PAPER

Compositional Service-Based Construction of Multi-Party Time-Sharing-Based Protocols

Monika KAPUS-KOLAR†, ***Nonmember***

SUMMARY Distributed execution of a service often means that various places compete for the right to progress. If they exchange the right by explicit communication, there is a continuous flow of protocol messages. If the maximum transit delay of the communication medium is short, a better solution is to restrict progress of places to their individual time windows. The paper describes how to derive such time-sharing-based multi-party protocols for well-formed services specified in LOTOS/T+. The method is compositional with respect to the structure of the given service specification, supporting alternative, sequential, interrupt and parallel composition of services.

key words: *distributed service implementation, protocol synthesis, LOTOS/T+*

1. Introduction

For its users, a distributed server is a black box interacting with its environment through a set of service access points (SAPs), as illustrated in Fig. 1(a). The behaviour of a server at its SAPs is the *service* it offers. The atomic instantaneous interactions constituting a service are its primitives (SPs).

In a more detailed view (Fig. 1(b)), each SAP belongs to a particular place and is there supported by a particular protocol entity (PE). If necessary, the PEs communicate over a medium, i.e. execute a *protocol* implementing the service. We limit our discussion to protocols operating over reliable media.

For rapid prototyping of distributed servers, it is important that derivation of protocol specifications from service specifications can be automated [1]. The subject of the present paper is protocol derivation based on specifications written in LOTOS [2], [3], a standard process-algebraic language for formal specification of concurrent and reactive systems. In the recent years, many algorithms have been proposed for the purpose, e.g. [4]–[12].

The most challenging problem in protocol synthesis is implementation of distributed external choice. External choice is the generic form of choice in LOTOS. For a pair of SPs, it means that a server virtually permanently offers to its users the possibility to invoke them, but takes care that at most one is actually invoked. If the SPs belong to different places, the choice is distributed and its implementation requires that the places repeatedly exchange the right to enable an SP.

Competing places can exchange the right either by explicit communication or by time-sharing. Some protocol synthesis methods based on the first principle are [4], [9]–[13]. A less desirable property of such protocols is continuous communication of competing places. If the maximum transit delay of the communication medium is bounded and known, that can be avoided by using time-sharing instead, as hinted in [6], [13], [14]. In our paper, we use time-sharing to simplify the method of [10] for compositional derivation of protocols.

The paper is organized as follows. We start with some examples for illustration and motivation (Section 2). Section 3 more precisely defines the adopted specification language, the server model, the concept of a well-formed service, and the protocol derivation problem. Section 4 describes the new protocol derivation method. Section 5 concludes the paper.

2. Motivation

Suppose that a distributed server consists of n terminals (i.e. n PEs), connected by a local network. We assume that the clocks of the terminals are well synchronized, by a protocol (e.g. [15]) running in the background.

Let each terminal be a simple device communicating with its user through a single button (one of the n SAPs). For such a button to be pushed (i.e. for an SP to be executed at the place), it is necessary not only that the user presses the button, but also that the button is unlocked by the terminal.

Suppose that the users have to find the answer to a question and then press their button. The task of the distributed server is to ensure that only the button of the fastest competitor will be pushed. In other words, the required service is to choose one among the n possible SPs, where the choice is expected to depend

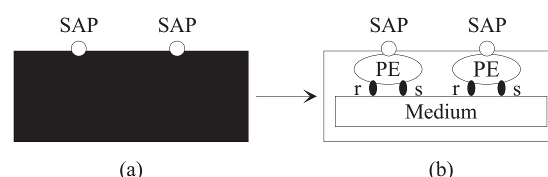


Fig. 1 A distributed server (a) and its internal structure (b)

Manuscript received September 17, 2002.
Manuscript revised April 1, 2003.

†The author is with the Jožef Stefan Institute, POB 3000, SI-1001 Ljubljana, Slovenia

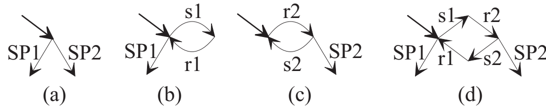


Fig. 2 A purely message-based implementation of binary distributed external choice: (a) the service, (b) PE_1, (c) PE_2, and (d) PE_1 and PE_2 combined over a reliable medium

exclusively on the behaviour of the service users, i.e. not on internal decisions of the server.

Obviously, no two buttons may ever be unlocked simultaneously, but every button must still be locked so seldom that its user does not perceive it. That requires that terminals are organized into a virtual token ring, where only the current token owner may have its button unlocked. Protocol synthesis methods like [4], [9]–[13] suggest that to pass the token, the owner sends a message to the next terminal in the ring. The strategy results in a continuous flow of messages (see Fig. 2).

Suppose that every terminal acts promptly, i.e. that upon receiving the token, it checks its user in the next time unit, and then, if the button has not been pushed, immediately passes the token. Let $\overline{d}$ and $d_{\max}$ denote the average and the maximum transit delay of a protocol message, where we assume that even permanent worst-case conditions secure the required quality of the service. Every button will be unlocked every $(n * (\overline{d} + 1))$ time units on the average, and every $(n * (d_{\max} + 1))$ time units in the worst case.

The key observation of our paper is that terminals can implement the worst-case periodic unlocking *without communicating*. It is important only that upon detecting that its button has been pushed, the winning terminal immediately reports to the others, so that they receive the message before they would unlock their buttons again (see Fig. 3). The alternative strategy requires only $(n - 1)$ messages. In comparison to the continuous message flow, this is a vast improvement.

If the described application is the only one running on the local network, such optimization is quite irrelevant. There are, however, cases, where it is crucial. Take, for example, the problem of factory automation. A factory consists of numerous machines and humans participating in a production process. They co-operate, but also compete for resources. At any moment, there are numerous concurrent activities, involving numerous distributed choices. Each activity runs its own protocol for co-ordination of its participants, and most of those protocols use the local communication network, so the traffic load might often be critically high. Consequently, reducing a continuous message flow to just a few messages is most welcome! One could argue that the described optimization typically requires a very short $d_{\max}$, while in an overloaded network, the delay tends to grow. However, if the optimization is introduced extensively, the network will no longer be overloaded and $d_{\max}$ will probably become acceptable.

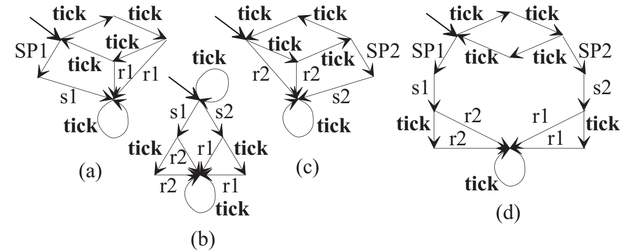


Fig. 3 A time-sharing-based implementation of binary distributed external choice: the relevant parts of (a) PE_1, (b) the medium and (c) PE_2, and (d) their combination

On the other hand, there are cases where time-sharing is the inferior strategy. Let us return to our first example. If time-sharing-based token-passing is introduced, $\overline{d}$ virtually increases to $d_{\max}$, i.e. buttons are on the average unlocked less often than with message-based token-passing. If $d_{\max}$ is much longer than $\overline{d}$, the service implementation will probably be judged with respect to its average performance, favouring the message-based solution.

