Deriving Protocols for Services Supporting Mobile Users

M. Kapus-Kolar [1]

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

Abstract

A Prolog tool for automated derivation of protocol specifications from service specifications is described. The server for which the protocol is derived may consist of any finite number of protocol entities co-operating over reliable unbounded first-in-first-out channels. Its service is expected to consist of service primitives that read or write unstructured global virtual variables, implicitly receive or compute their current values or delete their local copies. In addition, service primitives may access distributed virtual queues, to which they append elements with desired priority or consume their head elements. Service users are allowed to dynamically select the service-access point through which they interact with the distributed server. The adopted specification language has been inspired by LOTOS.

Key words: Services for mobile users; Distributed service implementation; Automated service-based protocol derivation; LOTOS

1 Introduction

Contemporary enterprises often require co-operation of a set of partners according to some predefined plan. It is convenient if each of the partners can concentrate entirely on its own work, while delegating the problems of communication and co-ordination with the others to a dedicated server. Examples of such enterprises are applications of intelligent telecommunications networks (IN), computer-supported co-operative work and automated factories.

In an abstract view, a server is a black box interacting with its users through a set of service-access points (SAPs). Its behaviour at SAPs is the service it offers. The service is designed to suit the particular plan of the enterprise. In some enterprises, e.g. in automated factories, the plan often changes, requiring implementation of a new service. Another example is introduction of new IN services, to satisfy the ever increasing users' demands.

If the users are physically distributed, so must be the server. In a more detailed, but still an abstract view, such a server consists of a set of protocol entities (PEs) supporting individual SAPs and co-ordinating their work by communicating over internal server channels. The overall activity of the PEs is often referred to as the protocol implementing the service.

Nowadays users expect that introduction of a new service is a matter of a couple of weeks. In an automated factory, that would rather be a couple of days or even hours. As an automated factory is a rather closed system, the exact nature of the supporting protocol is usually irrelevant, as long as it is correct, efficient and developed in time. The three goals can best be met by employing tools for automated derivation of protocol specifications from service specifications, that ensure protocol quality by construction. Even for protocols that are subsequently subjects to additional amendments or negotiations (e.g. for the purpose of standardisation), such a tool can be helpful, providing solutions to start with.

For that reason, many efforts have been put into automated derivation of protocol specifications from service specifications. Those methods are applicable to design of a wide range of systems, physical or logical, technical or social, wherever it is necessary to decompose a server into a number of interacting components.

A very exhaustive survey of protocol synthesis methods is given in [25]. The methods listed in [25] differ in the server architecture and in the specification language for which they are intended. In the present paper we propose a protocol synthesis method for servers with multiple components pairwise communicating over reliable unbounded first-in-first-out (FIFO) channels, for a simple specification language inspired by LOTOS [2].

LOTOS is a standard process-algebraic formal language, primarily intended for specification of the external behaviour of processes or of their interaction. Most of protocol synthesis methods for LOTOS or its derivatives [1,3,5,6,8–12,15,17–21,23,24] descend from the attribute-grammars-based [1]. We take [8] as the starting point. For data handling, we follow the ideas of [9], that integrates the concepts of [6] and [8].

[6,9] regard data items as unstructured, while our tool also supports implementation of virtual queues with elements overtaking each other according to their specified priorities [11]. The enhancement has been motivated by the fact that

[1] Email: monika.kapus-kolar@ijs.si

networking applications often require streams of data flowing from one PE to another. A second example (where priorities might be important) is a queue conveying requests from a set of clients to a set of servers.

Another additional feature of our tool is support for user mobility [12]. Service users are often mobile, i.e. they move from one SAP to another and want the server to offer them the service at the SAP to which they are currently attached. An example is a person walking around an picking up a phone here and a phone there. From the point of the service, it is not the physical moving of users or SAPs that matters; rather it is the logical moving of users relative to SAPs.

A detailed specification of the tool is given in [13] (though we continue its development). The tool is written in Prolog, for its algorithm heavily depends on manipulation of symbols. Consequently, when conceived in 1996, the tool was so slow that it could only pass as an interesting experiment. That it probably was, for it seems that no other author of protocol synthesis algorithms addresses the problems of supporting user mobility and distributed virtual queues. As now the available hardware is much faster, the tool seems quite a useful prototyping aid, worth further consideration.

The paper is structured as follows: Section 2 presents the adopted language for service and protocol specification. In Section 3 we introduce the basic ideas of the protocol-derivation process. Specifically, in its Subsection 3.1 we show that user mobility can be supported in a very trivial way. Sections 4 and 5 respectively describe handling of unstructured service parameters and implementation of distributed virtual queues. Section 6 presents calculation of service attributes that guide the protocol-derivation process. Section 7 describes how we cope with the complexity of the process. In Section 8, we compare our approach with similar algorithms and give some plans for further work.

2 Specification language

Automated protocol derivation is based on syntactic manipulations of service specifications. To be able to focus on the numerous semantic aspects of a service and its protocol, we resort to the common practice and work with a very abstract language. Particularly we omit all details of the service that don't influence the protocol, for example the exact nature of service actions and the contents of the data they handle, referring to them only by their names and location.

2.1 Service Specification

A service specification is expected to be structured according to the production rules in Table 1. It defines (1) the required service behaviour e. The input conditions of the service are specified in IL, IP and IQ, and its required output conditions in OP.

The basic constituents of a service behaviour are service primitives. A fixed-location service primitive (2) is identified by its $Name$ and the $Place$ (i.e. the SAP) of its execution. It is an atomic action of the PE supporting the place, potentially executed in co-operation with some service user. Alternatively, a service primitive may be assigned to a particular service $User$ and be executed as a service interaction at the place where the user is currently located (3).

During execution of a service primitive of any of the above types, the involved PE may access some unstructured service parameters, acting as virtual global variables of the server. An unstructured parameter listed in $Ipar$ is an input to the primitive, i.e. must be known to the PE before the primitive is executed. Within the primitive, such a parameter often serves as an output to a service user. An unstructured parameter listed in $Opar$ is an output of the primitive, i.e. the primitive sets it to a new value, that is subsequently available at the PE. The originator of the new value might be either the PE or the service user executing the primitive, or both in co-operation. If an unstructured parameter is listed in $Rpar$, its value doesn't change, but the primitive makes it available to the PE (e.g. through implicit reception or computation). If an unstructured parameter is listed in $Dpar$, the primitive makes the PE delete its local copy, while its global value remains unchanged.

Besides the unstructured parameters, a primitive may access some distributed virtual queues. A specified number of data elements may be removed (as an input to the primitive, specified in $Ique$) from the head, or appended (as an output of the primitive, specified in $Oque$) to the tail of a queue. When elements are appended to a queue, they overtake each other according to their specified priorities.

By a special service primitive (4) a $User$ indicates to the PE to which it is currently attached its moving to a particular $Place$. Such a move must never be concurrent to another service primitive explicitly involving the user. The initial location of individual service users is specified in IL.

Parts of a service may be specified for sequential (5), independent parallel (6), or alternative (7) execution. If $Guard$ in (7) is non-empty, selection between the alternatives is based on the current values of the unstructured service parameters listed in $Guard$. An empty $Guard$ implies that the decision is made upon the initial service primitive of e. In that case we require that all the potential initial primitives belong to the same place, to avoid distributed decision-making [1].

With respect to the service data, a state of the service provider is defined by the current distribution of the knowledge of the unstructured service parameters, and by the contents and its distribution of the virtual queues. IP and IQ respectively describe the initial state of the unstructured parameters and queues, while OP the required minimum final knowledge of the unstructured parameters.

$Ipar$, $Opar$, $Rpar$, $Dpar$ and $Guard$ are lists of parameter names.

