^*:((G(b')\neq\varnothing)\wedge(G(b'')\neq\varnothing)\wedge$ $(\mid(PC(b')\cap PC(b''))\mid>1))$ 2. $\neg\exists[b',b'']\subseteq B^*:(NX_x(b')\wedge((PC(b'')\setminus PC(x))\neq\varnothing))$
10	**ren** R **in** b_1
11	**hide** G **in** b_1

Table 2: The legal service behaviours

equivalent to the specified service and the system never stops with any of its channels non-empty.

Below we define a *mapping* $\mathbf{T}_c(b)$ which takes a service subbehaviour b and translates it into its counterpart at a component c. A service specification "**serv**:=b" as a whole is mapped into "$\mathbf{comp}_c$:=$\mathbf{T}_c(b)$".

Mapping **T** is guided by *precomputed attributes* of individual service parts. First of all, each part b must be assigned a unique identifier $I(b)$. We are also interested into its starting, its ending and all its participating components ($SC(b)$, $EC(b)$, and $PC(b)$), respectively executing a starting action, an ending action in the case of a successful termination, or any action within b. Let for $X\in\{S,E,P\}$ $XC_c(b)$ denote $c\in XC(b)$. Another attribute $NX_x(b)$ indicates that x is an exception raised, but not handled, within b. For illustration, see the examples in Table 5 to Table 7.

Rules for computation of the attributes are defined in Table 3, where the first column points, like in Table 4, to the legal service behaviour from Table 2 to which the row applies. Note that the EC rule for parallel composition is satisfactory only for independent execution, while synchronization requires investigation of the considered behaviour step-by-step.

If a c doesn't participate in a b, formally $\neg PC_c(b)$, then $\mathbf{T}_c(b)$ equals δ, if b has a successful termination, formally ($EC(b)\neq\emptyset$), and **stop** otherwise. For other cases, $\mathbf{T}_c(b)$ is defined in Section 3.2.

For a terminating b and a c that is not its ending component, [5] allows that execution of

No.	$SC_c(b)$	$EC_c(b)$	$PC_c(b)$	$NX_x(b)$
1	*false*	*false*	*false*	*false*
2	$c=c'$	$c=c'$	$c=c'$	*false*
3	$c=c'$	*false*	$c=c'$	$x=b$
4	$SC_c(b_1)$	$EC_c(b_2)$	$PC_c(b_1)\vee PC_c(b_2)$	$NX_x(b_1)\vee NX_x(b_2)$
5	$SC_c(b_1)$	*false*	$PC_c(b_1)$	$NX_x(b_1)$
6	$SC_c(b_1)$	$EC_c(b_1)\vee$ $\exists(x'\rightarrow b')\in H{:}EC_c(b')$	$PC_c(b_1)\vee$ $\exists(x'\rightarrow b')\in H{:}PC_c(b')$	$(NX_x(b_1)\wedge\neg\exists(x\rightarrow b')\in H)$ $\vee\exists(x'\rightarrow b')\in H{:}NX_x(b')$
7	$SC_c(b_1)$	$EC_c(b_1)\vee EC_c(b_2)$	$PC_c(b_1)\vee PC_c(b_2)$	$NX_x(b_1)\vee NX_x(b_2)$
8	$SC_c(b_2)$	$EC_c(b_2)$	$PC_c(b_2)$	$NX_x(b_1)\vee(NX_x(b_2)\wedge(x\neq x'))$
9	$\exists b'\in B^*{:}SC_c(b')$	$(\neg\exists b'\in B^*{:}(EC(b)=\emptyset))$ $\wedge\exists b'\in B^*{:}EC_c(b')$ (valid only for |||)	$\exists b'\in B^*{:}PC_c(b')$	$\exists b'\in B^*{:}NX_x(b')$
10	$SC_c(b_1)$	$EC_c(b_1)$	$PC_c(b_1)$	$(NX_x(b_1)\wedge\neg\exists(x\rightarrow x')\in H)$ $\vee\exists(x'\rightarrow x)\in H{:}NX_x(b_1)$
11	$SC_c(b_1)$	$EC_c(b_1)$	$PC_c(b_1)$	$NX_x(b_1)$

Table 3: Attribute evaluation rules

$\mathbf{T}_c(b)$ concludes either by inaction of c (later disrupted by some other service activity), or by δ initiated by c itself, or by δ initiated by another component. Here we allow only the second termination type, the usual type in LOTOS-based protocol synthesis [3].