However, the above is true only if the nature of the protocol doesn't influence $\overline{d}$ and $d_{\max}$. In the factory example, extensive introduction of time-sharing might decongest the medium to such an extent that $d_{\max}$ decreases below the previous $\overline{d}$, so that the service quality is better than with message-based token-passing.

Often the transit delay of the medium is most of the time quite short, but exceptional conditions might suddenly make it much longer than expected. If the application is not critical, one can still introduce time-sharing, estimating the maximum delay without considering the exceptional cases. However, all critical messages (in our first example, those reporting that a button has been pushed) must carry a time stamp, so that an exception can be signalled if they arrive after the deadline. Such a signal indicates that some SPs might have been illegally executed after the violated deadline.

Finally, there are cases where none of the two alternative strategies provides the required service quality. If the number of users in our first example is very large, buttons will not be unlocked sufficiently often even if token-passing is implemented in an optimal way. In such a case, the degree of centralization has to be increased, i.e. individual terminals must become responsible for multiple buttons. If there are less terminals involved in the distributed choice, they will receive the token more often, while the choice between buttons belonging to the same terminal is a local matter.

To conclude, there are many cases where time-sharing-based token-passing is the superior strategy. However, devising of suitable time constraints might be non-trivial, because individual SPs might be involved into several distributed conflicts simultaneously and because the transit delay of protocol messages might de-

pend on the direction of communication. So although time-sharing-based protocols introduce less messages, we need a method for their systematic construction. Such a method is proposed below.

3. Formalization of the Protocol Derivation Problem

3.1 Specification Language

The language we adopt for specification of services and protocols is LOTOS/T+, a timed version of LOTOS formally defined in [6] and outlined in Table 1.

"**stop**" denotes a deadlock process.

"**exit**" denotes a process ready to successfully terminate anytime, i.e. to execute a special event δ.

A process specified as "$a; P_2$" or "$a[T]; P_2$" executes an atomic, instantaneous action a, followed by process P_2. T, if present, is an additional requirement for t, the non-negative integer denoting the absolute global time instant when a is executed. We need four forms of T. An "$x = t$" specifies that a stores t into a variable x. A "$t = x$", an "$x \leq t \leq x + d$" or a "$t \textbf{ mod } d \in D$" specifies the legal values for t, respectively an instant, a closed interval and a cyclic series of instants.

A "$P_1 [] P_2$" defines a process willing to behave as P_1 or as P_2, where the choice is made upon the first event (an action or δ) of the selected alternative.

A "$P_1|[A]|P_2$" denotes a process executing processes P_1 and P_2 concurrently and synchronized on all actions listed in A and on δ. A shorthand for the purely asynchronous case is "$P_1|||P_2$". "$|[A]|$" will sometimes be used as a prefix operator, combining and arbitrary number of processes, where we define that parallel composition of an empty list of processes is equivalent to **exit**.

A "$P_1 [> P_2$" denotes a process basically executing P_1, but as long as P_1 doesn't successfully terminate, ready to disable it and transfer control to P_2, upon its first event.

A "$P_1 \gg P_2$" denotes a process which first executes P_1. When it successfully terminates, P_2 is enabled.

A "**hide** A **in** P_1" denotes process P_1 with all its actions listed in A hidden, i.e. not available for synchronization.

An "**asap** A **in** P_1" denotes process P_1 with all its actions listed in A executed as soon as possible.

A "*Proc*" denotes an instance of a process called *Proc*.

For a process P, let $Act(P)$ list all its non-hidden actions.

Table 1 Outline of the adopted specification language

$P ::=$ **stop** (untimed deadlock)
$\mid$ **exit** (successful termination)
$\mid a; P_2$ (action prefix, untimed)
$\mid a[T]; P_2$ (action prefix, timed)
$\mid P_1 [] P_2$ (choice)
$\mid P_1 \|[A]\| P_2$ (parallel composition)
$\mid P_1 [> P_2$ (disabling)
$\mid P_1 \gg P_2$ (enabling)
$\mid$ **hide** A **in** P_1 (hiding)
$\mid$ **asap** A **in** P_1 ("as soon as possible" execution)
$\mid$ *Proc* (process invocation)

Table 2 Server model

$$Server = \textbf{hide } Act(Medium) \textbf{ in asap } Act(Medium) \textbf{ in } ((|||_{\forall p} PE_p)|[Act(Medium)]|Medium)$$

$$Medium = (Medium|||(|||_{\forall p,p',m} Medium_m^{p,p'}))$$

$$Medium_m^{p,p'} = (\textbf{exit}[]\textbf{hide } \mathbf{d}_m^{p,p'} \textbf{ in } (\mathbf{s}_m^{p,p'}[x = t]; \mathbf{d}_m^{p,p'}[x \leq t \leq x + d^{p,p'}]; ((\mathbf{r}_m^{p,p'}; \textbf{exit})[]\textbf{exit})))$$

$$\forall p, p', p'' : (d^{p,p''} < (d^{p,p'} + d^{p',p''}))$$

3.2 Server Model

We assume that a distributed server interacts with its users at places 1 to N. In the following, let p, p' and p'' denote three different places. Each place p is supported by a PE PE_p. The remaining process in the server is $Medium$, the communication medium. The server structure is outlined in Table 2.

The PEs execute service primitives s. If $place(s)$, the pre-assigned place of an s, is p, the executor of s is PE_p. A PE_p might also send a protocol message m to a p', by an $\mathbf{s}_m^{p,p'}$, or receive m from p', by an $\mathbf{r}_m^{p',p}$. All $\mathbf{s}_m^{p,p'}$ and $\mathbf{r}_m^{p,p'}$ are executed as soon as possible and hidden from service users.

After $Medium$ accepts from a PE_p a message m for a $PE_{p'}$, it delivers it to place p' upon a hidden action $\mathbf{d}_m^{p,p'}$, after a transit delay which is not greater than a $d^{p,p'}$ negligibly short from the point of service users. So received by p', m waits (in a local buffer which is formally also a part of $Medium$) until claimed by $PE_{p'}$ upon an $\mathbf{r}_m^{p,p'}$. When there is no message in transit, $Medium$ is ready to successfully terminate.

3.3 Well-Formed Services

We define that a service process S is well-formed if it cannot deadlock and is structured as outlined in Table 3 (where necessary, use parentheses to make a service specification uniquely parsable).