IL is a list of elements of the form $User{:}Place{:}Cond$, stating that under the condition $Cond$, $User$ is initially located at $Place$.

Table 1
Service specification language

No.	Name	Definition
1	Service specification	$ServSpec \rightarrow$`SERVICE` $(IL\|IP\|OP\|IQ) = e$ `END`
2	Fixed-location service primitive	$e \rightarrow Name@Place(Ipar\|Opar\|Rpar\|Dpar\|Ique\|Oque)$
3	Service primitive for a particular user	$e \rightarrow Name*User(Ipar\|Opar\|Rpar\|Dpar\|Ique\|Oque)$
4	User move	$e \rightarrow User>Place$
5	Sequential composition	$e \rightarrow e_1;e_2$
6	Parallel composition	$e \rightarrow e_1\|\|\|e_2$
7	Alternative composition	$e \rightarrow e_1[Guard]e_2$

Table 2
An illustrative service specification (Example 1)

```
SERVICE(u1:1,u2:2,u3:3,u4:4|x:2,y:1,y:2|x:1:u|
        q:[4:4|2:1]:v,q:[|]:¬v) =
((a*u1(|x||||q:2:2);u2>4)[y]
 (b@1(|||||q:2:2)|||c*u2));
((d@1;e*u2(x|||||q:3:4);u4>3)[]
 (f@1;(g*u2(x|||||q:1:2)|||h*u3(||x|||q:1:1))));
(i*u1(|||x||q:3:2)|||j@4(||||q:1));
(((k@4(x);l*u4(y||||q:5))[]m@4(|x))||||
 ((n*u4;o@4(y))[]p*u4(y)))
END
```

IP and OP are lists of elements of the form Par:$Place$:$Cond$, stating that under the condition $Cond$, the parameter Par is, respectively, initially known or required to be finally known to the PE at $Place$.

With an element Que:N in the list $Ique$, the PE at the place takes the first N elements from Que.

$Oque$ is a list of elements Que:N:Pri, stating that the PE at the place appends to Que N elements with the priority Pri.

IQ is a list of elements Que:$State$:$Cond$, stating that under the condition $Cond$, the initial state of Que is $State$. $State$ is of the form $[FQ|RQ]$. FQ refers to the front elements of the queue, residing already at the places of their future consumption, while RQ to the rear elements of the queue, still residing at the places which have generated them. Each element of FQ or RQ is of the form $Place$:Pri, stating that the corresponding data item resides at $Place$ and its priority is Pri.

All the $Cond$ conditions are supposed to be functions of special, globally known parameters externally supplied to, but not handled by the server.

Example 1 *The service specification in Table 2 should illustrate a wide range of interesting situations. Its key parts are (like in Tables 3 and 4) emphasised to improve legibility.*

There are places `1` *to* `4`*, initially supporting users* `u1` *to* `u4`*, respectively.* `u2` *might later move to* `4`*, and* `u4` *to* `3`*. There are unstructured parameters* `x` *and* `y`*, queue* `q`*, and auxiliary parameters* `u` *and* `v`*.*

`x` *is initially known to* `2`*, and must under condition* `u` *be finally known to* `1`*. It is regenerated by* `1` *in* `a`*, and by* `4` *in* `m`*, used by* `2` *or* `4` *in* `e` *and* `g`*, and by* `4` *in* `k`*, implicitly received by* `3` *in* `h`*, and forgotten by* `1` *in* `i`*.*

`y` *is initially known to* `1` *and* `2`*, that use it for decision on the first alternative. If* `u4` *doesn't migrate to* `3`*,* `l` *and* `o` *(or* `p`*) at* `4` *use* `y` *concurrently.*

Under condition `v`*,* `q` *initially contains* `1` *element with priority* `4` *in its front part at place* `4`*, and* `1` *element with priority* `1` *in its rear part at place* `2`*. Under condition* ¬`v`*,* `q` *is initially empty. It is written to by* `1` *in* `a` *or* `b` *(*`2` *elements with priority* `2`*), and in* `i` *(*`3` *elements with priority* `2`*), by* `2` *or* `4` *in* `e` *(*`3` *elements with priority* `4`*) or* `g` *(*`1` *element with priority* `2`*), and by* `3` *in* `h` *(*`1` *element with priority* `1`*). Place* `4` *removes from it* `1` *element in* `j`*, and place* `3` *or place* `4` `5` *elements in* `l`*.* `g` *and* `h` *concurrently write to* `q`*, but with different priorities.* `i` *writes to* `q` *while* `j` *concurrently reads its stable front.*

2.2 Protocol Specification

The derived behaviour of individual places is specified basically in the same language as the required service, with some exceptions.

A behaviour specification derived for a $Place$ has four parameters. The first two respectively specify the input and the output knowledge of the unstructured parameters at $Place$. They are lists of elements Par:$Cond$, stating that Par is under the condition $Cond$ known to $Place$. The third and the fourth parameter respectively specify the input and the output state of the local queues that at $Place$ correspond to the global virtual queues. They are lists of elements Que:FN:RN:$Cond$, stating that, under condition $Cond$, FN elements of the front part and RN elements of the rear part of the queue Que reside at $Place$ (see Section 5 for further explanation).

Each alternative composition operator $[Guard]$ is extended into $[Guard|N]$, where N identifies the particular operator.

A $Guard \rightarrow Spec$ is a specification $Spec$ guarded by $Guard$. If $Guard$ is `true`, the term evaluates to $Spec$. If $Guard$ is `false`, the term evaluates to an empty term ε.

- A $Guard \rightarrow Spec$ might stand in a position where a behaviour specification would be expected, i.e. $Spec$ might represent a behaviour. If such a term degenerates into ε, it can be eliminated by the absorption rules given in [6]. An example is a $Spec$ representing a service primitive not intended for execution at the place to which the specification belongs.
- If a $Guard \rightarrow Spec$ belongs to a set, its evaluation to ε is equivalent to deletion of $Spec$ from the set. Examples of such sets are the TO and RG sets introduced in the next two paragraphs, respectively.

  A `trans(`$N|TO$`)` specifies a set TO of the transmission

obligations for a context N. `trans(`N`|)` is equivalent to ε. The elements of TO are of the form $Guard \rightarrow Spec$. $Spec$ may be of the form $Place$, stating that a message containing (possibly among other elements) the context identifier N must be sent to $Place$. Second, $Spec$ may be of the form `<`$Place$`,`Par`>`, stating that a message containing the context identifier N and the value (properly identified) of the unstructured parameter Par must be sent to $Place$. Third, $Spec$ may be of the form `<`$Places$`,`Que`>`, stating that the front elements of the queue Que must be sent to the places listed in $Places$, in that order, properly identified and accompanied with the context identifier N.

Transmission obligations within a single `trans` may be mapped into messages in any manner suitable for a particular system, while mixing of message elements from different contexts must be avoided. Hence a `trans(`$N|TO$`)` is an abstract (concise) representation of a set of guarded protocol message transmissions. Before a PE can execute its behaviour specification, all `trans` expressions in it must be suitably mapped.

A `rec(`$N|RG$`)` specifies the reception goals for a context N. `rec(`N`|)` is equivalent to ε. Guarded reception goals in RG are structured just like transmission obligations, except that they specify places to receive from and combinations of message elements which are to be received. An `any` specifies reception from any place. Mapping a `rec` into a set of guarded message receptions, one must bear in mind how the message elements have been combined into messages in the implementation of the `trans` for the particular context. Before a PE can execute its behaviour specification, all `rec` expressions in it must be suitably mapped.

A `set(`Var`,`Val`)` sets the local copy of the auxiliary variable Var to the value Val.

