Comments on deriving protocol specifications from service specifications written in LOTOS

Monika Kapus-Kolar

Jožef Stefan Institute

POB 3000, SI-1001 Ljubljana, Slovenia

Fax: +386 61 1262 102

monika.kapus-kolar@ijs.si

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, Ljubljana, where she is currently a researcher at the Department of Digital Communications and Networks. Her current research interests include formal specification techniques and methods for development of distributed systems and computer networks.

Comments on deriving protocol specifications from service specifications written in LOTOS

Abstract: An algorithm by Kant, Higashino and Bochmann for service-based protocol synthesis in the standard specification language LOTOS is discussed. It is demonstrated that the transformations for distributed implementation of synchronised parallel execution and of disabling are not correct in a general case.

Keywords: distributed service implementation, protocol synthesis, compositional correctness-preserving transformation, LOTOS

1 Introduction

[3] proposes an algorithm for service-based protocol synthesis for a distributed server consisting of an arbitrary finite number of protocol entities (PEs) pairwise communicating over reliable, unbounded FIFO channels. The algorithm takes a specification of the expected external behaviour of the server (the service), with each service action already allocated to one of the PEs, and derives behaviours of individual PEs (the protocol) together implementing the service. The specification language is Basic LOTOS [1].

The algorithm is compositional, i.e. if a service expression e is a composition of a set of subexpressions e', $\mathbf{T}_p(e)$ - the mapping of e onto a PE p - is expressed in terms of $\mathbf{T}_p(e')$ for each of the e', where implementation of each e' uses an individual set of protocol messages. Hence [3] actually proposes transformations for distributed implementation of individual LOTOS behaviour composition operators.

In the present paper we report that two of the transformations are not correct in a general case, i.e. they might generate a protocol with an unspecified reception or a deadlock. Section 2 discusses implementation of synchronised parallel execution, and Section 3 implementation

of disabling. Section 4 contains conclusions. It is expected that the reader is familiar with [3]. The notation used is also that of [3].

2 Problems with implementation of synchronised parallel execution

Let a server consist of PEs 1 and 2 and let

$e = e_1|[b^2]|e_2 = (((a^1;\text{exit})|||(b^2;\text{exit}))>>(c^2;\text{exit}))|[b^2]|(d^1;b^2;\text{exit})$

be the behaviour that we are implementing. The protocol generated by the algorithm of [3] would be of the form

$\mathbf{T}_1(e) = \mathbf{T}_1(e_1)|||\mathbf{T}_1(e_2) \approx (a^1;s_2(1);\text{exit})|||(d^1;s_2(2);\text{exit})$

$\mathbf{T}_2(e) = \mathbf{T}_2(e_1)|[b^2]|\mathbf{T}_2(e_2) \approx (b^2;r_1(1);c^2;\text{exit})|[b^2]|(r_1(2);b^2;\text{exit})$

where an "a^p" identifies action a at PE p, an "$s_p(m)$" transmission of protocol message m to p, an "$r_p(m)$" reception of m from p, and the protocol messages 1 and 2 respectively identify the ">>" operator and the ";" operator connecting d^1 and b^2.

One of the possible runs of the distributed server is "$a^1;s_2(1);d^1;s_2(2)$". After the actions, the e_1 part could execute b^2, but that requires synchronisation with b^2 in the e_2 part, that is guarded by a non-executable $r_1(2)$. For note that the e_1 and the e_2 part of the server use the same protocol channel, where the message currently available for reception is 1, not 2. If the capacity of the channel buffer is 1, as assumed in Section 5 of [3], the server blocks even before $s_2(2)$. A solution would be to introduce, at least virtually, separate channels from PE 1 to PE 2 for each of the two server parts. Besides it seems that such problems never occur if parallel service parts are executed independently (the case of pure interleaving).

3 Problems with implementation of disabling

Implementing an e of the form $e_1[>e_2$, where e_2 is supposed to be written in an action prefix form and have a single starting participant (SP), [3] doesn't require instantaneous disabling of

the e_1 part by the e_2 part of the server, for that is in a distributed system with asynchronous internal communication generally impossible. Upon an initial action within the e_2 part, a protocol message is sent to all PEs to disable their e_1 parts, but before receiving the message, a recipient might execute some further actions within its e_1 part.

For any PE p, $\mathbf{T}_p(e)=(\mathbf{T}_p(e_1)>>\mathbf{Rel}_p(e_1))[>\mathbf{T}_p(e_2)$, i.e. upon termination of the e_1 part, a protocol message is sent from every ending participant (EP) of e_1 ([3] requires that the only starting participant of e_2 is one of them) to any PE in the server. The message is intended for prevention of disabling after e_1 terminates, though we accept that communication delays might sometimes render the prevention unsuccessful.