Note the restricted use of **stop**. If a service process deadlocks, it can only be the "$(S_1|||\textbf{stop})$" part

Table 3 Outline of the service specification sublanguage

$S ::= s; \textbf{exit}$
$\mid s; S_2$ <u>where</u> $s \notin Act(S_2)$
$\mid S_1 [] S_2$ <u>where</u> $Act(S_1) \cap Act(S_2) = \emptyset$
$\mid S_1 \|[Act(S_1) \cap Act(S_2)]\| S_2$
$\mid (S_1 \|\|\| \textbf{stop}) [> S_2$ <u>where</u> $Act(S_1) \cap Act(S_2) = \emptyset$
$\mid S_1 \gg S_2$ <u>where</u> $Act(S_1) \cap Act(S_2) = \emptyset$

of an S of the form "$(S_1|||\mathbf{stop})[> S_2$", so that when "$S_1|||\mathbf{stop}$" deadlocks, S is still able to continue, by starting S_2.

Note also that **exit** never occurs in a decisive position. In other words, whenever a service reaches a state where it may continue either as an S_1 or an S_2, the next event can only be an SP (i.e. never a δ), either a starting SP of S_1 or a starting SP of S_2. With additional restrictions, we secure that such S_1 and S_2 have different starting SPs, so that service users can make the choice by selecting either an SP in S_1 or an SP in S_2. For example, "$\mathbf{exit}[](s;S)$" is not well-formed, because there is a decisive **exit**. "$((s_1;s_2;\mathbf{exit})|||\mathbf{stop})[> (s_2;s_3;\mathbf{exit})$" is also ill-formed, because upon execution of s_1, it reduces to "$((s_2;\mathbf{exit})|||\mathbf{stop})[> (s_2;s_3;\mathbf{exit})$", which can continue either by the left or by the right s_2.

Hence a well-formed service progresses steadily and deterministically. Determinism is not really a restriction, for non-determinism can always be simulated by interpreting different SPs as equivalent, or by synchronizing users only on the non-dummy SPs not intended for hiding. Even the restriction that S_1 in an "$S_1[> S_2$" must never terminate successfully can be circumvented by introduction of dummy SPs, e.g. by enhancing the service into a well-formed

$$((((S_1 \gg (done;\mathbf{exit}))|||\mathbf{stop})[> ((term;\mathbf{exit})[]S_2))$$
$$|[Act(S_2) + \{done, term\}]|((done;term;\mathbf{exit})[]S_2))$$

To simplify protocol synthesis, we also require that no s is executed more than once. Again interpreting different SPs as equivalent can help.

3.4 Protocol Derivation Problem

Given a well-formed service S and $place(s)$ for all s in $Act(S)$, we are looking for such PE_p that $Server$ will be equivalent to S from the point of service users, formally ($Server\ \underline{te}\ S$), where $\underline{te}$ denotes untimed testing equivalence [16].

For a service S, let $\mathbf{M}_p(S)$ denote the required PE_p. In other words, we are looking for a transformation $\mathbf{M}$ capable to generate a correct protocol for any well-formed service and any partitioning of SPs. We require that $\mathbf{M}$ is compositional, i.e. that for every service S specified as a composition of a pair of services S_1 and S_2, every $\mathbf{M}_p(S)$ is a composition of processes $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$.

Let $\mathbf{M}(S)$ denote $(|||_{\forall p}\mathbf{M}_p(S))$, i.e. the protocol for S. Let predicate $PP_p(S)$ indicate whether $Act(\mathbf{M}_p(S))$ is non-empty, i.e. whether p participates in $\mathbf{M}(S)$.

4. Principles of Protocol Derivation

Principles of protocol derivation will be discussed on an illustrative example presented in Table 4. In the example, the derived PE specifications are already simplified modulo timed weak bisimulation equivalence $\approx_t$ [14].

Table 4 An example service and its protocol

$$S = ((S_1|[a]|S_2) \gg (g;\mathbf{exit}))$$
$$S_1 = ((a;\mathbf{exit})[](b;(((e;f;\mathbf{exit})|||\mathbf{stop})[> (d;\mathbf{exit}))))$$
$$S_2 = ((a;\mathbf{exit})[](c;\mathbf{exit}))$$
$$place(a) = place(g) = 1$$
$$place(b) = place(e) = 2$$
$$place(c) = place(d) = place(f) = 3$$
$$\mathbf{M}_1(S) \approx_t ((\mathbf{M}_1(S_1)|[a, \mathbf{s}_a^{1,3}]|\mathbf{M}_2(S_2)) \gg (g;\mathbf{exit}))$$
$$\mathbf{M}_1(S_1) \approx_t ((a[T_a];((\mathbf{s}_a^{1,2};\mathbf{exit})|||(\mathbf{s}_a^{1,3};\mathbf{exit})))[](\mathbf{r}_b^{2,1};\mathbf{r}_d^{3,1};\mathbf{exit}))$$
$$\mathbf{M}_1(S_2) \approx_t ((a[T_a];\mathbf{s}_a^{1,3};\mathbf{exit})[](\mathbf{r}_c^{3,1};\mathbf{exit}))$$
$$\mathbf{M}_2(S) \approx_t ((\mathbf{r}_a^{1,2};\mathbf{exit})[]((b[T_b];((\mathbf{s}_b^{2,1};\mathbf{exit})|||(\mathbf{s}_b^{2,3};\mathbf{exit}))) \gg ((e[T_e];\mathbf{s}_e^{2,3};\mathbf{stop})[> (\mathbf{r}_d^{3,2};\mathbf{exit}))))$$
$$\mathbf{M}_3(S) \approx_t (\mathbf{M}_3(S_1)|[\mathbf{r}_a^{1,3}]|\mathbf{M}_3(S_2))$$
$$\mathbf{M}_3(S_1) \approx_t ((\mathbf{r}_a^{1,3};\mathbf{exit})[](\mathbf{r}_b^{2,3};((\mathbf{r}_e^{2,3};f;\mathbf{stop})[> (d[T_d];((\mathbf{s}_d^{3,1};\mathbf{exit})|||(\mathbf{s}_d^{3,2};\mathbf{exit}))))))$$
$$\mathbf{M}_3(S_2) \approx_t ((\mathbf{r}_a^{1,3};\mathbf{exit})[](c[T_c];\mathbf{s}_c^{3,1};\mathbf{exit}))$$
$$d^{1,2} = d^{2,1} = 3\ ,\ d^{2,3} = d^{3,2} = 4\ ,\ d^{3,1} = d^{1,3} = 5$$
$$T_a = (t\ \mathbf{mod}\ 12 = 0)$$
$$T_b = (t\ \mathbf{mod}\ 12 \in \{4,5,6,7,8\})$$
$$T_c = (t\ \mathbf{mod}\ 12 = 6)$$
$$T_d = (t\ \mathbf{mod}\ 10 = 0)$$
$$T_e = (t\ \mathbf{mod}\ 10 = 5)$$

4.1 Basic Strategy

The service implementation strategy is to have every s executed by $PE_{place(s)}$ and immediately reported to all p needing such a report, formally $RP_p(s)$. If s is reported to p within a particular protocol $\mathbf{M}(S)$, that is denoted by $RP_p(s,S)$. Take, for example, a service of the form "$S_1|[a]|S_2$", where S_1 is "$a;b;\mathbf{exit}$", S_2 is "$a;c;\mathbf{exit}$", a and b belong to a place p, and c belongs to a place p'. Within S_2, a at p guards c at p', hence within $\mathbf{M}(S_2)$, p must report a to p', formally $RP_{p'}(a)$, more precisely, $RP_{p'}(a,S_2)$. Within $\mathbf{M}(S_1)$, reporting of a to p' is not necessary, formally $\neg RP_{p'}(a,S_1)$.

