Protocol Synthesis from Context-Free Processes using Event Structures

Akio Nakata
Department of Computer Science
Hiroshima City University
3-4-1, Ozuka-higashi, Asaminami-ku,
Hiroshima 731-3194, Japan
nakata@cs.hiroshima-cu.ac.jp

Teruo Higashino Kenichi Taniguchi
Department of Information and Computer Sciences,
Osaka University
1-3, Machikaneyama-cho, Toyonaka,
Osaka 560-8531, Japan
{higashino,taniguchi}@ics.es.osaka-u.ac.jp

Abstract

In this paper, we propose a protocol synthesis method based on a partial order model (called event structures) for the class of context-free processes. First, we assign a unique name called event ID to every event executable by a given service specification. An event ID is a finite sequence of symbols derived from the context-free process specification. Then we show that some interesting sets of events are expressible by regular expressions on symbols, and that the event structure can be finitely represented by a set of relations among the regular expressions. Finally, we present a method to derive a protocol specification which implements a given service specification on distributed nodes, by using the obtained finite representation of event structures. The derived protocol specification contains the minimum message exchanges necessary to ensure the partial order of events of the service specification.

keywords: *formal description techniques, distributed systems, protocol synthesis, context-free processes, event structures*

1. Introduction

Protocol synthesis problem[14] is a problem to derive automatically a set of specifications (called *protocol specification*) containing message exchanges among computers (called *nodes*) connected with networks, from a given *service specification* of a distributed system. If we assume that there are no errors nor message lossage in communication channels, the protocol synthesis problem is usually reduced to the problem to guarantee partial order of events in the service when executed concurrently on all distributed nodes. Many solutions of the protocol synthesis problem have been proposed so far for various specification models and under various assumptions [2, 3, 4, 5, 7, 9, 11, 12, 15]. Especially, the protocol synthesis method [7, 9] for the specification language LOTOS[8] is useful because it is very expressive and practical. LOTOS has an expressive power to describe the class of processes called *context-free processes,* which is more expressive than FSMs.

However, the existing protocol synthesis method for LOTOS have some problems as follows. The method in [7] can be only applied to finite state LOTOS specifications. The method in [9] can be applied to the class of context-free processes. But it imposes several strong restrictions on service specification. Moreover, the derived protocol specification may contain some redundant message exchanges. If we can analyze the global partial order structure of events and identify the preceding/succeeding events of every event, we can derive a protocol specification which contains only the message exchanges necessary to guarantee the partial order of events.

In this paper, we propose a method to obtain a finite representation of *event structures*[10, 13, 16] from context-free process specifications. And then, we propose a protocol synthesis method using the finite representation of event structures. Here, an *event* is an occurrence of an action in the execution sequence of a process. For example, in the execution sequence "abac", the first occurrence of the action "a" and its second occurrence are distinguished events. An *event structure,* which is a well-studied concurrent process model based on partial order, is a set of a causal relation and a conflict relation defined on all such events executable by a process.

In this paper, first we assign a unique name (an *event ID*) to each event executable by a context-free process. An event ID is generally a finite sequence of symbols derived automatically from the syntactical tree of the context-free process specification. Then, we show that some interesting sets of event IDs are regular languages, and that we can derive automatically the regular expressions of the sets of event IDs. Using the regular expressions, we can obtain a finite representation of the event structure of the given context-free process specification automatically. We refer to it as a *symbolic event structure*. Finally, we propose a protocol synthesis method using symbolic event structures. We also give a tiny example of context-free processes in order to demonstrate our method.

The rest of this paper is organized as follows. In Section 2, we define the class of the specification language, context-free processes. In Section 3, we propose a method to assign a unique event ID to each event using the syntactical tree of a process specification. We also show that some interesting sets of event IDs are regular. In Section 4, we introduce a notion of a symbolic event structure. Also we show that it is automatically derivable from a context-free process specification. In Section 5, we propose a protocol synthesis method using symbolic event structures. Section 6 concludes this paper.

2. Context-Free Processes

In this section, we define the syntax and semantics of context-free process specifications.