Unfortunately [5] indicates that the adopted termination type is allowed only under special restrictions. In particular, in [5] a c sometimes extends a $\mathbf{T}_c(b)$ by reporting to some non-participant c' of b. If c' is not a starting component of the behaviour b' following b, an exception raised within b' by a component other than c' might prevent reception of the message. Thus we below pose such restrictions on the service that no such message is ever necessary.

Other types of restrictions are already known from [5]. There is, however, a slight additional limitation: To simplify mapping $\mathbf{T}$, "**trap** H **in** b_1" shall be used exclusively for exception handling, while handling of a δ within a b_1 shall always be specified as sequential composition "$b_1;b_2$" (see Section 2).

Table 4 defines three auxiliary functions for mapping $\mathbf{T}$: $\mathbf{S}_c(C,m)$ and $\mathbf{R}_c(C,m)$ respectively implement sending or receiving at a c of a message m between c and any c' in $C\backslash\{c\}$, independently for each c'. $\mathbf{X}_c(C,C',m)$ implements the actions of c necessary to perform the task of each member of C' receiving m from each member of C other than itself.

3.2 Distributed Implementation of Individual Behaviour Types

In this section we discuss implementation of individual service behaviour types by their partici-

No.	$\mathbf{T}_c(b)$ where $PC_c(b)$
2,3	b
4	$(\mathbf{T}_c(b_1);\mathbf{X}_c(EC(b_1),SC(b_2),I(b_1));\mathbf{T}_c(b_2))$
5	**loop** $(\mathbf{T}_c(b_1);\mathbf{X}_c(EC(b_1),SC(b_1),I(b_1)))$
6	$((\mathbf{trap}\ \{(x\rightarrow\mathbf{T}_c(b'))\mid(((x\rightarrow b')\in H)\wedge PC_c(x))\}\ \mathbf{in}\ \mathbf{T}_c(b_1))$ $[>([](\{\mathbf{T}_c(b')\mid(((x\rightarrow b')\in H)\wedge\neg PC_c(x))\})))$
7	$(\mathbf{T}_c(b_1)[]\mathbf{T}_c(b_2))$
8	if $NX_x(b_2)$ then $(\mathbf{T}_c(b_1)[x'>$if $PC_x(b)$ then **trap** $(x'\rightarrow(\mathbf{S}_c(PC(b),I(b_2));\mathbf{X}_c((PC(b)\backslash PC(x')),SC(b_2),I(b_2));x'))$ **in** $\mathbf{T}_c(b_2)$ else $(\mathbf{T}_c(b_2)[>(\mathbf{R}_c(PC(x'),I(b_2));\mathbf{X}_c((PC(b)\backslash PC(x')),SC(b_2),I(b_2));x'))$ endif) else $(\mathbf{T}_c(b_1)[>\mathbf{T}_c(b_2))$ endif
9	**par** $[\{g\#n\mid((g\#n\in S)\wedge PC_c(g))\}]\ \{[\{g\mid((g\in G(b'))\wedge PC_c(g))\}]\mathbf{T}_c(b')\mid(b'\in B^*)\}^*$
10	**ren** $\{(b'\rightarrow b'')\mid(((b'\rightarrow b'')\in R)\wedge PC_c(b'))\}$ **in** $\mathbf{T}_c(b_1)$
11	**hide** $\{g\mid((g\in G)\wedge PC_c(g))\}$ **in** $\mathbf{T}_c(b_1)$
$\mathbf{S}_c(C,m):=\|\|\|(\{\mathbf{s}_c(m)\mid(c'\in(C\backslash\{c\}))\})$ $\mathbf{R}_c(C,m):=\|\|\|(\{\mathbf{r}_c(m)\mid(c'\in(C\backslash\{c\}))\})$	$\mathbf{X}_c(C,C',m):=($if $c\in C$ then $\mathbf{S}_c(C',m)$ else δ endif $\|\|\|$ if $c\in C'$ then $\mathbf{R}_c(C,m)$ else δ endif)

Table 4: Implementation of individual behaviour types

pating components.

An *individual action* is implemented by the executing system component as it is (Table 4(2,3)).

Implementing a *sequential composition* "$b_1;b_2$" (Table 4(4)), a c first executes $\mathbf{T}_c(b_1)$. If an ending component of b_1, c reports local termination of b_1 to the starting components of b_2 other than itself. If a starting component of b_2, c receives all the reports on termination of b_1. Finally c executes $\mathbf{T}_c(b_2)$.