For illustration, we present in Table 3 a protocol specification derived by our tool for the service described in Example 1. At that point, the reader should just briefly glance over it, to get familiar with the protocol specification language.

Places are expected to interpret their behaviour specifications in the following way: Whenever a place sets the value of a variable, it immediately propagates it into the guards of its future behaviour, deletes the guards which already evaluate to `true`, and eliminates the ε terms resulting from the guards evaluating to `false`. Subsequently, the place continues service execution according to the newly adapted behaviour specification. Care is taken that places are supplied with sufficient knowledge for guard evaluation before the guarded actions become pending for execution.

Example 2 *Table 4 shows the state of the service and the corresponding PE specifications from Tables 2 and 3 after the external service parameters* `u` *and* `v` *and the auxiliary variables* `alt(1)` *and* `alt(2)` *(i.e. their local copies) have been set to* `true`*. In the adapted service specification we see that the first two of the four consecutive service parts have already been executed. In the place specifications we observe that, with the knowledge obtained so far, the places have been able to evaluate the guards of all their immediately executable actions.*

Table 4
Situation after a partial run of the server from Tables 2 and 3.

```
SERVICE(u1:1,u2:4,u3:3,u4:3|
        x:1,x:2,x:4,y:1,y:2,y:3|x:1|
        q:[4:4|4:4,4:4,4:4,4:4,1:2,1:2,2:1]) =
(i*u1(|||x||q:3:2)|||j@4(||||q:1));
(((k@4(x);l*u4(y||||q:5))[]m@4(|x))|||
 ((n*u4;o@4(y))[]p*u4(y)))
END

ENTITY_1(x,y|y,x|q:0:2|q:0:3:alt(3),q:0:5:¬alt(3)) =
rec(24|3,4);i*u1(|||x||q:3:2);trans(22|4,3);
(((rec(13|any);trans(14|<[3,3],q>))[|3]rec(16|any))|||
 ((rec(17|,any);trans(18|<4,y>))[|4]rec(20,any)));
rec(27|<any,x>)
END

ENTITY_2(x,y|y|q:0:1|q:0:1) =
(rec(13|any)[|3]rec(16|any))
END

ENTITY_3(y|y|q:0:0|q:0:0) =
trans(24|1,4);rec(22|1,4);
(((rec(14|4,<[4,4,4,1,1],q>);l*u4(y||||q:5))[|3]
  rec(16|any))|||
 ((n*u4;trans(17|1);trans(18|4))[|4]
  (p*u4(y);trans(20|1,4))))
END

ENTITY_4(x|x,y:alt(4)|q:1:3|
         q:0:0:alt(3),q:0:3:¬alt(3)) =
trans(24|1);rec(24|3);
j@4(||||q:1);trans(22|3);rec(22|1);
(((k@4(x);trans(13|1,2);trans(14|3,<[3,3,3],q>))[|3]
  (m@4(|x);trans(16|1,2,3)))|||
 ((rec(18|3,<any,y>);o@4(y))[|4]rec(20|any)));
trans(27|<1,x>))
END
```

Table 5
A brief presentation of the protocol derivation mapping

No.	Mapping rule
1	$\mathbf{T}_p(ServSpec) :=$ `ENTITY_`$p(IP(p)\|OP(p)\|IQ(p)\|OQ(p))$ $= \mathbf{T}_p((e;\varepsilon))$ `END`
2	$\mathbf{T}_p(e) := ((p = Place) \rightarrow e)$
3,4	$\mathbf{T}_p(e) := (att\_b(User, p, e) \rightarrow e)$
5	$\mathbf{T}_p(e) := (\mathbf{T'}_p(e_1);$`trans(`$N|\ldots$`)`$;$`rec(`$N|\ldots$`)`$;\mathbf{T}_p(e_2))$
6	$\mathbf{T}_p(e) := (\mathbf{T}_p(e_1)\|\|\|\mathbf{T}_p(e_2))$
7	$\mathbf{T}_p(e) :=$ $((\mathbf{T'}_p(e_1);(\ldots \rightarrow$ `set(alt(`N`),true)`$))[Guard|N]$ $(\mathbf{T'}_p(e_2);(\ldots \rightarrow$ `set(alt(`N`),false)`$)))$
where	$\mathbf{T}_p(\varepsilon) := \varepsilon$, $\mathbf{T'}_p(e) :=$ $(\mathbf{T}_p(e);$`trans(`$N|\ldots$`)`$;$`rec(`$N|\ldots$`)`$)$

3 Basic Ideas of the Protocol-Derivation Process

Behaviour specification for a place p is derived from a service specification by the mapping $\mathbf{T}$ briefly presented in Table 5 (where the first column identifies the production rule(s) from Table 1 to which the particular row applies), followed by simplification that always includes initial evaluation of guards and elimination of ε terms. As in earlier similar algorithms, mapping of a service specification proceeds in a compositional way, by mapping its constituent parts (subexpressions, service subbehaviours). Protocol derivation is guided