Beside reports on individual SPs, there are no other protocol messages, as distributed conflicts are managed by time constraints. For an m reporting an s, it is only important that it unambiguously identifies s, so without loss of generality, we assume that m is the SP identifier s. As for reception of an m, the strategy is that the recipient PE_p receives it as soon as it is delivered to p.

Action reporting should be kept to a minimum, while respecting the following rules.

1) In an S specified as "$s;S_2$", s is an immediate guard of the starting SPs of S_2. Hence if s belongs to a p, and a starting SP of S_2 to a p', that requires $RP_{p'}(s,S)$. In our example, $RP_3(b)$ and $RP_3(e)$.

2) In an S specified as "$S_1 \gg S_2$", the ending SPs of S_1 are immediate guards of the starting SPs of S_2. Hence if there is an ending SP s of S_1 at a p, and a starting SP of S_2 at a p', $RP_{p'}(s,S_1)$ is required. In our example, $RP_1(c)$ and $RP_1(d)$.

3) In an S specified as "$(S_1|||\mathbf{stop})[> S_2$", the starting SPs of S_2 are disruptive for the SPs of S_1.

Table 5 Transformation $\mathbf{M}_p(S)$

$$
\begin{array}{l}
\mathbf{M}_p(s;\mathbf{exit}) := \underline{\text{if}}\ place(s) = p\ \underline{\text{then}} \\
\qquad\qquad (s[T_s]; (|||_{\forall p' | RP_{p'}(s,S)} (\mathbf{s}_s^{p,p'}; \mathbf{exit}))) \\
\qquad\qquad \underline{\text{else}}\ \underline{\text{if}}\ RP_p(s,S)\ \underline{\text{then}}\ (\mathbf{r}_s^{place(s),p}; \mathbf{exit}) \\
\qquad\qquad\quad \underline{\text{else}}\ \mathbf{exit}\ \underline{\text{endif}}\ \underline{\text{endif}} \\
\mathbf{M}_p(s; S_2) := \mathbf{M}_p((s;\mathbf{exit}) \gg S_2) \\
\mathbf{M}_p(S_1|[Act(S_1) \cap Act(S_2)]|S_2) := \\
\ (\mathbf{M}_p(S_1)|[Act(\mathbf{M}_p(S_1)) \cap Act(\mathbf{M}_p(S_2))]|\mathbf{M}_p(S_2)) \\
\mathbf{M}_p(S_1 \gg S_2) := (\underline{\text{if}}\ PP_p(S_1)\ \underline{\text{then}}\ \mathbf{M}_p(S_1) \gg \underline{\text{endif}}\ \mathbf{M}_p(S_2)) \\
\mathbf{M}_p(S_1[]S_2) := (\mathbf{M}_p(S_1)[]\mathbf{M}_p(S_2)) \\
\mathbf{M}_p((S_1|||\mathbf{stop})[> S_2) := ((\mathbf{M}_p(S_1)|||\mathbf{stop})[> \mathbf{M}_p(S_2))
\end{array}
$$

Hence if there is a starting SP s of S_2 at a p, and an SP of S_1 at a p', $RP_{p'}(s, S_2)$ is required. In our example, $RP_2(d)$.

4) In an S specified as "$S_1[]S_2$", the starting SPs of S_2 are disruptive for the starting SPs of S_1. Hence if there is a starting SP s of S_2 at a p, and a starting SP of S_1 at a p', $RP_{p'}(s, S_2)$ is required. Analogously, if there is a starting SP s of S_1 at a p, and a starting SP of S_2 at a p', $RP_{p'}(s, S_1)$ is required. In our example, $RP_2(a, S_1)$, $RP_1(b, S_1)$, $RP_3(a, S_2)$ and $RP_1(c, S_2)$.

5) For an $\mathbf{M}((S_1|||\mathbf{stop})[> S_2)$, we require that $PP_p(S_1)$ implies $PP_p(S_2)$ for every p, so that every participant is informed of the disabling. If not otherwise, a $PP_p(S_2)$ can be secured by setting to true $RP_p(s, S_2)$ for an s in S_2.

6) For an $\mathbf{M}(S_1[]S_2)$, we require $(PP_p(S_1) = PP_p(S_2))$ for every p, so that every participant is informed of the choice. If not otherwise, a $PP_p(S_k)$ can be secured by setting to true $RP_p(s, S_k)$ for an s in S_k. In our example, $RP_3(a, S_1)$.

7) To implement an $RP_p(s, S)$ for an S specified as "$S_1|[Acts]|S_2$" with s in $Acts$, it suffices to implement $RP_p(s, S_1)$ or $RP_p(s, S_2)$. In our example, we were free to decide whether $RP_2(a)$ should imply $RP_2(a, S_2)$ or not. However, if it did, $RP_2(c)$ would also be required by the rule 6 above.

Transformation $\mathbf{M}$ is specified in Table 5, where T_s is the time constraint of s, defined in Section 4.2.

4.2 Time Sharing

The necessary time constraints are implemented as static time-sharing. Each s is assigned its time constraint T_s of the form "$t\ \mathbf{mod}\ d_s \in D_s$". The constraints should be such that SPs are enabled as often as possible, while respecting the following restrictions, in which t satisfies T_s and t' is the first time satisfying $((t' \geq t) \wedge (t'\ \mathbf{mod}\ d_{s'} \in D_{s'}))$.

1) If in an "$S_1[]S_2$", s is a starting SP of S_1 and s' a starting SP of S_2, or vice versa, and $(place(s) \neq place(s'))$, then $(t' > (t + d^{place(s),place(s')}))$, so that with rule 4 in Section 4.1, $PE_{place(s')}$ detects s before it would enable s' again.

2) If in an "$(S_1|||\mathbf{stop})[> S_2$", s is a starting SP of S_2 and s' an SP of S_1, and $(place(s) \neq place(s'))$, then $(t' > (t + d^{place(s),place(s')}))$, so that with rule 3 in Section 4.1, $PE_{place(s')}$ detects s before it would enable s' again.

3) If in an "$(S_1|||\mathbf{stop})[> S_2$", s is an SP of S_1 and s' a starting SP of S_2, then for every p with $RP_p(s, S_1)$, $(t' > (t + d^{place(s),p}))$, so that reporting of s (if it occurs) is completed before s' is enabled again.

Computing time constraints for a service, first partition SPs in such a way that an SP and the SPs for which it is disruptive are always in the same group. In our example, there are groups $\{a, b, c\}$ and $\{e, f, d\}$.