Definition 1 The syntax of *behaviour expressions* of *context-free processes* is defined by the following BNF.

$$B ::= stop | exit | (B) | P | a; B | B [] B | B >> B$$

where, $a \in Act$ is an action name and Act is a set of an action name. We denote a set of behaviour expressions by Bex. □

Definition 2 A *context-free process specification Spec* is a 3-tuple $\langle Procs, S, Defs\rangle$, where $Procs$ is a finite set of process names, S is an *initial process name*, and $Defs \overset{\text{def}}{=} \{(P, B) | P \in Procs, B \in Bex\}$ is a finite mapping from each process name to a behaviour expression. We call $(P, B) \in Defs$ as a *process definition*, denoted by $P := B$.

The intuitive meaning of behaviour expressions is as follows. The behaviour expression $stop$ is a process which does nothing. $exit$ is a process which just executes an action δ, which represents a *successful termination*, and then stops. $a; B$ is a process which executes an action a and then behaves like B. $B_1 [] B_2$ is a process which behaves either B_1 or B_2. The choice of B_1 or B_2 is made when executing the first action. $B_1 >> B_2$ is a process which behaves like B_1 until B_1 successfully terminates, that is, until B_1 executed δ, and then it behaves like B_2. P is a process which behaves like the right hand B of the process definition $P := B$. Finally, the behaviour of a context-free process specification $Spec$ is the same as the behaviour expression S, where S is an initial process name of $Spec$.

Formally, the structured operational semantics of behaviour expressions of context-free process specifications is given as Figure 1.

3. Event Identifiers

In this section, we introduce the notion of *event identifiers* (abbreviated as *event IDs*). To define an event ID, we need two notions, an *occurrence position* and a *process invocation stack*. Our intention is to use the first one in order to identify the syntactical (static) occurrence of each event (or each subexpression), and the latter one in order to identify the dynamic occurrence of each event. The main idea of an event ID is that if the above two notions of occurrence are combined, we can completely identify any event executed by the possibly recursive process. The notion of an event ID is inspired by both [1] and [10].

Definition 3 For each process definition $P := B$ of a context-free process specification, let $T(B)$ be the syntactical tree of the behaviour expression B. Then we represent a path from the root node of $T(B)$ to any other node by $(P :=, {op_1}_{d_1}, {op_2}_{d_2}, \ldots, {op_{n-1}}_{d_{n-1}}, op_n)$, where for each $k \in \{1, \ldots, n-1\}, op_k \in \{[], ;, >>\}, d_k \in \{L, R\}$ and for $k = n, op_n \in \{[], ;, >>\} \cup Act \cup \{stop, exit\}$. ${op_k}_L$ $[{op_k}_R]$ means that the left [right] subtree of the subtree whose root node is an operator op_k is traversed. We refer to the path as an *occurrence position* of the subexpression whose root node of the corresponding subtree is op_n in the process P. □

For example, consider a process definition $P := (a; Q)$ $[] ((b; exit) >> (a; (d; exit)))$. The syntactical tree of this process definition is illustrated in Fig. 2. The occurrence position of the subexpression $(d; exit)$ in P is represented by the path $(P :=, []_R, >>_R, ;_R, ;)$. Also, the occurrence position of the process invocation Q is represented by $(P :=, []_L, ;_R, Q)$. The occurrence positions of the action name a are $(P :=, []_L, ;_L, a)$ and $(P :=, []_R, >>_R, ;_L, a)$. In the rest of this paper, we may assign a unique number to each occurrence position and abbreviate it to the number.

Next, we will define the notion of a *process invocation stack*. Intuitively, a process invocation stack holds the information about which process (say process A) has invoked the current running process (at which occurrence position), and which process (say process B) has invoked the process A, and so on.

Definition 4 A *process invocation stack* is a sequence of occurrence positions of process invocations which is in the form of

$$(S :=, \ldots, P_1)(P_1 :=, \ldots, P_2) \ldots$$
$$(P_{k-2} :=, \ldots, P_{k-1})(P_{k-1} :=, \ldots, P_k).$$

(There are possibly duplicate process names in $S, P_1, \ldots, P_k$.)