However, the transformation has several problematic properties.

1) The start of $\mathbf{T}_p(e_2)$ might prevent reception of some protocol message already sent to p within the e_1 part.
2) Similarly, a reception within $\mathbf{Rel}_p(e_1)$ might be disruptable by $\mathbf{T}_p(e_2)$.
3) It might happen that a non-starting participant p of e_2 already terminates $\mathbf{T}_p(e_1)>>\mathbf{Rel}_p(e_1)$ and thereby $\mathbf{T}_p(e)$, while for the starting participant p' of e_2 $\mathbf{T}_{p'}(e_2)$ is still executable, so that execution of the e_2 part actually starts, but is subsequently blocked because of non-co-operation of p.

To understand the first two problems, consider the following example: Let a server consist of PEs 1 and 2 and let

$e = e_1[>e_2 = (a^1;((b^1;\text{exit})|||(c^2;\text{exit})))[>(d^2;((e^1;\text{exit})|||(f^2;\text{exit})))$

be the behaviour that we are implementing. The protocol generated by the algorithm of [3] could be of the form

$\mathbf{T}_1(e) \approx (a^1;s_2(1);b^1;((s_2(2);\text{exit})|||(r_2(2);\text{exit})))[>(r_2(3);e^1;\text{exit})$

$\mathbf{T}_2(e) \approx (r_1(1);c^2;((s_1(2);\text{exit})|||(r_1(2);\text{exit})))[>(d^2;s_1(3);f^2;\text{exit})$

The scenario "$a^1;s_2(1);d^2;\ldots$" renders $r_1(1)$ non-executable (the first problem), while "$a^1;s_2(1);r_1(1);b^1;s_2(2);d^2;\ldots$" renders $r_1(2)$ non-executable (the second problem).

The first problem can be avoided by requiring $|AP(e_1)|=1$ [2], where attribute AP lists all the participants of a particular service part. Together with restrictions R2 and R3 in [3] that implies $\exists p': AP(e_1)=SP(e_2)=EP(e)=\{p'\}$ (localised decision-making) and also solves the second problem:

1) $\mathbf{Rel}_p(e_1)$ contains no reception.
2) At any other PE p, $\mathbf{Rel}_p(e_1)$ is a reception from p', and $\mathbf{T}_p(e_2)$ also starts by a reception from p'. As the two messages travel along the same FIFO channel, the message in $\mathbf{T}_p(e_2)$ never disrupts the message in $\mathbf{Rel}_p(e_1)$ that can only be sent as the first of the two.

However, the third problem remains, as demonstrated below. Let a server consist of PEs 1 and 2 and let

$e = e_1[>e_2 = (a^1;\text{exit})[>(b^1;c^2;d^1;\text{exit})$

be the behaviour that we are implementing. The protocol generated by the algorithm of [3] could be of the form

$\mathbf{T}_1(e) \approx (a^1;s_2(1);\text{exit})[>(b^1;s_2(2);r_2(3);d^1;\text{exit})$

$\mathbf{T}_2(e) \approx (r_1(1);\text{exit})[>(r_1(2);c^2;s_1(3);\text{exit})$

If e is followed by some other service part, exit of any $\mathbf{T}_p(e)$ (below denoted as "exit^p") is a local matter of p, thus the following scenario is executable: "$a^1;s_2(1);r_1(1);\text{exit}^2;b^1;s_2(2)$". We see that PE 2 receives message 1, indicating that e_2 will not be activated, and terminates. However, exit^1 that should follow the transmission of the message is disrupted by activation of $\mathbf{T}_1(e_2)$, resulting in a deadlock because of non-co-operation of PE 2 in the e_2 part. The deadlock even occurs if the capacity of the channel buffer is 1, as assumed in Section 5 of [3].

Obviously for the decision-making p' it is not acceptable to have $\mathbf{Rel}_p(e_1)$ on the left side of the "[>" operator. That is avoided in [2], where a solution is given for two-party systems

with synchronous channels, but probably also works for FIFO channels. For the last example, a protocol in lines with [2] would be

$\mathbf{T}_1(e) \approx ((a^1;exit)[>(b^1;s_2(2);r_2(3);d^1;exit))>>(s_2(1);exit)$

$\mathbf{T}_2(e) \approx (r_1(1);exit)[](r_1(2);c^2;s_1(3);r_1(1);exit)$

i.e. the decision-making PE 1 in all cases first terminates ($\mathbf{T}_1(e_1)[>\mathbf{T}_1(e_2)$) and only then proceeds to informing PE 2.