Computing constraints for such a group, first find the restrictions applying to individual pairs of SPs. An individual SP might be subject to several such restrictions, possibly stemming from various composition operators from various levels of the service specification. For our example group $\{a, b, c\}$, there are four restrictions of type 1, respectively regulating enabling of a after b, of b after a, of a after c, and of c after a. The first two restrictions stem from the "[]" operator in S_1, while the other two from the "[]" operator in S_2.

A set of restrictions for a group of SPs can usually be satisfied in multiple ways, differing in how often individual SPs are enabled. For our group $\{a, b, c\}$, one possible combination of T_a, T_b and T_c is given in Table 4. An alternative solution can be computed with the following generally applicable strategy:

1) Find such constraints that SPs are enabled in turns, so that the next SP is enabled in the time instant after the arrival deadline of reports on the previous SP. Suppose that in our example, a is enabled at time 0. The latest arrival time of reports on a is 5, so b is enabled at time 6. The latest arrival time of reports on b is 10, so c is enabled at time 11. The latest arrival time of reports on c is 16, so a is enabled again at time 17, i.e. at time 0 of the next cycle. Hence T_a is "$t\ \mathbf{mod}\ 17 = 0$", T_b is "$t\ \mathbf{mod}\ 17 = 6$", and T_c is "$t\ \mathbf{mod}\ 17 = 11$".

2) Having found such a correct, though not necessarily optimal solution, further relax the constraints as much as possible. For our example, such a relaxation would give "$t\ \mathbf{mod}\ 17 \in \{0, 1, 2\}$" for T_a, "$t\ \mathbf{mod}\ 17 \in \{6 \ldots 13\}$" for T_b, and "$t\ \mathbf{mod}\ 17 \in \{8 \ldots 11\}$" for T_c. The new solution enables SPs more often, but has a longer cycle.

4.3 Implementation of Individual Service Primitives

Implementing an "$s;\mathbf{exit}$", we implement execution and reporting of s. The resulting server first executes s at $place(s)$, at a legal t. Afterwards, all the required reports are sent concurrently. They are promptly received, and the server successfully terminates. Modulo te, the hidden exchange of reports is irrelevant for the service, and so is the fact that s is not enabled continuously. Hence the protocol has all the required properties.

Table 6 Outline of proof for $S=(S_1\vert[\mathit{Act}(S_1)\cap \mathit{Act}(S_2)]\vert S_2)$

Step	Expression
	$(S_1\vert[\mathit{Act}(S_1)\cap \mathit{Act}(S_2)]\vert S_2)$
(1)	$\underline{\text{te}}\ ((\mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in\ asap}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ (\mathbf{M}(S_1)\vert[\mathit{Act}(\mathit{Medium})]\vert \mathit{Medium}))\vert[\mathit{Act}(S_1)\cap \mathit{Act}(S_2)]\vert$ $(\mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in\ asap}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ (\mathbf{M}(S_2)\vert[\mathit{Act}(\mathit{Medium})]\vert \mathit{Medium})))$
(2)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ ((\mathbf{asap}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ (\mathbf{M}(S_1)\vert[\mathit{Act}(\mathit{Medium})]\vert \mathit{Medium}))\vert[\mathit{Act}(S_1)\cap \mathit{Act}(S_2)]\vert$ $(\mathbf{asap}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ (\mathbf{M}(S_2)\vert[\mathit{Act}(\mathit{Medium})]\vert \mathit{Medium})))$
(3)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ ((\mathbf{M}(S_1)\vert[\mathit{Act}(\mathit{Med}(S_1))]\vert \mathit{Med}(S_1))\vert[\mathit{Act}(S_1)\cap \mathit{Act}(S_2)]\vert(\mathbf{M}(S_2)\vert[\mathit{Act}(\mathit{Med}(S_2))]\vert \mathit{Med}(S_2)))$ $\mathit{Med}(S)=(\vert\vert\vert_{\forall s\vert\exists p:RP_p(s,S)}(\vert[s]\vert_{\forall p\vert RP_p(s,S)}\mathit{Med}_s^p))$ $\mathit{Med}_s^p=(\mathbf{exit}[](s[x=t];\mathbf{s}_s^{place(s),p}[t=x];\mathbf{r}_s^{place(s),p}[x\leq t\leq x+d^{place(s),p}];\mathbf{exit}))$
(4)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ ((\mathbf{M}(S_1)\vert[\mathit{Act}(\mathit{Med}(S_1))]\vert \mathit{Med}(S_1))\vert[(\mathit{Act}(S_1)\cap \mathit{Act}(S_2))\cup(\mathit{Act}(\mathit{Med}(S_1))\cap \mathit{Act}(\mathit{Med}(S_2)))]\vert$ $(\mathbf{M}(S_2)\vert[\mathit{Act}(\mathit{Med}(S_2))]\vert \mathit{Med}(S_2)))$
(5)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ ((\mathbf{M}(S_1)\vert[\mathit{Act}(\mathbf{M}(S_1))\cap \mathit{Act}(\mathbf{M}(S_2))]\vert\mathbf{M}(S_2))\vert[\mathit{Act}(\mathit{Med}(S_1))\cup \mathit{Act}(\mathit{Med}(S_2))]\vert$ $(\mathit{Med}(S_1)\vert[\mathit{Act}(\mathit{Med}(S_1))\cap \mathit{Act}(\mathit{Med}(S_2))]\vert \mathit{Med}(S_2)))$
(6)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ ((\mathbf{M}(S_1)\vert[\mathit{Act}(\mathbf{M}(S_1))\cap \mathit{Act}(\mathbf{M}(S_2))]\vert\mathbf{M}(S_2))\vert[\mathit{Act}(\mathit{Med}(S))]\vert \mathit{Med}(S))$
(7)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in\ asap}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ (((\vert\vert\vert_{\forall p}\mathbf{M}_p(S_1))\vert[\mathit{Act}(\mathbf{M}(S_1))\cap \mathit{Act}(\mathbf{M}(S_2))]\vert(\vert\vert\vert_{\forall p}\mathbf{M}_p(S_2)))$ $\vert[\mathit{Act}(\mathit{Medium})]\vert \mathit{Medium})$
(8)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in\ asap}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ ((\vert\vert\vert_{\forall p}(\mathbf{M}_p(S_1)\vert[\mathit{Act}(\mathbf{M}_p(S_1))\cap \mathit{Act}(\mathbf{M}_p(S_2))]\vert\mathbf{M}_p(S_2)))$ $\vert[\mathit{Act}(\mathit{Medium})]\vert \mathit{Medium})$
(9)	$\underline{\text{te}}\ \mathbf{hide}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in\ asap}\ \mathit{Act}(\mathit{Medium})\ \mathbf{in}\ ((\vert\vert\vert_{\forall p}\mathbf{M}_p(S))\vert[\mathit{Act}(\mathit{Medium})]\vert \mathit{Medium})$

4.4 Implementation of Parallel Composition

Executing an S specified as "$S_1\vert[\mathit{Act}(S_1)\cap \mathit{Act}(S_2)]\vert S_2$", each place p executes $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$ concurrently and synchronized on their common actions [10].