An *iteration* "**loop** b_1" is implemented in the same manner (Table 4(5), examples in Table 7), as it is equivalent to "b_1;**loop** b_1". With the universally adopted behaviour termination type, re-use of behaviour identifiers within b_1 is no longer problematic, so it suffices for the starting components of the current cycle to receive termination reports from the ending components of the previous cycle, not from all its participants, as it is necessary in the general case [5].

Implementing an *exception trapping* "**trap** H **in** b_1", a c basically executes $\mathbf{T}_c(b_1)$ (Table 4(6), an example in Table 5). Upon detecting a local exception x, c activates the implementation of its handler $\mathbf{T}_c(b')$. The adopted restrictions ensure [5] that any other system component c' is at that moment idle, and that all protocol channels are empty, at least within the implementation of the service part b, as required. The idling of such a $c' \in PC_c(b)$ is sooner or later disrupted by $\mathbf{T}_c(b')$, as $PC_c(b)$ implies $PC_c(b')$, if b' proceeds towards a successful termination (Table 2(6.3)). Likewise, if b_1 successfully terminates without any exception raised, any par-

A service part: $b =$ **trap** $(x^2 \rightarrow (e^1 ||| f^2))$ $(\delta \rightarrow ((c^1 ||| b^2);x^2))$ **in** $((a^1;d^3)[](b^1;x^2))$
Amended into: $b' :=$ **trap** $(y^2 \rightarrow (e^1 ||| f^2))$ **in** $(((a^1;d^3)[](b^1;y^2));((c^1 ||| b^2);x^2))$
Amended into: $b'' :=$ **trap** $(y^2 \rightarrow ((\mathbf{i}^2;(e^1 ||| \mathbf{i}^3)) ||| f^2))$ **in** $(((a^1;(d^3;\mathbf{i}^2))[](b^1;y^2));((c^1 ||| b^2);x^2))$
Annotated with subbehaviour attributes (I,SC,EC,PC,NX_{x2},NX_{y2}):
(**trap** $(y^2 \rightarrow ((\mathbf{i}^2{}_{1,2,2,2,F,F};(e^1{}_{\_,1,1,1,F,F} ||| \mathbf{i}^3{}_{\_,3,3,3,F,F})_{\_,13,13,13,F,F}) ||| f^2{}_{\_,2,2,2,F,F})_{\_,2,13,123,F,F}$
in $(((a^1{}_{2,1,1,1,F,F};(d^3{}_{6,3,3,3,F,F};\mathbf{i}^2{}_{\_,2,2,2,F,F})_{\_,3,2,23,F,F})_{\_,1,2,123,F,F}[]$
$(b^1{}_{3,1,1,1,F,F};y^2{}_{\_,2,\varnothing,2,F,T})_{\_,1,\varnothing,12,F,T})_{4,1,2,123,F,T};$
$((c^1{}_{\_,1,1,1,F,F} ||| b^2{}_{\_,2,2,2,F,F})_{5,12,12,12,F,F};x^2{}_{\_,2,\varnothing,2,T,F})_{\_,12,\varnothing,12,T,F})_{\_,1,\varnothing,123,T,T})_{\_,1,123,123,T,F}$

Distributed implementation:
$\mathbf{T}_1(b'') :=$ $((((a^1;(\mathbf{s}_3(2) ||| \delta);\delta)[](b^1;\mathbf{s}_2(3);\mathbf{stop}));(\delta ||| \mathbf{r}_2(4));((c^1 ||| \delta);(\mathbf{s}_2(5) ||| \delta);\mathbf{stop}))$
$[>((\delta;(\delta ||| \mathbf{r}_2(1));(e^1 ||| \delta)) ||| \delta))$
$\approx ((((a^1;\mathbf{s}_3(2))[](b^1;\mathbf{s}_2(3);\mathbf{stop}));\mathbf{r}_2(4);c^1;\mathbf{s}_2(5);\mathbf{stop})[>(\mathbf{r}_2(1);e^1))$
$\mathbf{T}_2(b'') :=$ **trap** $(y^2 \rightarrow ((\mathbf{i}^2;((\mathbf{s}_1(1) ||| \mathbf{s}_3(1)) ||| \delta);\delta) ||| f^2))$
in $(((\delta;(\delta ||| \delta);(\delta;(\delta ||| \mathbf{r}_3(6));\mathbf{i}^2))[](\delta;(\delta ||| \mathbf{r}_1(3));y^2));(\mathbf{s}_1(4) ||| \delta);((\delta ||| b^2);(\delta ||| \mathbf{r}_1(5));x^2))$
$\approx$ **trap** $(y^2 \rightarrow (\mathbf{s}_1(1) ||| \mathbf{s}_3(1) ||| f^2))$ **in** $((\mathbf{r}_3(6)[](\mathbf{r}_1(3);y^2));\mathbf{s}_1(4);b^2;\mathbf{r}_1(5);x^2)$
$\mathbf{T}_3(b'') :=$ $((((\delta;(\delta ||| \mathbf{r}_1(2));(d^3;(\mathbf{s}_2(6) ||| \delta);\delta))[]\mathbf{stop});(\delta ||| \delta);\mathbf{stop})[>((\delta;(\delta ||| \mathbf{r}_2(1));(\delta ||| \mathbf{i}^3)) ||| \delta))$
$\approx ((\mathbf{r}_1(2);d^3;\mathbf{s}_2(6);\mathbf{stop})[>\mathbf{r}_2(1))$