But even that solution is not satisfactory in all contexts. Suppose that the service part e runs in parallel with another service part e', to form an e'' as follows:

$e'' = e|[a^1,b^1]|e' = ((a^1;exit)[>(b^1;c^2;d^1;exit))|[a^1,b^1]|(a^1;b^1;c^2;d^1;exit)$

The service is non-blocking and its only executable scenario is "$a^1;b^1;c^2;d^1;exit$", for an exit immediately after a^1 in the e_1 part of e would require synchronisation with an exit in the e' part, that is at that point of service execution not available. The derived protocol could be

$\mathbf{T}_1(e'') \approx \mathbf{T}_1(e)|[a^1,b^1]|\mathbf{T}_1(e')$

$\approx (((a^1;exit)[>(b^1;s_2(2);r_2(3);d^1;exit))>>(s_2(1);exit))|[a^1,b^1]|(a^1;b^1;s_2(4);r_2(5);d^1;exit)$

$\mathbf{T}_2(e'') \approx \mathbf{T}_2(e)|||\mathbf{T}_2(e')$

$\approx ((r_1(1);exit)[](r_1(2);c^2;s_1(3);r_1(1);exit))|||(r_1(4);c^2;s_1(5);exit)$

Suppose that the distributed implementations of the two parallel service parts employ separate sets of protocol channels, to avoid the problems discussed in Section 2. There is a scenario "$a^1;s_2(1);\ r_1(1)$" leading to a deadlock caused by the fact that the e part decides on its own to terminate immediately after a^1, refusing to participate in b^1. The source of the problem is the protocol message 1 sent upon the termination of the e_1 part of e. If no such message was sent, $\mathbf{T}_1(e)$ would be of the form ($\mathbf{T}_1(e_1)[>\mathbf{T}_1(e_2)$), and hence its termination properly synchronised with termination of $\mathbf{T}_1(e')$.

[4] identifies the above problem as the problem of reporting terminations that are both decision-making for a particular service part e and synchronised with the environment of e. It

has been demonstrated (also for the multi-party case) that such reporting can often be avoided, thereby preventing errors of the above type.

4 Discussion and conclusions

[3] is an enhancement of the protocol derivation algorithm proposed in [5] by transformations for distributed implementation of synchronised parallel execution and of disabling. Unfortunately we have been able to show that none of the two transformations is correct in a general case, though the first one becomes nonproblematic if the underlying communication service is slightly adapted.

For implementation of disabling, alternative solutions are given in [2,4], but only for entirely local decision-making and, in the case of [2], for two-party systems only. Besides, the solution in [2] might be incorrect if termination of the considered service part *e* requires synchronisation with the environment of *e*.

The example of [6] gives hope that it would also be possible to precisely handle distributed decision-making, but then the protocol derivation transformation would be much less compositional, and consequently difficult to understand and to implement. Hence we conclude that the idea of [3] to give the disabling operator a semantics that is less rigorous, but more suitable for distributed systems, is good and worth pursuing, however, the proposed protocol derivation transformation should be amended to avoid the potential unspecified receptions and deadlocks pointed out above or, if complete correction is not possible, the limits of its applicability should be more clearly stated.

References

1. Bolognesi T, Brinksma E: Introduction to the ISO specification language LOTOS. Comput Networks ISDN Syst 14(1):25-59 (1987)

2. Brinksma E, Langerak R: Functionality decomposition by compositional correctness preserving transformation. SACJ/SART 13:2-13 (1995)
3. Kant C, Higashino T, Bochmann Gv: Deriving protocol specifications from service specifications written in LOTOS. Distrib Comput 10:29-47 (1996)
4. Kapus-Kolar M: Employing disruptions for more efficient functionality decomposition in LOTOS. Submitted for publication; an extended version available as Jožef Stefan Institute Technical Report 7878, 1998; a preliminary version in Proc. EUROMICRO 97, IEEE Computer Society, Los Alamitos Washington Brussels Tokyo 1997, pp 464-471
5. Khendek F, Bochmann Gv, Kant C: New results on deriving protocol specifications from service specifications. In Proc. ACM SIGCOMM'89, pp 136-145, 1989
6. Langerak R: Decomposition of functionality: a correctness-preserving LOTOS transformation. In: Logrippo L, Probert RL, Ural H (eds) Protocol specification, testing, and verification, X. North-Holland, Amsterdam NewYork Oxford Tokyo 1990, pp 229-242