An *event ID* of an action a [or $exit$] is a sequence of occurrence position such that an occurrence position of a

$$\frac{true}{exit \xrightarrow{\delta} stop} \qquad \frac{a \in Act}{a; B \xrightarrow{a} B}$$

$$\frac{B_1 \xrightarrow{\alpha} B'}{B_1 [] B_2 \xrightarrow{\alpha} B'} \qquad \frac{B_2 \xrightarrow{\alpha} B'}{B_1 [] B_2 \xrightarrow{\alpha} B'}$$

$$\frac{B_1 \xrightarrow{\alpha} B' \quad \alpha \neq \delta}{B_1 >> B_2 \xrightarrow{\alpha} B' >> B_2} \qquad \frac{B_1 \xrightarrow{\delta} B'}{B_1 >> B_2 \xrightarrow{i} B_2}$$

$$\frac{B \xrightarrow{\alpha} B' \quad P := B \in Defs \quad \alpha \in Act \cup \{i, \delta\}}{P \xrightarrow{\alpha} B'}$$

Figure 1. Structured Operational Semantics for Context-Free Processes

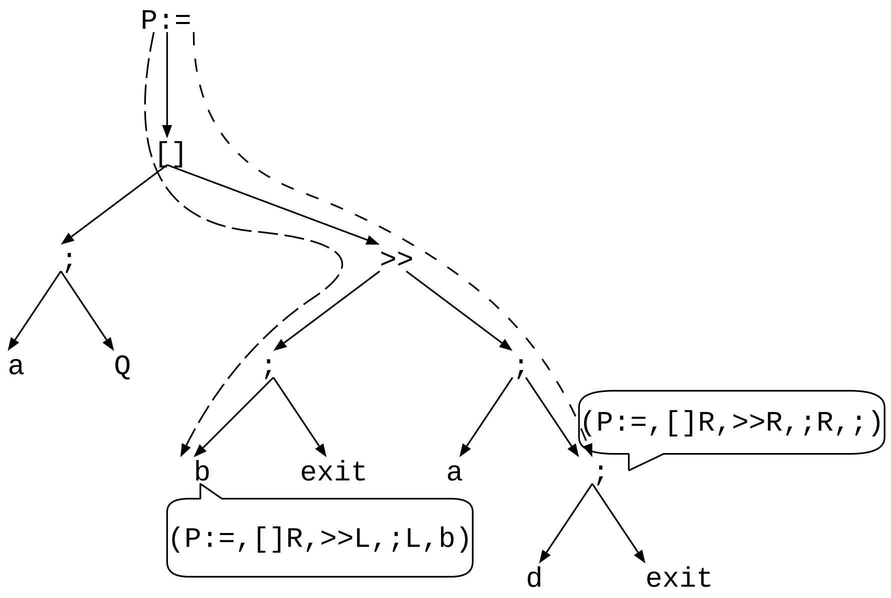


Figure 2. Syntactical Tree and Occurrence Position

[$exit$, resp.] in the process P_k is appended in the last of a process invocation stack. □

Example 1 The followings are some examples of event IDs.

- $(S :=, []_L, >>_R, ;_L, a)$
- $(S :=, >>_R, P)(P :=, []_R, ;_R, exit)$
- $(S :=, \ldots, P)(P :=, \ldots, Q)(Q :=, \ldots, P)(P :=, \ldots, a)$

Note that the second event ID represents the action $exit$ executed by the process P, which has been invoked by the process S at the right hand of the operator $>>$. The third event ID represents the action a, where the initial process S has invoked the process P, P has invoked the process Q, Q has invoked P (P and Q are mutually recursive processes), and finally P executed the action a. □

Example 2 For the context-free process specification

$$\begin{aligned} Spec \;=\; & \langle \{S, P, Q\}, S, \\ & \{S := a; P, \\ & \;P := (b; Q) >> (a; P), \\ & \;Q := (c; P)[](d; exit)\}\rangle, \end{aligned}$$

the following is a sequence of event IDs executable by

Spec:

$(S :=, ;_L, a)$,
$(S :=, ;_R, P)(P :=, >>_L, ;_L, b)$,
$(S :=, ;_R, P)(P :=, >>_L, ;_R, Q)(Q :=, []_R, ;_L, d)$,
$(S :=, ;_R, P)(P :=, >>_L, ;_R, Q)(Q :=, []_R, ;_R, exit)$,
$(S :=, ;_R, P)(P :=, >>_R, ;_L, a)$,
$(S :=, ;_R, P)(P :=, >>_R, ;_R, P)(P :=, >>_L, ;_L, b)$,
$\ldots$

Observe that the first and the second occurrences of the action b have different event IDs, since the process invocation stacks are different. □

For a set of event IDs of a context-free process specification, the following holds.