Table 3
A protocol specification derived for the service described in Example 1

```
ENTITY_1(y|y,x:u|q:0:0|q:0:0:(alt(3)∧¬v∧¬alt(2)),
         q:0:1:(v∧alt(3)∧¬alt(2)),
         q:0:2:(alt(2)∧alt(3)∧¬v),
         q:0:3:(v∧alt(2)∧alt(3)),
         q:0:4:(¬v∧¬alt(2)∧¬alt(3)),
         q:0:5:((v∧¬alt(3))∨(alt(2)∧¬alt(3)))) =
(((a*u1(|x|||q:2:2);trans(2|2,<4,x>);trans(3|3);
   set(alt(1),true))[y|1]
  (b@1(|||||q:2:2);trans(4|3,4);set(alt(1),false)));
 rec(26|2);
 ((d@1;trans(8|<3,y>,(alt(1)→4),(¬alt(1)→2));
   trans(11|2);set(alt(2),true))[|2]
  (f@1;trans(10|3,<4,y>,(alt(1)→4),(¬alt(1)→2));
   trans(12|2);set(alt(2),false)));
 trans(24|((¬v∧¬alt(2))→<[4],q>));
 rec(24|3,4,(¬alt(1)→2));
 i*u1(|||x||q:3:2);trans(22|4,(alt(2)→3));
 (((rec(13|any);
    trans(14|((v∧alt(2))→<[3,3],q>),
             ((alt(2)∧¬v)→<[3,3,3],q>),
             (¬alt(2)→<[4,4,4,4],q>)))[|3]
   rec(16|any))|||
  ((rec(17|,any);trans(18|(alt(2)→<4,y>)))[|4]
   rec(20,any)));
 rec(27|(u→<any,x>)))
END

ENTITY_2(x,y|y,x:(alt(3)∧¬alt(1))|
         q:0:0:¬v,q:0:1:v|
         q:0:0:((alt(1)∧¬v)∨(alt(3)∧¬v)),
         q:0:1:((v∧alt(1))∨(v∧alt(3))∨
                (¬v∧¬alt(1)∧¬alt(2)∧¬alt(3))),
         q:0:2:((v∧¬alt(1)∧¬alt(2)∧¬alt(3))∨
                (alt(2)∧¬v∧¬alt(1)∧¬alt(3))),
         q:0:4:(v∧alt(2)∧¬alt(1)∧¬alt(3))) =
(((rec(2|1);u2>4;set(alt(1),true))[y|1]
  (c*u2;trans(4|3,4);set(alt(1),false)));
 trans(26|1,(¬alt(1)→<4,x>));
 ((rec(8|(¬alt(1)→1));(¬alt(1)→e*u2(x||||q:3:4));
   trans(6|(¬alt(1)→4));rec(11|any);
   set(alt(2),true))[|2]
  (rec(10|(¬alt(1)→1));
   (¬alt(1)→g*u2(x||||q:1:2));
   rec(12|any);set(alt(2),false)));
 trans(24|((alt(2)∧¬v∧¬alt(1))→<[4],q>),
          (¬alt(1)→1),(¬alt(1)→4));
 ((rec(13|any);
   trans(14|((v∧alt(2)∧¬alt(1))→<[3,3,3],q>),
            ((alt(2)∧¬v∧¬alt(1))→<[3,3],q>),
            ((¬alt(1)∧¬alt(2))→<[4],q>)))[|3]
  rec(16|any)))
END
```

```
ENTITY_3(|x:(alt(3)∧¬alt(2)),y:alt(2)|q:0:0|
         q:0:0:alt(2),q:0:1:¬alt(2)) =
(((rec(3|any);set(alt(1),true))[y|1]
  (rec(4|any);set(alt(1),false)));
 ((rec(8|<any,y>);set(alt(2),true))[|2]
  (rec(10|1);h*u3(||x|||q:1:1);set(alt(2),false)));
 trans(24|1,4);rec(22|(alt(2)→1),(alt(2)→4));
 (((rec(14|(alt(2)→4),
           ((v∧alt(1)∧alt(2))→<[4,4,4,1,1],q>),
           ((v∧alt(2)∧¬alt(1))→<[2,2,2,1,1],q>),
           ((alt(1)∧alt(2)∧¬v)→<[4,4,1,1,1],q>),
           ((alt(2)∧¬v∧¬alt(1))→<[2,2,1,1,1],q>));
   (alt(2)→l*u4(y||||q:5)))[|3]
  rec(16|any))|||
 (((alt(2)→n*u4);trans(17|(alt(2)→1));
   trans(18|(alt(2)→4)))[|4]
  ((alt(2)→p*u4(y));
   trans(20|(alt(2)→1),(alt(2)→4))))))
END

ENTITY_4(|x,y:(alt(4)∨¬alt(2))|q:0:0:¬v,q:1:0:v|
         q:0:0:(alt(3)∨¬alt(1)),
         q:0:1:(alt(1)∧¬alt(2)∧¬alt(3)),
         q:0:2:(alt(1)∧alt(2)∧¬v∧¬alt(3)),
         q:0:3:(v∧alt(1)∧alt(2)∧¬alt(3))) =
(((rec(2|<any,x>);set(alt(1),true))[y|1]
  (rec(4|any);set(alt(1),false)));
 rec(26|(¬alt(1)→<any,x>));
 ((rec(8|(alt(1)→1));(alt(1)→e*u2(x||||q:3:4));
   rec(6|(¬alt(1)→2));u4>3;set(alt(2),true))[|2]
  (rec(10|<any,y>,(alt(1)→1));
   (alt(1)→g*u2(x||||q:1:2));
   set(alt(2),false)));
 trans(24|1,((alt(1)∧alt(2)∧¬v)→<[4],q>));
 rec(24|3,((alt(1)∧alt(2)∧¬v)→<[4],q>),
          ((alt(2)∧¬v∧¬alt(1))→<[2],q>),
          ((¬v∧¬alt(2))→<[1],q>),(¬alt(1)→2));
 j@4(||||q:1);trans(22|(alt(2)→3));rec(22|1);
 (((k@4(x);trans(13|1,2);
    trans(14|(alt(2)→3),
             ((v∧alt(1)∧alt(2))→<[3,3,3],q>),
             ((alt(1)∧alt(2)∧¬v)→<[3,3],q>),
             ((alt(1)∧¬alt(2))→<[4],q>));
    rec(14|((v∧alt(1)∧¬alt(2))→<[1,1,4,1,1],q>),
           ((v∧¬alt(1)∧¬alt(2))→<[1,1,2,1,1],q>),
           ((alt(1)∧¬v∧¬alt(2))→<[1,4,1,1,1],q>),
           ((¬v∧¬alt(1)∧¬alt(2))→<[1,2,1,1,1],q>));
    (¬alt(2)→l*u4(y||||q:5)))[|3]
   (m@4(|x);trans(16|1,2,3)))|||
  (((¬alt(2)→n*u4);trans(17|(¬alt(2)→1));
    rec(18|(alt(2)→3),(alt(2)→<any,y>));o@4(y))[|4]
   ((¬alt(2)→p*u4(y));rec(20|(alt(2)→any)))));
 trans(27|(u→<1,x>)))
END
```

by various pre-calculated attributes of the parts (more details in Section 6).

3.1 Supporting User Mobility

Though it might seem strange, we shall start by explaining how user mobility is supported, so that we can stop worrying about it.

User mobility can be implemented simply by properly guarding protocol actions by variables recording past decisions in the service execution. In Section 3.5 we explain how places are timely supplied with the knowledge necessary for evaluation of action guards. As by the time that actions become pending, their guards are completely evaluated, we may in the subsequent discussion forget about the guards, i.e. about mobility.

Example 3 *Let's return to the Example 2. The auxiliary variable* `alt(2)` *has been set to* `true` *because the first alternative has been selected upon the second choice in the service. That means that user* `u4` *has moved from place* `4` *to place* `3`*. In Table 3 we see that both places record the value of* `alt(2)`*. So in the future they know, for example, that* `l*u4(...)` *must be executed at place* `3`*, not at place* `4` *(see Table 4).*

3.2 Inter-Place Communication

As the first step towards protocol correctness, we ensure that every protocol message sent is actually received, by following the strategy of [8].

Unlike to [6], protocol messages serving for data exchange and those serving for inter-place synchronisation are treated in a uniform way, i.e. using `trans` procedures for their combined transmission and `rec` procedures for their combined reception. Any message so exchanged serves for synchronisation of its sender and its recipient, while messages with elements additional to the context identifier N also serve for data exchange.

The structure of a service specification reveals the points in the service execution where protocol message exchanges might be necessary. In Table 5 we see that the points are situated between consecutive service parts, i.e. upon sequential composition operators, and at the ends of the service parts mapped by $\mathbf{T}'$ instead of $\mathbf{T}$. Every point represents an individual message exchange context, i.e. a message exchange procedure (MEP) consisting of the message transmissions and their corresponding receptions pertaining to the context.

The derived behaviour of any place p has basically the same structure as the service augmented with identifiers of its MEPs. Every service primitive or MEP has its image at p and the operators combining subbehaviours are the same as in the service.

Behaviour composition operators specify the partial order in which actions are executed. By having message receptions always specified in the same points as the corresponding transmissions, we ensure that the partial order specified for the receptions is the same as for the transmissions. In other words, the recipients never require that messages are received in an order different from that in which they have been sent. Hence if places properly progress through the service, they will also properly receive the protocol messages sent.

Example 4 *The above concepts are illustrated in Table 6, where* e *is a small service behaviour, that is in* e' *augmented with identifiers of its MEPs. The subexpressions of* e *and the MEPs are then mapped onto individual places* `1` *and* `2`*, and the two local specifications subsequently simplified.*

Proper progress of the distributed system is ensured by proper implementation of each individual service primitive and each individual composition operator, as discussed in the following subsections. Individual service behaviour types are mapped very much like in [8].