The correctness proof is outlined in Table 6. As in the following sections, we assume that $\mathbf{M}(S_1)$ and $\mathbf{M}(S_2)$ have all the expected properties.

In the first step, each S_k in S is replaced with its distributed implementation. That is possible because for every s on which S_1 and S_2 are synchronized, the time constraint is the same in every $\mathbf{M}(S_k)$, namely T_s, so that once s is logically enabled, it is repeatedly available in both $\mathbf{M}(S_k)$ simultaneously.

In the second step, hiding of protocol exchanges is moved one level higher.

In the third step, the irrelevant parts of the two media are omitted. Knowing that both partial protocols actually execute their protocol exchanges promptly, the **asap** constraints are encoded directly as time constraints of the media.

In the fourth step, the two partial servers are synchronized on their common protocol exchanges. In such a server, no partial server ever hinders prompt report transmission in its peer or delays a report reception in its peer more than the medium might do. Hence all the time constraints necessary for prevention of divergence in execution of S_1 or S_2 are still valid, implying that any SP trace executable by a partial server within the new context is valid, implying that every executable SP trace is valid. On the other hand, if users invoke SPs strictly one after another and with time gaps longer than the maximum transit delay, the server repeatedly enters (by executing all the outstanding transmissions and receptions of SP reports) a state in which both partial servers are stable, i.e. ready for all their legal next SPs, so that the entire server is in a stable state and ready for all the legal next SPs. Every legal SP trace is executable in such a way, and the server has no stable states besides those reachable as above. Hence the executable SP traces of the server are exactly the traces of S, and the server never refuses their invocation.

In the fifth step, processes are regrouped into the $\mathbf{M}$ processes and the medium processes. Regrouping has originally been studied for LOTOS processes [17], but as **tick** steps modelling time progress may be interpreted as ordinary actions on which all concurrent processes synchronize, the rules obviously also apply to LOTOS/T+ processes.

In the sixth step, the medium is simplified by merging individual pairs of identical, fully synchronized Med_s^p. Suitably regrouped, the remaining Med_s^p constitute exactly $\mathit{Med}(S)$.

Knowing that each $\mathbf{M}(S_k)$ is always ready to transmit and receive SP reports as soon as possible, some time constraints of the medium are in the seventh step converted into the usual **asap** constraints. Also, the irrelevant parts of the medium are re-introduced and the structure of each $\mathbf{M}(S_k)$ is revealed.

In the eighth step, the $\mathbf{M}$ processes are regrouped into pairs belonging to individual places. In the ninth step, the resulting server is recognized as exactly the proposed distributed implementation of S. That concludes the proof.

4.5 Implementation of Sequential Composition

Executing an S specified as "$S_1 \gg S_2$", any p with $\neg PP_p(S_1)$ executes just $\mathbf{M}_p(S_2)$. For such a p, $\mathbf{M}_p(S_1)$ is equivalent to **exit**, hence we may pretend that $\mathbf{M}_p(S)$ is actually "$\mathbf{M}_p(S_1)\vert\vert\vert\mathbf{M}_p(S_2)$".

For a p with $PP_p(S_1)$, $\mathbf{M}_p(S)$ is "$\mathbf{M}_p(S_1) \gg \mathbf{M}_p(S_2)$", but we pretend that it is actually "$\mathbf{M}_p(S_1)\vert\vert\vert\mathbf{M}_p(S_2)$" while an additional local constraint takes care that $\mathbf{M}_p(S_2)$ doesn't start until $\mathbf{M}_p(S_1)$ isn't ready for δ. For the scheme to work, $\mathbf{M}_p(S_1)$ must never enable δ as an alternative to an action.

By induction, we prove that the latter is true for any $\mathbf{M}_p(S)$.

1) If S is an "s; **exit**", that is obviously true.

2) If S is an "$S_1|[\ldots]|S_2$" or an "$S_1 \gg S_2$", $\mathbf{M}_p(S)$ inherits the property from $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$.

3) If S is an "$(S_1|||\mathbf{stop})[> S_2$", such inheritance works provided that $PP_p(S_1)$ implies $PP_p(S_2)$, and that is secured by rule 5 in Section 4.1.

4) If S is an "$S_1[]S_2$", such inheritance works provided that $(PP_p(S_1) = PP_p(S_2))$, and that is secured by rule 6 in Section 4.1.

Hence we may indeed pretend that all p execute "$\mathbf{M}_p(S_1)|||\mathbf{M}_p(S_2)$", while the additional local constraints take care of the rest.

With $((Act(S_1) \cap Act(S_2)) = \emptyset)$, "$\mathbf{M}_p(S_1)|||\mathbf{M}_p(S_2)$" is exactly $\mathbf{M}_p(S_1|[Act(S_1) \cap Act(S_2)]|S_2)$. In other words, the server basically executes the protocol for "$S_1|[Act(S_1) \cap Act(S_2)]|S_2$", for which Section 4.4 guarantees that it is correct.

Proving $\mathbf{M}(S)$, it remains to prove that the additional constraints secure that S_2 is executed strictly after S_1. It suffices to prove that no starting SP s' of S_2 is ever executed before an ending SP s of S_1. If s and s' both belong to a p', the additional constraint at p' is definitely sufficient. If s belongs to another place p, the constraint is sufficient provided that s is reported to p', and that is secured by rule 1 or rule 2 in Section 4.1. That concludes the proof.

4.6 Implementation of Choice

Executing an S specified as "$S_1[]S_2$", every p chooses between $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$. $((Act(S_1) \cap Act(S_2)) = \emptyset)$ implies $((Act(\mathbf{M}(S_1)) \cap Act(\mathbf{M}(S_2))) = \emptyset)$, hence $\mathbf{M}(S_1)$ and $\mathbf{M}(S_2)$ don't interfere. To prove $\mathbf{M}(S)$, it suffices to prove that the local choices are globally consistent.

Without loss of generality, we assume that the first event in an $\mathbf{M}_p(S)$ belongs to $\mathbf{M}_p(S_1)$.

1) If it is a δ, there is no action alternative to it in $\mathbf{M}_p(S)$ (see Section 4.5), implying that $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$ are both equivalent to **exit**, implying that any choice at p is globally consistent.

2) If it is an action, it is the first SP executed in S_1 or such an SP s has already been executed. Hence the choice at p is globally consistent provided that no SP of S_2 has been executed so far. Let s' be the first SP executed in S_2. If it has been executed before s, $place(s)$ has, according to rule 1 in Section 4.2, executed s after detecting s' upon an action in $\mathbf{M}_{place(s)}(S_2)$ (s' itself or reception of a report on it). If s' has been executed after s, $place(s')$ has, according to rule 1 in Section 4.2, executed it after detecting s upon an action in $\mathbf{M}_{place(s')}(S_1)$ (s itself or reception of a report on it). But no p can execute an action in $\mathbf{M}_p(S_1)$ after an action in $\mathbf{M}_p(S_2)$, or vice versa, hence there has been no such s'. That concludes the proof.