Theorem 1 A set of event IDs of every context-free process specification *Spec* is a regular language, where its alphabet is a set of occurrence positions.

[Proof sketch] From the following context-free process specification

$$\begin{array}{l} P_1 := \ldots \underset{k}{P_i} \ldots \underset{l}{P_j} \\ \ldots \\ P_i := \underset{m}{a}; \ldots \end{array}$$

where P_1 is an initial process name, k and l are occurrence positions of process invocations P_i and P_j in P_1, respectively, and m is an occurrence position of the action name a in P_i, construct a deterministic finite automaton (DFA) as follows:

$$\begin{array}{ccc} P_1 & -k-> & P_i \\ P_1 & -l-> & P_j \\ & \ldots & \\ P_i & -m-> & a \end{array}$$

In this DFA, the set of states are the set of process names, the set of input symbols are the set of occurrence positions, the initial state is the initial process name, and the set of final states are the set of action names. Clearly, this DFA accepts exactly all event IDs of events executable by the process P_1. □

In the next definition we define two interesting sets of event IDs.

Definition 5 For any occurrence position p of any subexpression B of any context-free process specification, we call a set of event IDs of the first [the last] executable events of B as a *set of starting events* [a *set of ending events*, resp.], denoted by $SE(p)$ [$EE(p)$, resp.]. □

From Theorem 1, immediately the following corollary holds.

Corollary 1 $SE(p)$ and $EE(p)$ are regular.

[Proof sketch] Similar to Theorem 1. The main idea of the proof is that we only allow occurrence positions which do not contain any $;_R$'s and $>>_R$'s (in case of $SE(p)$) as input symbols when constructing the DFA. The details are omitted. □

4. Symbolic Event Structures

In this section, we define symbolic event structures formally.

Definition 6 Let c denote a variable ranges over process invocation stacks, and let r and r' denote regular expressions whose alphabet is a set of occurrence positions. Let "." denote the concatenation operator for any two sequences and/or regular expressions of occurrence positions. Let the (extended) regular expression $c.r$ denote a set of event IDs whose suffixes match the regular expression r (the rest of the sequence is assigned to the variable c). A *symbolic event structure* is a pair of relations R_1 and R_2, where

$$R_i = \{(c.r, c.r') | c \text{ is a variable, } r \text{ and } r' \text{ are regular expressions.}\}$$

for each $i \in \{1, 2\}$. Let

$$c.r \longrightarrow c.r'$$

denote $(c.r, c.r') \in R_1$ and we refer to it as a *symbolic causal relation*. And let

$$c.r \rightarrow\leftarrow c.r'$$

denote $(c.r, c.r') \in R_2$ and we refer to it as a *symbolic conflict relation*. □

The intuitive meanings of symbolic causal/conflict relations are as follows. A symbolic causal relation $c.r \longrightarrow c.r'$ means that any event whose event ID e' matches $c.r'$ must be executed just after some event whose event ID e matches $c.r$. Note that the two event IDs e and e' have the same prefix c. A symbolic conflict relation $c.r \rightarrow\leftarrow c.r'$ means that any event whose event ID matches $c.r$ and any event whose event ID matches $c.r'$ must be exclusively executed.

We can automatically derive a symbolic event structure for any context-free process specification *Spec*. Specifically, we can derive either a causal relation or a conflict relation for each occurrence of the operators ;, [] and $>>$ as follows.

For the case $P := \ldots(a;B)\ldots$, that is, for the occurrence of the operator ; in the process definition of P, we derive a causal relation

$$\begin{array}{l} c.(P :=, \ldots, ;_L, a) \\ \quad \longrightarrow c.\mathit{Rex}(SE((P :=, \ldots, ;_R, B))), \end{array}$$

where $\mathit{Rex}(E)$ denotes a regular expression of a regular set E, and c is a variable which matches any process invocation stack ending with an occurrence position in the form of $(\ldots, P)$.