Table 5: An example implementation of trapping

ticipant c of b is aware of that, as in that case $PC_c(b)$ implies $PC_c(b_1)$. Transfer of control upon an x in b_1 to its handler b' is sequential, but no synchronization is required upon it, as the only starting component of b' is the executor of x, the only ending component of b_1 in the exceptional case (Table 2(6.2)). If H also contains a handler for δ in b_1 (against Table 2(6.1)), that can be amended, but might require renaming of some exceptions (see the first step in the example).

Implementing a *choice* "$b_1[]b_2$", it suffices to individually implement the two alternatives b_1 and b_2 (Table 4(7)), because there is a system component locally guarding the start of both (Table 2(7.1)). If, for example, b_1 is the selected alternative, any participant c of b will detect that before b_1 terminates, for $PC_c(b)$ implies $PC_c(b_1)$ for the case (Table 2(7.2)).

Implementing a *suspend/resume* "$b_1[x'>b_2$", a c basically executes $\mathbf{T}_c(b_1)$, potentially suspended by $\mathbf{T}_c(b_2)$ (Table 4(8), an example in Table 6). Actually, $\mathbf{T}_c(b_1)$ differs from **stop** only for a c', the only participant of b_1 and the only starting component of b_2 (Table 2(8.1)). c' locally guards suspension and resumption of b_1. If the resumption never occurs, i.e. $\neg NX_x(b_2)$, the only remaining problem is how to implement termination of b upon termination of b_1 [4], but we avoid it by forbidding termination of b_1 (Table 2(8.2)). If such a termination exists, it can be transformed into an exception and handled as such [5] (see the example). Resumption

A service part: $b = ((a^1;b^1)[x^2>((c^1;d^3)[](e^1;x^2)))$
Amended into: $b' :=$ **trap** $(y^1 \rightarrow \delta^1)$ **in** $(((a^1;b^1);y^1)[x^2>((c^1;d^3)[](e^1;x^2)))$
Amended into: $b'' :=$ **trap** $(y^1 \rightarrow (\delta^1;(\mathbf{i}^2|||\mathbf{i}^3)))$ **in** $(((a^1;b^1);y^1)[x^2>((c^1;(d^3|||\mathbf{i}^2))[](e^1;x^2)))$
Annotated with subbehaviour attributes $(I,SC,EC,PC,NX_{x2},NX_{y1})$:
$(\mathbf{trap}\ (y^1 \rightarrow (\delta^1{}_{1,1,1,1,F,F};(\mathbf{i}^2{}_{\_,2,2,2,F,F}|||\mathbf{i}^3{}_{\_,3,3,3,F,F})_{\_,23,23,23,F,F})_{\_,1,23,123,F,F})$
$\mathbf{in}\ (((a^1{}_{\_,1,1,1,F,F};b^1{}_{\_,1,1,1,F,F})_{\_,1,1,1,F,F};y^1{}_{\_,1,\varnothing,1,F,T})_{\_,1,\varnothing,1,F,T}[x^2>$
$((c^1{}_{3,1,1,1,F,F};(d^3{}_{\_,3,3,3,F,F}|||\mathbf{i}^2{}_{\_,2,2,2,F,F})_{\_,23,23,23,F,F})_{\_,1,23,123,F,F}[]$
$(e^1{}_{4,1,1,1,F,F};x^2{}_{\_,2,\varnothing,2,T,F})_{\_,1,\varnothing,12,T,F})_{2,1,23,123,T,F})_{\_,1,23,123,F,T})_{\_,1,23,123,F,F}$