4.7 Implementation of Disabling

Executing an S specified as "$(S_1|||\mathbf{stop})[> S_2$", every p basically executes $\mathbf{M}_p(S_2)$, though it might execute a part of $\mathbf{M}_p(S_1)$ before $\mathbf{M}_p(S_2)$ starts. $((Act(S_1) \cap Act(S_2)) = \emptyset)$ implies $((Act(\mathbf{M}_p(S_1)) \cap Act(\mathbf{M}_p(S_2))) = \emptyset)$, hence $\mathbf{M}(S_1)$ and $\mathbf{M}(S_2)$ don't interfere. To prove $\mathbf{M}(S)$, it suffices to prove that no p abandons $\mathbf{M}_p(S_1)$ prematurely, that no SP in $\mathbf{M}(S_1)$ occurs after the first SP in $\mathbf{M}(S_2)$, and that every message sent within $\mathbf{M}(S_1)$ is also received.

A p abandons $\mathbf{M}_p(S_1)$ upon the first event in $\mathbf{M}_p(S_2)$. That might be prematurely only if the event is executed before the first SP in $\mathbf{M}(S_2)$, but in that case, it is a δ and there is no action alternative to it in $\mathbf{M}_p(S)$ (see Section 4.5), implying that $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$ are both equivalent to **exit**, implying that p may abandon $\mathbf{M}_p(S_1)$ anytime.

To prove the rest, let s be the first SP in $\mathbf{M}(S_2)$ and s' an SP executed in $\mathbf{M}(S_1)$. If s' has been executed after s, $place(s')$ has, according to rule 2 in Section 4.2, executed it after detecting s upon an action in $\mathbf{M}_{place(s')}(S_2)$ (s itself or reception of a report on it). But no p can execute an action in $\mathbf{M}_p(S_1)$ after an action in $\mathbf{M}_p(S_2)$, hence s' must have been executed before s. That implies that a report on s' could be still in transit or in the input buffer, but according to rule 3 in Section 4.2, that is impossible. That concludes the proof.

4.8 Possibilities for Further Optimization

The first optimization concerns implementation of an "$S_1 \gg S_2$". For implementation of sequential composition, it suffices that an ending place of S_1 reports to the starting places of S_2 only its last SP of S_1 [18].

The second optimization concerns implementation of an "$(S_1|||\mathbf{stop})[> S_2$". For implementation of disabling, reception of reports belonging to $\mathbf{M}(S_1)$ is irrelevant after activation of $\mathbf{M}(S_2)$, i.e. rule 3 in Section 4.2 is redundant for it. If we neglect the rule, such a report, if sent, will still reach the destination place, though it might remain in the input buffer.

Both types of optimization might cause that some reports are not received as originally planned. Hence such optimization is allowed only to an extent not involving reports for which prompt reception is crucial.

Another possible optimization concerns the contents of protocol messages. Originally, messages are strictly different. That is not always necessary, though possibilities for message reuse are not always easily identifiable.

5. Concluding Remarks

At least during periods of strong network congestion,

time-sharing-based virtual token-passing is superior to message-based passing.

Protocols with time-sharing-based token-passing are much simpler, because PEs report only what they have done, while in purely message-based protocols, places sometimes report also their current unwillingness for particular actions, i.e. that they are in some aspect passive [4], [10].

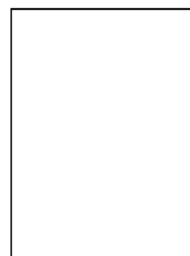
By letting protocol messages carry data as appropriate, the proposed service implementation strategy can be easily generalized to services with data-carrying SPs (see, for example, [19]).

The proposed strategy is also suitable for timed services. Protocol synthesis for such a service proceeds as for an untimed service, except that one first narrows the time windows of individual SPs to an extent which can be satisfied in a distributed implementation [6]. If time-sharing is used, it might be that an SP, even when already logically enabled, is available for execution only from time to time. So when allotting time to individual places, care must be taken that the originally specified time constraints of their SPs are met.

Derivation of time-sharing-based protocols from LOTOS-based service specifications is a bit tricky, because an SP might be specified as a rendezvous of several concurrent services S_1 to S_n. One must be aware that such an SP is available only when available in all the protocols $\mathbf{M}(S_1)$ to $\mathbf{M}(S_n)$. However, there are specification styles for which the concept of multirendezvous is indispensable, in particular the extremely expressive constraint-oriented style [20].

References

[1] K. Saleh, "Synthesis of communication protocols: An annotated bibliography," Computer Communication Review, vol. 26, no. 4, pp. 40–59, 1996.
[2] ISO/IEC, "Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour," ISO 8807, 1989.
[3] T. Bolognesi and E. Brinksma, "Introduction to the ISO specification language LOTOS," Computer Networks and ISDN Systems, vol. 14, no. 1, pp. 25–59, 1987.
[4] R. Langerak, "Decomposition of functionality: A correctness preserving LOTOS transformation," in Protocol Specification, Testing and Verification, X, eds. L. Logrippo, R. L. Probert, and H. Ural, pp. 203–218, North-Holland, 1990.
[5] E. Brinksma, R. Langerak, "Functionality decomposition by compositional correctness preserving transformation," South African Computer Journal, no. 13, pp. 2–13, 1995.
[6] A. Nakata, T. Higashino, and K. Taniguchi, "Protocol synthesis from timed and structured specifications," in Proc. of ICNP'95, pp. 74–81, IEEE Computer Society Press, 1995.
[7] C. Kant, T. Higashino, and G. von Bochmann, "Deriving protocol specifications from service specifications written in LOTOS," Distributed Computing, vol. 10, no. 1, pp. 29–47, 1996.
[8] B. B. Bista, K. Takahashi, and N. Shiratori, "A compositional approach for constructing communication services and protocols," IEICE Trans. Fundamentals, vol. E82-A, no. 11, pp. 2546–2557, 1999.
[9] B. B. Bista, K. Takahashi, and N. Shiratori, "Construction of service and protocol specifications," in Proc. ICOIN'2001, pp. 171–178, IEEE Computer Society Press, 2001.
[10] N. Maneerat, R. Varakulsiripunth, D. Seki, K. Yoshida, K. Takahashi, Y. Kato, B. B. Bista, and N. Shiratori, "Composition method of communication system specifications in asynchronous model and its support system," in Proc. of ICON'2001, pp. 64–69, IEEE Computer Society Press, 2001.
[11] B. B. Bista, N. Shiratori, "Construction of a multiple entities communication protocol by compositional approach," in Proc. DEXA'2001 Int. Workshop on Network-Based Information Systems, pp. 162–166, IEEE Computer Society Press, 2001.
[12] B. B. Bista, K. Takahashi, and N. Shiratori, "On constructing n-entities communication protocol and service with alternative and concurrent functions," IEICE Trans. Fundamentals, vol. E85-A, no. 11, pp. 2426–2435, 2002.
[13] M. Kapus-Kolar, "Global conflict resolution in automated service-based protocol synthesis," South African Computer Journal, vol. 27, pp. 34–48, 2001.
[14] A. Nakata, "Symbolic Bisimulation Checking and Decomposition of Real-Time Service Specifications," Ph.D. thesis, Osaka University, 1997.
[15] R. Tilak, A. D. George, and R. W. Todd, "Layered interactive convergence for distributed clock synchronization," Microprocessors and Microsystems, vol. 26, no. 9–10, pp. 407–420, 2002.
[16] E. Brinksma, G. Scollo, and C. Steenbergen, "LOTOS specifications, their implementations and their tests," in Protocol Specification, Testing, and Verification, VI, eds. B. Sarikaya and G. von Bochmann, pp. 349–360, North-Holland, 1987.
[17] T. Bolognesi, D. de Frutos-Escrig, and Y. Ortega-Mallén, "Graphical composition theorems for parallel and hiding operators," in Formal Description Techniques, III, eds. J. Quemada, J. Mañas, and E. Vázques, pp. 459–470, North-Holland, 1991.
[18] F. Khendek, G. von Bochmann, and C. Kant, "New results on deriving protocol specifications from service specifications," in Proc. of ACM SIGCOMM'89, pp. 136–145, ACM Press, 1989.
[19] M. Kapus-Kolar, "Deriving protocols for services supporting mobile users," Information and Software Technology, vol. 42, no. 9, pp. 619–631, 2000.
[20] C.A. Vissers, G. Scollo, M. van Sinderen, and H. Brinksma, "Specification styles in distributed systems design and verification," Theoretical Computer Science, vol. 89, pp. 179–206, 1991.