Similarly, for the case $P := \ldots(B[]C)\ldots$, we derive

$$\begin{array}{l} c.\mathit{Rex}(SE((P :=, \ldots, []_L, B))) \\ \quad \rightarrow\!\leftarrow c.\mathit{Rex}(SE((P :=, \ldots, []_R, C))), \end{array}$$

and for the case $P := \ldots(B >> C)\ldots$, we derive

$$\begin{array}{l} c.\mathit{Rex}(EE((P :=, \ldots, >>_L, B))) \\ \quad \longrightarrow c.\mathit{Rex}(SE((P :=, \ldots, >>_R, C))). \end{array}$$

From Corollary 1, we can construct DFAs automatically which accept $SE()$ and $EE()$ and derive automatically the corresponding regular expressions from the DFAs[6].

Example 3 (Distributed Stack) Fig. 3 is an example of a context-free process specification of a simple (but infinite state) service which inputs data x from node A by the action `push^A?x`, pushes x on the top of the stack, and retrieves the data from the top of the stack and outputs to the node B by the action `pop^B!x`. If the last data in the stack has been retrieved, the process successfully terminates. It is specified that both `pop^B!x` and the next push action `push^A?y` are always executable, except the first action `push^A?x`.

Let $SE(P, k_L)$ $[SE(P, k_R)]$ denote the set of the first events of the left [right, resp.] subexpression of the occurrence position k. Similarly, let $EE(P, k_L)$ $[EE(P, k_R)]$ denote the set of the ending events of the left [right, resp.] subexpression of k.

The event structure specified by each occurrence of the operators is expressed completely by the following relations.

[Causal relations]

$$\begin{array}{rcl} c.\mathit{Rex}(EE(S, 8_L)) & \longrightarrow & c.\mathit{Rex}(SE(S, 8_R)) \\ c.\mathit{Rex}(EE(P, 9_L)) & \longrightarrow & c.\mathit{Rex}(SE(P, 9_R)) \\ c.\mathit{Rex}(EE(P, 11_L)) & \longrightarrow & c.\mathit{Rex}(SE(P, 11_R)) \\ c.\mathit{Rex}(EE(P, 12_L)) & \longrightarrow & c.\mathit{Rex}(SE(P, 12_R)) \end{array}$$

[Conflict relations]

$$c.\mathit{Rex}(SE(P, 10_L)) \quad \rightarrow\!\leftarrow \quad c.\mathit{Rex}(SE(P, 10_R))$$

Clearly, $\mathit{Rex}(EE(P, 8_L)) = 1$ and $\mathit{Rex}(SE(P, 8_R)) = 2(3|5)$ hold (here, $(r|r')$ denote a union operator of regular expressions, i.e., if the sequence matches either r or r', then it matches $(r|r')$).

$\mathit{Rex}(EE(P, 12_L)) = 6.\mathit{Rex}(EE(P))$ holds. A DFA which accepts $EE(P)$ is constructed as $M =< \{P, \mathit{exit}\}, \Sigma, \{P-4->\mathit{exit}, P-7->P\}, P, \{\mathit{exit}\} >$ (Σ is a set of occurrence positions each of which does not contain $;_L$ and $>>_L$). From the DFA, the corresponding regular expression is constructed as $\mathit{Rex}(EE(P)) = (7)^*4$ (here, $(r)^*$ denotes Kleene's star closure, i.e., it represents 0 or more repetition of the sequences which match the regular expression r). Hence, $\mathit{Rex}(EE(P, 12_L)) = 6(7)^*4$ holds. Other regular expressions are obtained similarly.

Finally, the following symbolic event structure is obtained.

$$\begin{array}{rcl} c.1 & \longrightarrow & c.2(3|5) \\ c.3 & \longrightarrow & c.4 \\ c.5 & \longrightarrow & c.6(3|5) \\ c.6(7)^*4 & \longrightarrow & c.7(3|5) \\ c.3 & \rightarrow\!\leftarrow & c.5 \end{array}$$

□

5. Protocol Synthesis

In this section, we describe a protocol synthesis method using the notion of symbolic event structures.

For a given service specification written in context-free process specification, we derive a specification of each node under the following policy:

1. Basically, a specification of node i is a projection $\mathit{Proj}_i(S)$ onto node i of the service specification S, except that some necessary message exchanging actions and some auxiliary data manipulations are added.