Distributed implementation:
$\mathbf{T}_1(b'') :=$ **trap** $(y^1 \rightarrow (\delta^1;((\mathbf{s}_2(1)|||\mathbf{s}_3(1))|||\delta);\delta))$
in $(((a^1;(\delta|||\delta);b^1);(\delta|||\delta);y^1)$
$[x^2>(((c^1;((\mathbf{s}_2(3)|||\mathbf{s}_3(3))|||\delta);\delta)[](e^1;(\mathbf{s}_2(4)|||\delta);\mathbf{stop}))[>(\mathbf{r}_2(2);(\delta|||\mathbf{r}_3(2));x^2)))$
$\approx$ **trap** $(y^1 \rightarrow (\mathbf{s}_2(1)|||\mathbf{s}_3(1)))$
in $((a^1;b^1;y^1)[x^2>(((c^1;(\mathbf{s}_2(3)|||\mathbf{s}_3(3)))[](e^1;\mathbf{s}_2(4);\mathbf{stop}))[>(\mathbf{r}_2(2);\mathbf{r}_3(2);x^2)))$
$\mathbf{T}_2(b'') :=$ $((\mathbf{stop}[x^2>\mathbf{trap}\ (x^2 \rightarrow ((\mathbf{s}_1(2)|||\mathbf{s}_3(2));x^2))\ \mathbf{in}\ ((\delta;(\delta|||\mathbf{r}_1(3));(\delta|||\mathbf{i}^2))[](\delta;(\delta|||\mathbf{r}_1(4));x^2)))$
$[>(\delta;(\delta|||\mathbf{r}_1(1));(\mathbf{i}^2|||\delta)))$
$\approx$ $((\mathbf{stop}[x^2>\mathbf{trap}\ (x^2 \rightarrow ((\mathbf{s}_1(2)|||\mathbf{s}_3(2));x^2))\ \mathbf{in}\ (\mathbf{r}_1(3)[](\mathbf{r}_1(4);x^2)))[>\mathbf{r}_1(1))$
$\mathbf{T}_3(b'') :=$ $((\mathbf{stop}[x^2>(((\delta;(\delta|||\mathbf{r}_1(3));(d^3|||\delta))[]\mathbf{stop})[>(\mathbf{r}_2(2);(\mathbf{s}_1(2)|||\delta);x^2)))$
$[>(\delta;(\delta|||\mathbf{r}_1(1));(\delta|||\mathbf{i}^3)))$
$\approx$ $((\mathbf{stop}[x^2>((\mathbf{r}_1(3);d^3)[>(\mathbf{r}_2(2);\mathbf{s}_1(2);x^2)))[>\mathbf{r}_1(1))$

Table 6: An example implementation of suspend/resume

of b_1 upon an x' in b_2 is implemented in the following way: A c'' traps the x' to inform the other participants of b. Subsequently, all participants of b except c'' (an optimization over [5]) report to the starting component of b_2 that they are ready to restart b_2. c'' needn't report because implicitly known to be ready. Finally, any participant c of b locally re-raises x' to return control to $\mathbf{T}_c(b_1)$.

Implementing a *parallel composition* "**par** [S] {[$G(b')$]b'|($b'\in B^*$)}*" at a participant c, we implement each individual b' as $\mathbf{T}_c(b')$ and locally synchronize it with other $\mathbf{T}_c(b'')$ as prescribed by S and $G(b')$ (Table 4(9), an example in Table 7). However, for this solution to work, we must pose two restrictions. As first we require that the distributed implementation of a b' doesn't share any protocol channels with implementations of the other service parts if b' synchronizes with them. In a brute-force way that is ensured in Table 2(9.1), while Table 2(9.2) ensures that no component ever raises an exception concurrently to a service action at another component or when a protocol message is in transit, at least within the implementation of the service part b, as required.

As we allow only local *action renaming*, its implementation is simple. It commutes with mapping **T** (Table 4(10)). The same holds for implementation of *hiding* (Table 4(11)), as hiding of an action doesn't change its location.