Monika Kapus-Kolar received the B.S. degree in electrical engineering from the University of Maribor, Slovenia, in 1981, and the M.S. and Ph.D. degrees in computer science from the University of Ljubljana, Slovenia, in 1984 and 1989, respectively. Since 1981 she has been with the Jožef Stefan Institute in Ljubljana. Her current research interests include formal specification techniques and methods for development of distributed systems and computer networks.

A Correction to “Compositional Service-Based Construction of Multi-Party Time-Sharing-Based Protocols”

Jožef Stefan Institute Technical Report #8896

Monika Kapus-Kolar

October 21, 2003

Abstract

We describe and correct a subtle error in our protocol synthesis algorithm published in *IEICE Trans. on Fundamentals, vol.E86-A, no. 9, pp. 2405–2412.*

Keywords: distributed service implementation, protocol synthesis, LOTOS/T+.

1 Introduction

In [1], we have proposed a method for deriving protocol specifications from service specifications written in LOTOS/T+ [2]. The method is based on the following assumptions:

1. When a place p executes a service primitive (SP) s for which reporting to another place p' is specified, it transmits the required report on s with no delay.
2. The worst-case transit delay for the protocol message is a finite and globally known $d^{p,p'}$.
3. When the message reaches p', it is with no delay accepted by $PE_{p'}$, the protocol entity of the place.

However, we have discovered that the third assumption is not always true in the derived protocols. Fortunately, the error can be easily corrected.

2 An Example of an Erroneous Protocol

Take the service “$a; S$”, where S is “$(((b; \mathbf{exit})|||\mathbf{stop})[> (c; \mathbf{exit}))$”. In the service, SP a is followed by SP b potentially disrupted by SP c. Let a, b and c belong to different places p, p' and p'', respectively, where $d^{p'',p'}$ is 2.

According to [1], a possible solution for the distributed conflict between b and c is to allow execution of b at times 0,4,8..., and execution of c at times 1,5,9.... With these restrictions, a prompt report on c reaches p' before b is enabled again. a must also be reported, both to p' and p'', for it guards execution of b and c.

Suppose that $PE_{p''}$ receives a report on a at time 3, and afterwards executes c at time 5. On the other hand, suppose that $PE_{p'}$ receives a report on a much later, at time 8. According to [1], this is also the moment when $PE_{p'}$ starts executing the protocol for S. Hence at time 8, $PE_{p'}$ suddenly becomes ready both for b and for a report on c. A report on c is already in the input buffer, having reached p' at time 7 or earlier. However, that does not prevent $PE_{p'}$ from executing b instead, but b after c is illegal.

3 A Solution to the Problem

The problem lies in how [1] encodes the intended behaviour of protocol entities. Take $\mathbf{M}(S_1)$ and $\mathbf{M}(S_2)$, the protocols derived for a service S_1 and a subsequent service S_2. For a PE_p with duties in $\mathbf{M}(S_1)$, [1] defines that $\mathbf{M}_p(S_1 \gg S_2)$, its behaviour within the protocol $\mathbf{M}(S_1 \gg S_2)$, is “$\mathbf{M}_p(S_1) \gg \mathbf{M}_p(S_2)$”. Hence $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$ are in strong sequential composition, while protocol message receptions in $\mathbf{M}_p(S_2)$ should actually commute with actions in $\mathbf{M}_p(S_1)$.

A starting action of $\mathbf{M}_p(S_2)$ might be an SP or a protocol message reception. Let $SS_p(S_2)$ be the set of those starting actions which are SPs.

If $SS_p(S_2)$ is empty, "$\mathbf{M}_p(S_1)|||\mathbf{M}_p(S_2)$", i.e. parallel composition of $\mathbf{M}_p(S_1)$ and $\mathbf{M}_p(S_2)$, is an adequate $\mathbf{M}_p(S_1 \gg S_2)$, because a premature start of $\mathbf{M}_p(S_2)$ is prevented already by other places.

If $SS_p(S_2)$ is not empty, an adequate $\mathbf{M}_p(S_1 \gg S_2)$ is
"$(\mathbf{M}_p(S_1) \gg ((|||_{\forall s \in SS_p(S_2)}(s; \mathbf{stop}))[> \mathbf{exit}))|[SS_p(S_2)]|\mathbf{M}_p(S_2)$",
where the operator "$|[SS_p(S_2)]|$" defines that the two parallel processes may execute actions in $SS_p(S_2)$ and successful termination "**exit**" only in co-operation. Hence as in the case of "$\mathbf{M}_p(S_1) \gg \mathbf{M}_p(S_2)$", PE_p takes care that actions in $SS_p(S_2)$ never occur before successful termination of $\mathbf{M}_p(S_1)$.

As a conclusion, we observe that $\mathbf{M}_p(S_1 \gg S_2)$ is the only case where [1] introduces guarding of protocol message receptions. Hence with the describe correction, all message are indeed accepted with no delay.

References

[1] M. Kapus-Kolar, "Compositional service-based construction of multi-party time-sharing-based protocols," IEICE Trans. on Fundamentals, vol.E86-A, no. 9, pp. 2405–2412, 2003.

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