2. For each process definition $P := B$ of the service specification, we derive $P_i(c) := \mathit{Proj}_i(B)$ for each node i. c is a variable which represents a process invocation stack.

3. For each process invocation P of the service specification, we derive $P_i(c.k)$ for each node i. k is an occurrence position of P in the service specification.

4. For each subexpression $a;B$, we derive $\mathit{Pre} >> a;\mathit{Post} >> \mathit{Proj}_i(B)$ if the action a is assigned to node i, otherwise we derive $\mathit{Proj}_i(B)$, for each node i. Here, the process Pre receives the preceding event IDs (if any) of the event ID of a. The preceding event ID of the same node is not received. Regular expression of the preceding event IDs is chosen by looking

```
            8
 S:=push^A?x;P(x)
    1        2


               9      10          11      12
 P(x):=(pop^B!x;exit) [] ((push^A?y;P(y)) >> P(x))
        3       4          5        6        7

[Occurrence positions]
actions:  1,3,4,5
process invocations: 2,6,7
operators: 8,9,10,11,12
```

Figure 3. Example of Service Specification

some alphabets(occurrence positions) from the top of the process invocation stack c.

The process $Post$ sends the event ID $c.p$ of the action a to the other nodes (if any) that is waiting for receiving this event ID.

5. For each subexpression $exit$, we derive $Pre >> Post >> exit$, where the process $Post$ sends the event ID of $exit$ if the preceding event is assigned to node i. Pre receives the preceding event IDs of $exit$.

6. For each subexpression $B >> B'$, we derive $Proj_i(B) >> Proj_i(B')$.

7. For each subexpression $B[]B'$, if the conflicting first events $e \in SE(B)$, $e \in SE(B')$ in B and B', respectively, are assigned to the different nodes, we derive the specification for each node as follows. Specifically, using the auxiliary processes $HaveToken_i(c.p, B)$ and $WaitToken_j(c.p, B')$, we derive

$$HaveToken_j(c.p, Proj_i(B))[]Proj_i(B')$$

for the specification of node i, and

$$Proj_j(B)[]WaitToken_i(c.p, Proj_j(B'))$$

for the specification of node j. Here, c is a process invocation stack and p is an occurrence position of the operator []. Without loss of generality, we assume that $SE(B')$ [$SE(B)$] contains no event assigned to node i [node j, respectively] [1].

$HaveToken_j(c.p, B)$ is a process which chooses locally at node i either to send an sequence cp to node j as a token and then invoke $WaitToken_j(c.p, B)$, or to execute B. $WaitToken_i(c.p, B')$ is a process which waits for receiving a token $c.p$ from node i and then invoke $HaveToken_i(c.p, B')$.

[1]Otherwise, B' can be decomposed into $B''[]B_i$, where $SE(B'')$ contains no event assigned to node i and $SE(B_i)$ contains only events assigned to node i. In this case, we consider $(B[]B_i)[]B''$ instead of $B[]B'$.

Example 4 For the service specification in Example 3, we can derive a protocol specification as follows. First, we assign each event to either node A or B as follows.

```
node A  :  c.1,c.5
node B  :  c.3,c.4
```
where $\mathtt{c} = (2|6|7)^*$

According the policy as above, we derive the protocol specification shown in Fig. 4. In Fig. 4, `send_i(c.p)` is an action which sends the event ID $c.p$ to the node i, and `recv_i(c.r)` is an action which receives any event ID which matches the regular expression $c.r$. In this example, we also use `send_i(c.p,x)` and `recv_i(c.r,x)`, which means sending data x with event ID $c.p$, and receiving data x with event ID matching $c.r$, respectively. Note that the data part of the specification is not automatically derived. We manually added the actions `send_i(c.p,x)` or `recv_i(c.r,x)`, respectively, to the derived protocol specification.