Table 6
Illustration to Example 4

e = `(((a@1;b@2);c@1)|||((d@2;e@1)[]f@2))`

e' = `(((((a@1;`MEP_1`);`MEP_2`;b@2);`MEP_3`);`MEP_4`;c@1)|||`
`((((d@2;`MEP_5`);`MEP_6`;e@1);`MEP_7`)[](f@2;`MEP_8`)))`

$\mathbf{T_1}(e) \approx$ `(((((a@1;`ε`;`ε`);trans(2|2);`ε`;`ε`);`ε`;`ε`);`
`  `ε`;rec(4|2);c@1)|||`
`  ((((`ε`;`ε`;`ε`);`ε`;rec(6|2);e@1);`ε`;`ε`)[]`
`  (`ε`;`ε`;rec(8|2))))`
$\approx$ `((a@1;trans(2|2);rec(4|2);c@1)|||`
`((rec(6|2);e@1)[]rec(8|2)))`

$\mathbf{T_2}(e) \approx$ `(((((`ε`;`ε`);`ε`;rec(2|1);b@2);`ε`;`ε`);`
`  trans(4|1);`ε`;`ε`)|||`
`  ((((d@2;`ε`;`ε`);trans(6|1);`ε`;`ε`);`ε`;`ε`)[]`
`  (f@2;trans(8|1);`ε`)))`
$\approx$ `((rec(2|1);b@2;trans(4|1))|||`
`((d@2;trans(6|1))[](f@2;trans(8|1))))`

3.3 Implementation of Individual Service Primitives

An individual service primitive (see Table 5(2–4)) is implemented at any place p as it is, but preceded by a guard defining the conditions under which it is to be executed at p. For a fixed location primitive (2), the guard is `true` for the place to which it belongs, and `false` otherwise [8]. If it is executed for a mobile $User$ (3,4), its execution is at any place guarded by its dynamically evaluated attribute $att\_b(User, p, e)$, that specifies whether $User$ is just before executing the service part e attached to p.

A service primitive is executed by an individual place p, and its distributed implementation introduces no protocol messages, i.e. its implementation is trivially correct. However, if it is to be executable, the other places must ensure that its input parameters are available at p in time. How this is achieved, we explain in Sections 4 and 5.

3.4 Implementation of Parallel Composition

Implementation of parallel composition (6) requires no additional inter-place synchronisation. Its correctness is based on the cross-cut theorem [4] on re-grouping of parallel processes. For a pair of e_1 and e_2 running in parallel, there are processes $\mathbf{T}_p(e_1)$ and $\mathbf{T}_p(e_2)$ for each place p. We have the pool of processes organised as a group of local processes $(\mathbf{T}_p(e_1)|||\mathbf{T}_p(e_2))$ communicating over the protocol channels, and we would like to show that that is from the point of the service equivalent to the group of the $\mathbf{T}_p(e_1)$ processes (the distributed implementation of e_1) running in parallel to the group of the $\mathbf{T}_p(e_2)$ processes (the distributed implementation of e_2).

The problem is that in the second case, the two groups use for intra-group communication the same set of protocol channels and might thus potentially interfere on them [16]. That is because the channels support asynchronous communication, and not synchronous like in [4]. Fortunately, in [14] we have been able to show that there is no interference on FIFO protocol channels, if the parallel composition operator introduces no synchronisation between e_1 and e_2 and if

places are always willing to receive protocol messages in the order in which they have been sent (see Section 3.2).

Example 5 *In the example in Table 6, the two parallel service parts both use the protocol channel leading from place* `2` *to place* `1`*, but the reader can check that the messages issued on the channel concurrently are nevertheless properly received.*

3.5 Implementation of Choice

Like in [8], we require that an e of the form $(e_1[Guard]e_2)$ (7) is such that distributed implementation of the choice requires no distributed decision-making, i.e. no additional protocol messages.

In [8], *Guard* is always empty, i.e. there are no service parameters whose value would resolve the choice in advance. In that case it is required that all the starting service primitives of e_1 and e_2 belong to the same place. The choice is made locally at the place upon the first primitive of the selected alternative, while other places enter the alternative upon messages received within the implementation of the alternative.

Example 6 *In the example given in Table 6, place* `2` *makes the choice and invites into the selected alternative place* `1`*. Note that the invitation is also sent in the second alternative, although place* `1` *has no service primitive specified in it. That is to prevent place* `1` *from permanent waiting for a potential activation of the first alternative [8].*

If *Guard* is not empty, the choice is implied by the values of the service parameters listed in it. In that case we make sure that the executors of the starting service primitives of the alternatives (their starting places) know the values before execution of e starts. Hence again places enter one of the alternatives in a co-ordinated manner.

Example 7 *In the example in Table 2, the first choice is based on the value of service parameter* `y`*. The places that consequently must (and do) know* `y` *initially are place* `1` *(as the executor of* `a*u1` *and* `b@1`*) and place* `2` *(as the executor of* `c*u2`*).*

In Example 6 we have met a place originally participating in only one of the alternatives, while the missing invitation to the other alternative e_2 has been introduced by using $\mathbf{T}'_p(e_2)$ instead of $\mathbf{T}_p(e_2)$. In such manner, a place always participates in both alternatives or in none. Thereby we ensure that the participants are properly co-ordinated not only upon entering e, but also upon terminating it.

By participating in the implementation of e, a place implicitly receives information on the selected alternative. As explained in Section 3.1, such information might be necessary for supporting user mobility. It might also help a place to determine the current distribution of the knowledge of a service parameter.

Example 8 *Suppose that a place p forgets a parameter x in one alternative, but not in the other. If another place p' must afterwards supply x to p, p' should know the decision upon the choice, to avoid the transmission in the case that p still knows x.*

Initially, each choice operator (identified by an N) in the service specification is assigned a virtual global Boolean variable `alt(`N`)`, whose value is set when the decision is made and indicates whether the first of the two alternatives has been selected. Whenever a place p detects an interesting decision, it makes a local copy of the associated variable. p sets the variable upon terminating the selected alternative. The `set` command is guarded, where the guard indicates whether p might need the variable for evaluation of guards of its future actions. If a p originally participates in none of a pair of alternatives, but must detect the choice, care is taken that an invitation is sent to p at the end of each individual alternative.

Example 9 *For the service in Table 2, Table 3 shows that the identifier of the first choice operator is* `1`*, i.e. the decision recorded in* `alt(1)`*. Place* `3` *originally participates in none of the alternatives, but* `alt(1)` *is present in its future guards. Therefore place* `3` *is informed on the selected alternative (by* `rec(3|any)` *or* `rec(4|any)`*, respectively) and sets its copy of the variable.*

In [8], the place informing other places that a particular alternative has just been completed is always the unique starting place of the alternative. If *Guard* is non-empty, there might be several starting places. Fortunately, the exact identity of the sender is irrelevant for the protocol correctness and we are free to chose any of the starting places. If there are several such places, we make sure that each of them has at that point of service execution sufficient knowledge to determine on its own whether to send the message or not.

[8] also requires that alternatives e_1 and e_2 have equal ending places, i.e. places executing their ending actions, but the requirement is superfluous [17].

3.6 Implementation of Sequential Composition

Executing its implementation of an $(e_1;e_2)$, a place first executes its e_1 part, possibly concluded by some additional protocol message exchanges upon its completion, that are introduced by mapping $\mathbf{T}'$. Namely, if there is a place p expected to transmit some service parameter values upon the subsequent sequential composition operator, but not yet knowing that service execution is progressing towards the MEP, the transmission is stimulated by a message sent to p within the MEP at the end of e_1. The sender is selected in the same manner as for the messages sent upon termination of an alternative within a choice (see Section 3.5).