4 Discussion and Conclusions

We have presented a method for deriving protocol specifications from service specifications written in a language that is semantically close to basic E-LOTOS. The method is basically a special case of [5], restricting the termination type for implementation of individual service parts to the one usually adopted for LOTOS [3]. The restriction has substantially simplified the

A service: **serv**:= **par** $[a^1\#2] < [a^1]$ **loop** $(a^1;b^1)$, $[a^1]$ **loop** $(a^1;b^2)$, $[a^1]$ **loop** $(a^1;b^3)$, **loop** $((a^1\|\|\|b^2);(a^1\|\|\|b^3)) >$
Annotated with subbehaviour attributes (I,SC,EC,PC):
(**par** $[a^1\#2] < [a^1]$ (**loop** $(a^1{}_{,1,1,1};b^1{}_{,1,1,1})_{,1,1,1})_{,1,\varnothing,1}$, $[a^1]$ (**loop** $(a^1{}_{1,1,1,1};b^2{}_{,2,2,2})_{2,1,2,12})_{,1,\varnothing,12}$, $[a^1]$ (**loop** $(a^1{}_{3,1,1,1};b^3{}_{,3,3,3})_{4,1,3,13})_{,1,\varnothing,13}$, (**loop** $((a^1{}_{,1,1,1}\|\|\|b^2{}_{,2,2,2})_{5,12,12,12};(a^1{}_{,1,1,1}\|\|\|b^3{}_{,3,3,3})_{,13,13,13})_{6,12,13,123})_{,12,\varnothing,123} >)_{,12,\varnothing,123}$
Distributed implementation:
comp$_1$:= **par** $[a^1\#2] < [a^1]$ **loop** $(a^1;b^1)$, $[a^1]$ **loop** $(a^1;\mathbf{s}_2(1);\mathbf{r}_2(2))$, $[a^1]$ **loop** $(a^1;\mathbf{s}_3(3);\mathbf{r}_3(4))$, **loop** $(a^1;(\mathbf{s}_3(5)\|\|\|\mathbf{r}_2(5));a^1;(\mathbf{s}_2(6)\|\|\|\mathbf{r}_3(6))) >$
comp$_2$:= ((**loop** $(\mathbf{r}_1(1);b^2;\mathbf{s}_1(2))$)|||(**loop** $(b^2;(\mathbf{s}_1(5)\|\|\|\mathbf{s}_3(5));(\mathbf{r}_1(6)\|\|\|\mathbf{r}_3(6)))$))
comp$_3$:= ((**loop** $(\mathbf{r}_1(3);b^3;\mathbf{s}_1(4))$)|||(**loop** $((\mathbf{r}_1(5)\|\|\|\mathbf{r}_2(5));b^3;(\mathbf{s}_1(6)\|\|\|\mathbf{s}_2(6)))$))

Table 7: An example implementation of iteration and parallel composition

protocol derivation mapping, enabling us to identify some additional optimizations for the special case.

With its simplicity, the method seems a good candidate for further extensions, e.g. to support distributed handling of service parameters and quantitative timing requirements, that are also expressible in E-LOTOS. Currently, implementation of the following extensions to basic LOTOS is supported: iteration, exception handling, suspension/resumption and generalized parallel composition.

As a conclusion, we express our hope that the glimpse into E-LOTOS has motivated the reader for its further study, for it shall probably develop into a powerful, user-friendly, well-supported and widely employed standard formal specification language.

References

1. ISO, *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observable Behaviour*, IS 8807, 1989.
2. ISO/IEC JTC1/SC21 WG7, Final Committee Draft on *Enhancements to LOTOS*, May 1998, see ftp://ftp.dit.upm.es/pub/lotos/elotos/Working.Docs.
3. C. Kant, T. Higashino, G. v. Bochmann, "Deriving protocol specifications from service specifications written in LOTOS," *Distributed Computing*, vol. 10, 1996, pp. 29-47.
4. M. Kapus-Kolar, "Employing disruptions for more efficient functionality decomposition in LOTOS," submitted for publication, an extended version available at Jožef Stefan Institute, Technical Report 7878, 1998, a preliminary version in *Proc.22nd EUROMICRO Conf.*, Budapest, 1997, pp. 464-471.
5. M. Kapus-Kolar, "Deriving protocol specifications from service specifications in Basic E-LOTOS," submitted for publication, an extended version available at Jožef Stefan Institute, Technical Report 7897, 1998.
6. K. Saleh, "Synthesis of communication protocols: An annotated bibliography," *Computer Communication Review*, vol. 26, no. 5, 1996, pp. 40-59.
7. World-Wide Environment for Learning LOTOS, http://www.cs.stir.uk/~kjt/research/well/well.html.