We also use a construct "if $[condition1]$ $then$ $action1$ $else$ $if \ldots$" in the following. We omit the precise definition here because the meaning of this construct is obvious. □

6. Conclusion

In this paper, we proposed a protocol synthesis method for the class of context-free processes, by using the notion of symbolic event structures. Our method is applicable to not only the class of FSMs, but also the non-finite-state service specification which can be specified in context-free processes. In comparison with [9], our method is better in message complexity. For example, consider the service specification such as below.

```
[A]
S_A(c):=push^A?x;send_B(c.1,x);P_A(x,c.2)
P_A(x,c):=(if c=c'.7 then recv_B(c'.6(7)*4));
            (recv_B(c.3);exit
            [] HaveToken_B(c.10,
                push^A?y;send_B(c.5,y);P_A(y,c.6) >> P_A(x,c.7)))

[B]
S_B(c):=P_B(x,c.2)
P_B(x,c):=(if c=c'.2 then recv_A(c'.1,x)
           else if c=c'.6 then recv_A(c'.5,x));
            (WaitToken_A(c.10, pop^B!x;send_A(c.3);send_A(c.4);exit)
            [] P_B(y,c.6) >> P_B(x,c.7))

where  for i in {A,B}, p : occurrence position,
       P: behaviour expression
  HaveToken_i(c.p,P):=P[]send_i(c.p);WaitToken_i(c.p,P)
  WaitToken_i(c.p,P):=recv_i(c.p);HaveToken_i(c.p,P)
```

Figure 4. Protocol Specification for Example 3

```
S:= (a^1;b^2;c^3;exit
    [] d^1;e^3;exit) >> (f^2;exit)
```

Note that `a^i` means that the action `a` must be executed at node i. The protocol specifications derived by [9] and our method are shown in Fig. 5. This specification has two alternative traces, $t_1 = (\texttt{a\^{}1}, \texttt{b\^{}2}, \texttt{c\^{}3}, \texttt{f\^{}2})$ and $t_2 = (\texttt{d\^{}1}, \texttt{e\^{}3}, \texttt{f\^{}2})$. For trace t_1, the number of exchanged messages by the two protocol specifications are 3 ([9]) and 3 (our method). However, for trace t_2, the number of exchanged messages by the two protocol specifications are 3 ([9]) and 2 (our method). In this case, the method of [9] derives one redundant message exchange and our method eliminates it. In general, in our method, message exchanges in the derived protocol specification only appear if and only if those are essentially necessary, i.e. there is a causal relation between the current event and the possible next events and some of the next events must be executed in different node. Therefore, theoretically no redundant message exchanges are introduced in our method.

The future work is to extend our method to deal with parallel composition, data inputs/outputs, and timing constraints.

References

[1] A. Bianchi, S. Coluccini, P. Degano, and C. Priami. An efficient verifier of truly concurrent properties. In *Proc. of 3rd Int'l Conf. on Parallel Computing Technologies (PaCT-95)*, volume 964 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, Sept. 1995.

[2] G. v. Bochmann and R. Gotzhein. Deriving protocol specification from service specifications. In *Proc. of the ACM SIGCOMM '86 Symposium*, pages 148–156, Vermont, USA, 1986.

[3] P. M. Chu and M. T. Liu. Protocol synthesis in a state transition model. In *Proc. IEEE COMPSAC '88*, pages 505–512, 1988.

[4] R. Gotzhein and G. v. Bochmann. Deriving protocol specifications from service specifications including parameters. *ACM Trans. on Computer Systems*, 8(4):255–283, 1990.

[5] T. Higashino, K. Okano, H. Imajo, and K. Taniguchi. Deriving protocol specifications from service specifications in extended FSM models. In *Proc. of the 13th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS-13)*, pages 141–148, 1993.