Example 10 *In Table 2, consider the service subexpression* `(k@4(...);l*u4(...))`*, an alternative of the third choice subexpression in the service. Place* `1` *doesn't participate in its service primitives, but is expected to transmit some elements of queue* `q` *upon the sequential composition operator (see* `trans(14|...)` *in Table 3). To initiate the transmission, place* `1` *must previously receive an indication that the particular alternative has been selected (specified by* `rec(13|any)`*).*

After their e_1 parts, places execute the MEP belonging to the sequential composition operator, before proceeding to their e_2 parts. The messages sent upon the operator serve primarily to prevent a premature start of e_2. For that purpose it is in principle necessary that the ending places of e_1 transmit to the starting places of e_2 [18,8]. However, if such a message has already been sent at the end of the implementation of e_1, it may be omitted from the MEP belonging to the sequential composition operator, for the recipient already knows that the sender has completed its service primitives in e_1. Another sort of messages sent upon the operator are messages carrying service parameters which have the exchange scheduled for the particular point.

In [8], a MEP belonging to a sequential composition operator is never preceded by a MEP upon the end of e_1, but we may pretend that both MEPs are actually a single MEP consisting of two consecutive parts.

Anyhow, there might be more and longer messages sent upon a sequential composition operator than in [8], but that is not problematic. For observe that for a proper reception of the messages it is sufficient that their senders enter execution of the $(e_1; e_2)$ before the transmission, and they do.

In Table 5(1) we see that there is an additional sequential composition operator introduced just after the e specifying the overall service behaviour. Where necessary, its serves for a final dispatching of the service parameters required to be known at the recipient places upon the service completion (the OP part of a service specification).

3.7 A Statement on Protocol Correctness

The correctness of a protocol derived for a service has two aspects.

- The control aspect secures that exactly the specified sequences of service actions are implemented, that the service users can select the available alternatives just if the server was not distributed, and that all the protocol messages sent are properly received.
- The data aspect secures that the protocol messages timely deliver service data to their recipients.

From the aspect of control, the protocols we derive are structured like in [8]. To be exact, we have so far in Section 3 identified some differences, but also explained that they are irrelevant for protocol correctness. Hence if [8] is correct from that aspect, so is our algorithm. The correctness of [8] has been proven in [7]. Later some flaws have been identified [16], but not in the implementation of the behaviour composition operators that we allow in service specifications. Thus we conclude that the protocols we derive are correct from the control aspect, and explain the handling of data in the following sections.

4 Handling of Unstructured Service Parameters

As already told, those parameters are virtual global variables that places concurrently access. The only restrictions that we pose on their use within a service are the natural ones. It must never be specified that a place uses a parameter value that is currently globally unknown or unstable, or is just being deleted at the place.

[9], the ideas of which we follow, gives a designer the freedom to choose the point of execution when an unstructured parameter is delivered to a PE needing it. Our algorithm sets the parameter distribution scheme automatically. In principle a parameter is transmitted to a place as soon as there is no doubt that it will be needed there, either for execution of a service primitive (Section 3.3) or to select the right service alternative upon a choice with a non-empty guard (Section 3.5). If a parameter is being supplied for execution of a service primitive for a mobile user, it might take long, before the executor of the primitive, i.e. the recipient of the parameter, is determined.

In some cases, however, a parameter exchange is speeded up into a MEP where it is not yet completely certain if the recipient will need the value. First, a parameter needed by a decision-making initial service primitive of an alternative is delivered before the choice expression is entered, to avoid changing the original form of decision-making from external to internal. Second, delivery of a parameter might be speeded up to avoid its concurrent duplicate transmission within service parts using the parameter concurrently.

Example 11 *Let's have a look at the primitive* `p*u4(y)` *in Table 2. Its input parameter* `y` *is from the beginning known to places* `1` *and* `2`*, and not forgotten by any of them, while the executor of the primitive is place* `3` *or* `4`*. Just in front of it, there is no sequential composition operator upon which the parameter could be delivered. Moreover, a message exchange at that point would resolve the choice for which* `p*u4(y)` *is an initial primitive, i.e. the decision-making action would be an internal action of the distributed server, while originally the decision-making action is the primitive executed with the consent of the user* `u4`*. Thus the parameter must be delivered before the choice expression* `((n*u4;o@4(y))[]p*u4(y))`*. In the particular case, that is convenient also because* `y` *is needed at the recipient place in both alternatives.*

In parallel to the choice expression, there is the expression `((k@4(x);l*u4(y...))[]m@4(|x))`*, that also needs* `y` *at the recipient place. To prevent duplicate transmission of* `y`*, we move its exchange before the start of the parallel service parts.*

Now which is the earliest convenient point for transmitting `y`*? It is just after* `d@1` *or just after* `f@1`*, because there we have a sequential composition operator and we already know that the executor of* `p*u4(y)` *and* `l*u4(y...)` *will be place* `3` *or place* `4`*, respectively. For the transmitter, we have decided that it should be place* `1` *(see Table 3).*

In [9], the current value of an unstructured parameter x is always distributed by its generating place. Since we allow

that the generator forgets such an x before distributing it, our tool allows x to be sent to a particular destination by any place knowing it at the MEP for which the transmission has been scheduled. If there are several such places, their guards for the transmission are conceived such that 1) x is sent by exactly one of the places, and 2) the number of the `alt` variables addressed in the guards is minimised.

Example 12 *Suppose that a place* `1` *always knows the value of a parameter x intended for transmission in the particular MEP, while places* `2` *and* `3` *know it only under conditions $Cond$ and $\neg Cond$, respectively. We decide that the transmitter should be place* `1`*, for otherwise the other two places would have to keep track of $Cond$.*

5 Implementation of Distributed Virtual Queues

Virtual queues are distributed over queues residing at individual places. Each place maintains one such queue per virtual queue. In its front part, elements generated by the place are stored and wait for transmission to their consumers. In its rear part, elements wait for consumption by service primitives at the place. So the front part of a virtual queue is distributed over the front parts of the corresponding local queues, and its rear part over their rear parts.

Upon each sequential composition operator, some initial elements from the rear part of a virtual queue might be moved to the tail of its front part. That is performed by the places storing the elements that are being moved, transmitting them to their future consumers, that by the time must be completely known. If the structure of the service specification is not correct, that is not the case and the protocol derivation fails. Note that a place might need to "transmit" to itself, if it is both the producer and the consumer of a queue element.

Example 13 *Let's return to the service from Example 1, to the particular service run addressed in Example 2. Initially, the state of the virtual queue* `q` *is* `[4:4|2:1]`*, i.e. there is a priority* `4` *element residing in its front part at place* `4`*, and a priority* `1` *element residing in its rear part at place* `2`*. The initial state of the corresponding local queue at place* `2` *is* `[|1]`*, i.e. there are no elements in its front part and a priority* `1` *element in its rear part. For place* `4`*, the state is* `[4|]`*, while for places* `1` *and* `3` *it is* `[|]`.

`a*u1`*, executed at place* `1`*, appends to* `q` `2` *elements with priority* `2`*. That changes the state of* `q` *into* `[4:4|1:2,1:2,2:1]`*, i.e. the new elements overtake the priority* `1` *element in its rear part. The state of the corresponding local queue at place* `1` *changes into* `[|2,2]`.

The next action on `q` *is* `e*u2`*, executed at place* `4`*, to which user* `u2` *has moved. The action appends to* `q` `3` *elements with priority* `4`*. That changes the state of* `q` *into* `[4:4|4:4,4:4,4:4,1:2,1:2,2:1]`*, i.e. the new elements overtake the priority* `1` *and priority* `2` *elements in its rear part. The state of the corresponding local queue at place* `4` *changes into* `[4|4,4,4]`*. That state of* `q` *is documented as its initial state in Table 4.*

`i*u1` *at place* `1` *appends to* `q` `3` *elements with priority* `2`*, while* `j@4` *concurrently removes* `1` *element from its stable front part. The resulting state of* `q` *is*

`[|4:4,4:4,4:4,1:2,1:2,1:2,1:2,1:2,2:1]`.

The states of the corresponding local queues at places `1` *and* `4` *change to* `[|2,2,2,2,2]` *and* `[|4,4,4]`*, respectively.*

In the following, let's assume that the first alternatives are selected upon the last two choices of the service. Hence `l*u4` *will remove* `5` *elements from the head of* `q`*. For that purpose,* `trans(14|...)` *actions send the elements to place* `3`*, the executor of* `l*u4`*. Place* `1` *sends two elements and place* `4` *three elements. Place* `3` *receives them in* `rec(14|...)`*, and properly puts the elements from place* `4` *in front of the elements from place* `1`*, as originally in* `q`*. The resulting state of* `q` *is* `[3:4,3:4,3:4,3:2,3:2|1:2,1:2,1:2,2:1]`*. The states of the corresponding local queues at places* `1`*,* `4` *and* `3` *change into* `[|2,2,2]`*,* `[|]` *and* `[4,4,4,2,2|]`*, respectively.*

After `l*u4`*, the state of* `q` *is* `[|1:2,1:2,1:2,2:1]`*, while its part at place* `3` *is* `[|]`.

A service specification must respect some additional rules regarding the virtual queues access. Concurrent reads from a queue are forbidden, for otherwise it would not be possible to determine which should be the service primitive using a particular queue element. Concurrent writes on a queue are allowed as long as they are with different priorities, for otherwise it would not be possible to determine the order in which places append the elements to the queue. Reading on an empty queue is not possible. Likewise it is forbidden to read on a queue while there is a pending concurrent higher-priority write on it, for the element being appended would be overtaking the element being read.

Hence it is only possible to read on the stable initial part of a virtual queue. The front part of a queue is stable by definition, while for the purpose of transmitting, elements may be removed only from the stable initial part of the rear part of the queue. The more a transmission of queue elements is delayed, the longer is the stable part of the rear part of the queue and the higher the probability that the required number of stable elements actually exists. Therefore the adopted policy is to delay transmission of queue elements till the last sequential composition operator preceding the service primitive that will use the elements.

6 Calculation of Service Attributes

Calculation of service-specification-subexpression attributes is the crucial and the difficult part of the protocol derivation process. An attribute typically describes a property of an expression or some aspect of the service provider's state (typically some property of the past history of events or of the possible futures) just before or just after execution of the expression. Since our algorithm automates implementation of decision making, attributes are functions of the variables that record decisions on alternatives.

The first task is to compute attribute *att_b* (introduced in Section 3.3) on which nearly all other attributes depend. It is computable if all user moves are legal. For every point in the service, it is also necessary to make distinction between the past (already known) decisions on alternatives, and the future (yet unknown) decisions. That is necessary to be able to effectively cut the cyclic dependences between attributes caused by the presence of user mobility. An example of such a cycle can be found in [12]. The problem lies in the fact that any precise communication planning requires consideration of the future needs of individual places, while those needs in the presence of user mobility often depend on past actions depending on the plan.

The second group of attributes is used for basic checks of the service specification structure. Alternative expressions are checked for adequacy of their starting places and concurrent expressions are checked for compatibility of their parameter access. One searches for the starting and the ending places of expressions, for the places using, generating, or forgetting unstructured parameters, and for places reading or writing on virtual queues.

The third group establishes places' needs for individual service parameters. First, cases where a place obtains an unstructured-parameter value without explicitly receiving it are identified. Then for each point in the service one establishes which places will in the future need the current values of the unstructured parameters and the current elements of the virtual queues.

Those needs imply parameter receptions, that further imply unstructured-parameter knowledge and virtual queues distribution at each individual point of service execution. The attributes are calculated in the next step.

Based on the knowledge-distribution attributes, parameter transmissions are planned and parameter transmitters selected. Those attributes facilitate forming of proper transmission obligations and reception goals upon sequential composition operators.

Service primitives, transmission obligations and reception goals are guarded by functions of past decisions, i.e. the `alt` variables. So one can calculate for each alternative, which places must be informed of the decision, i.e. who will transmit, receive or record the information. The calculations for the `alt` variables are basically the same as for the ordinary unstructured service parameters, except that they are transmitted wherever that might be (not definitely is) necessary for the future. That policy has been chosen because their transmission is trivial, implemented by the final protocol interactions introduced by the mapping $\mathbf{T}'$. As evident from the applications of $\mathbf{T}'$ in Table 5, such auxiliary protocol interactions not only inform on the selected alternatives, but in some cases apply also to the first elements of sequentially composed pairs, where a place might have to poll a peer for a parameter transmission. We could have introduce the auxiliary protocol interactions more selectively, but then deciding on when to transmit would sometimes imply additional information needs, thus inducing additional transmissions of `alt` variables ... and so on in a cascade. Obviously, detailed optimisation of that protocol aspect is not trivial.

7 Coping with Complexity

Implementing the tool, we soon realised that precise planning of data exchanges, particularly for virtual queue elements, made the protocol derivation process very complex. Not only that it was very time-consuming - the main problem was that even tiny service specifications made the tool run out of memory. The reason was that precise attribute calculation was not compositional with respect to the service specification structure, as one might speculate from the compositionality of the mapping $\mathbf{T}$. That is because forward chaining keeps alternating with backward chaining, the forward chaining propagating into the future the effects of the already executed actions, while the backward chaining propagating from the future the requirements for proper action planning. As we didn't want to trade the quality of the derived protocols for compositionality of the derivation process, we decided to make the tool rely more on external memory devices.

The effect of an expression on the knowledge distribution of an unstructured parameter is a simple logical function of parameter accesses during the expression, i.e. of some simple attributes of the expression, just as the distribution itself can be described by a finite set of Boolean variables. On the other hand, the effect of an expression on the state of a virtual queue is in general a complicated non-linear function of the queue state before the expression and there is potentially an unlimited number of possible inputs to the function, hence it is not feasible to pre-calculate (as an auxiliary attribute) the result of the function for each possible input. This is the "bottle-neck" of the tool, the only case during attribute evaluation where an auxiliary property of an expression must be calculated on the spot - not just by referring to the expression, but by thoroughly examining its structure. The bottle-neck is problematic for virtual queues that tend to become long or that are accessed very unpredictably.

In all other cases, evaluation of an expression attribute is purely flat (the attribute depends solely on some other pre-calculated attributes of the expression), purely synthesised (the attribute depends on the attribute and on some other pre-calculated attributes of the subexpressions) or purely inherited (the attribute depends on the attribute of the superexpression and some pre-calculated attributes of the sibling expressions). So an attribute can typically be calculated by traversing the tree that models the service specification structure in a bottom-up or a top-down fashion and in each step considering just the attributes of a node and its sons. Consequently, it is possible to store the pre-calculated attributes on files and read them on the fly, as necessary for each node or group of sibling nodes encountered, in the forward or in the reverse direction. Cumulative attributes can

be calculated iteratively, so that the number of the files which need to be simultaneously open is bounded.

The above tricks are very important, since for exact calculation it is necessary to store a number of attributes that is at least proportional to the number of the service subexpressions times the square of the number of the co-operating places times the number of the service parameters - not to speak of the queues! To illustrate the complexity of the task, let us mention that there are currently 64 different attribute types, many of them inducing two- or three-dimensional arrays of attributes for each service subexpression.

As for the temporal complexity, workstations have already reached a speed that is quite acceptable. The example presented in Tables 2 and 3 took about half a minute CPU time with SICStus Prolog running under Solaris on Sun Ultra 30 - certainly far less than it would take to a human. One should also add the time necessary for translating from an ordinary specification language to the adopted specification language, and vice versa, but that should be a marginal problem if compared to the complexity of the protocol derivation algorithm.