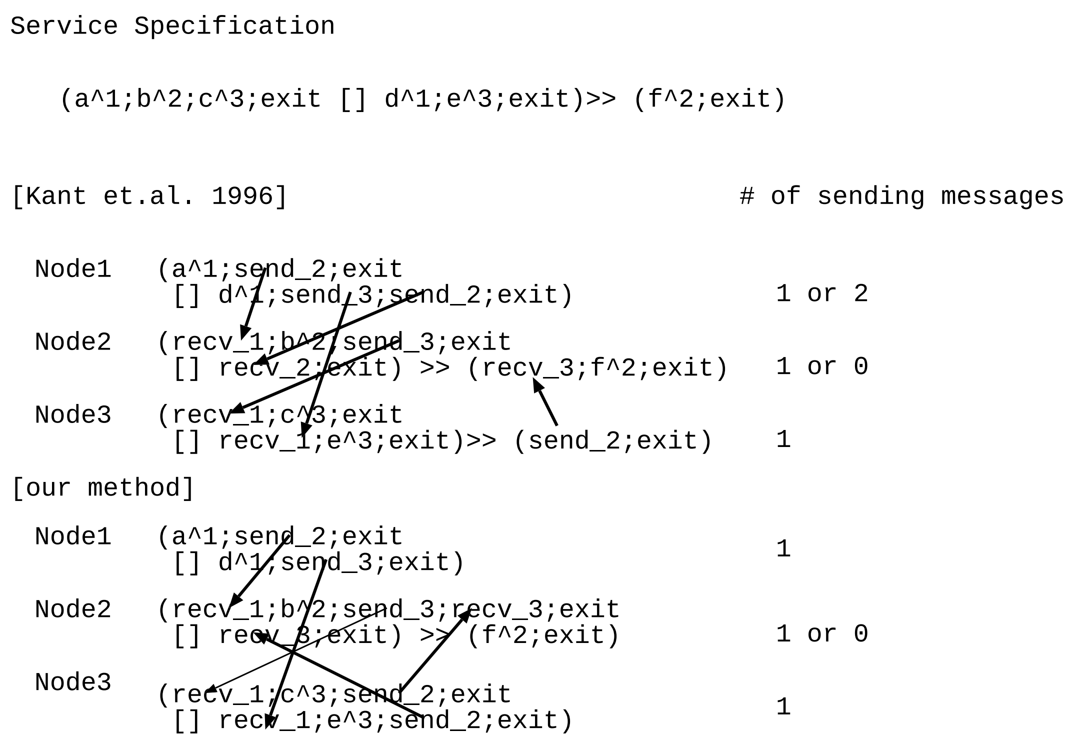
[6] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[7] M. Hultström. Structural decomposition. In S. T. Vuong and S. T. Chanson, editors, *Protocol Specification, Testing and Verification, XIV*, pages 201–216. IFIP, Chapman & Hall, 1995.

[8] ISO. *Information Processing System, Open Systems Interconnection, LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. IS 8807, Jan. 1989.

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

[10] J.-P. Katoen. *Quantitative and Qualitative Extensions of Event Structures.* CTIT Ph.D-thesis series No.96-09, Centre for Telematics and Information Technology, Enshede, The Netherlands, Sept. 1996.



```
Service Specification

   (a^1;b^2;c^3;exit [] d^1;e^3;exit)>> (f^2;exit)


[Kant et.al. 1996]                                # of sending messages

  Node1   (a^1;send_2;exit
           [] d^1;send_3;send_2;exit)                 1 or 2

  Node2   (recv_1;b^2;send_3;exit
           [] recv_2;exit) >> (recv_3;f^2;exit)       1 or 0

  Node3   (recv_1;c^3;exit
           [] recv_1;e^3;exit)>> (send_2;exit)        1

[our method]

  Node1   (a^1;send_2;exit
           [] d^1;send_3;exit)                        1

  Node2   (recv_1;b^2;send_3;recv_3;exit
           [] recv_3;exit) >> (f^2;exit)              1 or 0

  Node3
          (recv_1;c^3;send_2;exit
           [] recv_1;e^3;send_2;exit)                 1
```

Figure 5. Comparison between [9] and ours.

[11] F. Khendek, G. v. Bochmann, and C. Kant. New results on deriving protocol specifications from services specifications. In *Proc. of the ACM SIGCOMM '89*, pages 136–145, 1989.

[12] R. Langerak. Decomposition of functionality; a correctness-preserving LOTOS transformation. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Protocol Specification, Testing and Verification, X (PSTV-X)*, pages 229–242. IFIP, Elsevier Science Publishers B.V.(North-Holland), 1990.

[13] R. Langerak. Bundle event structures: a non-interleaving semantics for LOTOS. In M. Diaz and R. Groz, editors, *Formal Description Techniques, V*, pages 331–346. IFIP, Elsevier Science Publishers B.V.(North-Holland), 1993.

[14] R. L. Probert and K. Saleh. Synthesis of communication protocols: Survey and assessment. *IEEE Trans. Comput.*, 40(4):468–475, Apr. 1991.

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

[16] G. Winskel. Event structures. In J. de Bakker, P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 325–392. Springer-Verlag, 1989.