A more real-life-sized service specification could be mapped into a protocol part-by-part. If

- concurrent service parts are non-conflicting,
- transfer of control between consecutive service parts is always controlled by an individual place,
- the location of the service users, the knowledge of the unstructured service parameters and the virtual-queues states assumed upon the start of a service part are always secured at the end of the preceding service part, and
- the choice between alternative service parts is always controlled by an individual place and the alternatives have equal participants sets,

then merging of the protocols derived for the individual service parts requires no additional protocol interactions, just extension of the context identifiers for MEPs (the N carried in protocol messages) and choices (the N in `alt(`N`)`) with the identifier of the service part to which they belong. The composition could be easily automated.

8 Comparison with Similar Algorithms and Plans for Further Work

We know no other author supporting implementation of user mobility and distributed virtual queues in a protocol derivation algorithm for LOTOS or a similar specification language.

Unstructured service parameters are supported in [6,9,20]. [6] poses several rather unnatural restrictions on parameter access. The restrictions have probably been inspired by the original semantics of LOTOS, and strongly simplify the protocol derivation process. [9] has removed the restrictions, except for those preventing clashes upon concurrent access of virtual variables, but is not able to automatically generate guards for parameter exchanges. An interesting property of [9] is that it allows flexible scheduling of the exchanges, while we automatically enforce a particular policy. Anyhow, the degree of automation in our tool is much higher than in [9]. In [20], service primitives are not atomic events, but transactions expected to run a special subprotocol for locking and unlocking the virtual variables they access. The subprotocol is not an integral part of the derived formal protocol specification. None of the algorithms addresses the possibility that a place forgets a service parameter or obtains its value without explicitly communicating with a place currently knowing it.

We allow only local choices, or choices based on the value of some previously known service parameters. Some other algorithms also support distributed choices. [21] allows only a pair of places and no concurrency in the service, but distributed choices are implemented by hidden loops of protocol exchanges. In [20], a distributed choice can be resolved directly by the transaction execution subprotocol, if we make the starting transactions of alternatives adequately compete on a variable. The distributed implementation of the LOTOS disabling operator proposed in [23] suggests that a conflict can be efficiently resolved by introducing adequate real-time constraints, but [23] doesn't apply the idea to the implementation of the LOTOS choice operator.

We require that parallel service parts are independent from each other, while some other algorithms [3,8,15] allow them to synchronise on common service primitives. Among them, only [15] is aware that the implementations of the synchronised parts might collide on the asynchronous protocol channels they share [16].

We do not support implementation of the LOTOS disabling operator. Such attempts have been made in [3,8,15,23]. [23] resolves the distributed conflicts potentially occurring upon disabling by real-time constraints. [3,15] prevent the conflicts by requiring a high degree of centralisation. [8] resorts to a simplified dynamic semantics of the operator. Moreover, the implementations of the operator proposed in [3,8] are not correct in all contexts [16]. Perhaps we could follow the ideas of [22], but they would probably be difficult to implement in the presence of service parameters and mobility.

Some recent research [15,24] indicates that it would be possible to decrease the number of the protocol messages serving for inter-place synchronisation.

Like for example [3,8,20,23,24], we have already experimented with automated protocol derivation for services with process recursion [15]. In the presence of mobile users, and particularly of virtual queues, it would probably require several restrictions on the recursion type and a more advanced scheme for recording the server state, than just the `alt` variables.

Like [6,9], we handle service parameters in a very abstract way, referring only to the place and type of their access, and not to their exact values. That could be amended, like for example in [20,23].

It would be interesting to implement handling of real-time

requirements, like for example in [10,19,23].

9 Closing Remarks

As indicated in Section 7, derivation of protocols for services supporting mobile users and distributed virtual queues is a very complex task, usually too complex to be faultlessly and quickly carried out by a human. On the other hand, the task is logically very simple and can be easily mechanised. Prolog has proven an excellent programming language for the purpose, provided that the underlying machine is sufficiently powerful.

References

[1] G. von Bochmann and R. Gotzhein, Deriving protocol specifications from service specifications, in: *Proc. ACM SIGCOMM'86 Symp.* (Vermont, USA, 1986) 148–156.

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

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

[4] P. van Eijk, Tools for LOTOS specification style transformation, in: S. T. Vong, ed., *Formal Description Techniques II* (North-Holland, Amsterdam, 1990) 43–51.

[5] K. Go, A decomposition of a formal specification: An improved constraint-oriented method, *IEEE Trans. on Software Engineering* **25** (1999) 258–273.

[6] R. Gotzhein and G. von Bochmann, Deriving protocol specifications from service specifications including parameters, *ACM Trans. on Comput. Systems* **8** (1990) 255–283.

[7] C. Kant, T. Higashino and G. von Bochmann, *Deriving Protocol Specifications Written in LOTOS*, Research Report #805, Université de Montreal (1992).

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

[9] M. Kapus-Kolar, Deriving protocol specifications from service specifications including parameters, *Microprocessing & Microprogramming* **32** (1991) 731–738.

[10] M. Kapus-Kolar, Deriving protocol specifications from service specifications with heterogeneous timing requirements, in: *Proc. 3rd IEE Int. Conf. on Software Engineering for Real-Time Systems* (IEE, London, 1991) 266–270.

[11] M. Kapus-Kolar, Automated derivation of protocols handling distributed virtual queues, in: *Proc. 10th Int. Symp. on Computer and Information Sciences* (Ephesus, Izmir, Turkey, 1995).

[12] M. Kapus-Kolar, Deriving protocols for mobile service users, *Elektrotehniški vestnik* **62** (1995) 299–307.

[13] M. Kapus-Kolar, *A Prolog Tool for Protocol Derivation*, Jožef Stefan Institute Technical Report 7342 (1996).

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

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

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

[17] M. Kapus-Kolar, J. Rugelj and M. Bonač, Deriving protocol specifications from service specifications, in: M. H. Hamza, ed., *Proc. 9th IASTED Int. Symp. Applied Informatics* (Acta Press, Anaheim-Calgary-Zürich, 1991) 375–378.

[18] F. Khendek, G. von Bochmann and C. Kant, New results on deriving protocol specifications from service specifications, in: *Proc. ACM SIGCOMM'89 Symp.* (1989) 136–145.

[19] A. Khoumsi, New results for deriving protocol specifications from service specifications for real-time applications, in: *Proc. MCSEAI* (Tunis, 1998).

[20] A. Khoumsi and G. von Bochmann, Protocol synthesis using basic LOTOS and global variables, in: *Proc. Int. Conf. on Networks and Protocols* (Tokyo, 1995) 126–133.

[21] R. Langerak, Decomposition of functionality: A correctness-preserving LOTOS transformation, in: L. Logrippo, R. Probert and H. Ural (eds.), *Protocol Specification, Testing, and Verification X* (North-Holland, Amsterdam, 1990) 229–242.

[22] K. Naik, Z. Cheng and D. S. L. Wei, Distributed implementation of the disabling operator in LOTOS, *Information and Software Technology* **41** (1999) 123–130.

[23] A. Nakata, T. Higashino and K. Taniguchi, Protocol synthesis from timed and structured specifications, in: *Proc. Int. Conf. on Networks and Protocols* (Tokyo, 1995) 74–81.

[24] A. Nakata, T. Higashino and K. Taniguchi, Protocol synthesis from context-free processes using event structures", in: *Proceedings of IEEE 5th Int. Workshop on Real-Time Computing Systems and Applications* (IEEE CS Press, 1998) 173–180.

[25] K. Saleh, Synthesis of communication protocols: An annotated bibliography, *Computer Communication Review* **26** (1996) no. 5, 40–59.