PROSPEX: A Graphical LOTOS Simulator for Protocol Specifications with N Nodes

Keiichi YASUMOTO¹, Nonmember, Teruo HIGASHINO¹, Toshio MATSUURA¹, and Kenichi TANIGUCHI¹, Members

SUMMARY In LOTOS, requirements for a distributed system are described as a service definition. On the protocol level, each node(protocol entity) must exchange some data values and synchronization messages to provide a service described in a service definition. The tuple of the specifications of all nodes in the system which provide the service is called as a protocol specification. In order to develop the communication programs satisfying a given service definition, it is very important to develop the correct protocol specification. For this purpose, the simulation of protocol specifications is useful and it is desirable that the designer can observe how a protocol specification is executed in parallel and how synchronization messages are exchanged among the nodes. Therefore, we have developed a new tool named PROSPEX. For a given pair of a service definition and a protocol specification, it executes the protocol specification in parallel and shows its execution process graphically on X Window System. If the protocol specification executes an event sequence which does not satisfy the service definition, then PROSPEX informs it to the designer. In this paper, the design and usefulness of PROSPEX are described.

key words: LOTOS, Service definition, Protocol specification, Simulator; Correctness

1. Introduction

LOTOS [1] is a language developed within ISO for the formal description of communication protocols and distributed systems. Recently, the specifications of many OSI protocols are described in LOTOS [2]. In LOTOS, service primitives of each node(protocol entity) in a distributed system are called events, and the temporal ordering of the execution of the events in the system are described as a service definition. Service definitions are also called service specifications [3,7,22]. On the protocol level, several nodes cooperate to provide the required service. They exchange data values and synchronization messages to ensure the temporal ordering of the execution of the events through a communication medium. The temporal ordering of the execution of the events containing the sending and/or receiving actions of data values and synchronization messages is described as the specification of a node(protocol entity). The tuple of the specifications of all nodes is called a protocol specification [4,7,22]. By executing the protocol specification in parallel, the service required in a service definition is provided. If a protocol specification is observational equivalent [1] to a given service definition, then we say that the protocol specification is correct with respect to the service definition.

According to a given service definition, the designer must develop the communication program which runs at each node. Since it is complicated to develop such programs from the service definition directly, the designer usually develops such programs according to the protocol specifications. Therefore, we need correct protocol specifications.

In order to develop correct protocol specifications, there are two techniques: (1) analysis and (2) synthesis. Some analysis techniques [4,5] have been proposed to determine whether given two specifications are observational equivalent although there are some restrictions. Using these techniques, we can determine whether the protocol specification given by a designer is correct or not with respect to the service definition. But, the correct protocol specification must be found by the designer. Usually, it can be obtained by trial and error. On the other hand, some synthesis techniques have been also proposed to derive correct protocol specifications automatically from given service definitions [6,7,8,9]. Although these synthesis techniques are promising, there are still many restrictions for the class of service definitions and most of them do not treat the data types of LOTOS specifications. They may also generate redundant sending/receiving actions of synchronization messages for some specific examples. Therefore, in general, we need modify the derived protocol specifications even if we use the synthesis techniques.

To use support tools for the simulation and analysis of the service definitions and protocol specifications is essential when we use the above techniques. It is desirable that the designer can observe how a protocol specification is executed in parallel and check whether it executes an event sequence which satisfies the service definition. Therefore, we have developed a new LOTOS tool named PROSPEX (PROtocol SPecification EXecutor) for the simulation and analysis of protocol specifications [10].

In PROSPEX, if a service definition and a protocol specification with N nodes are given, then PROSPEX generates a monitor S₀ and N simulators S₁, S₂, . . . , S_{N–1} and S_N. Each simulator S_i (1 ≤ i ≤ N) executes the specification of the i-th node and exchanges data values and synchronization messages each other. In order that the designer may understand the execution process of the nodes easily, each simulator draws the syntax tree of the current behaviour expression of a given node graphically on a display. For analysis of the protocol specification, the monitor S₀ shows some warning messages based on the information from the simulators S₁, . . . , S_{N–1} and S_N when the temporal ordering of the execution of the protocol specification does not satisfy
the service definition.

For efficient execution of LOTOS programs, the compilers such as Ref.[20],[21] are useful. For analyzing LOTOS specifications, the simulators are useful. Although some LOTOS simulators have been proposed [11],[12],[13],[14], these simulators are usually used for simulating a LOTOS specification. On the other hand, our PROSPEX can simulate more than one LOTOS specification simultaneously and monitor whether their executing processes satisfy the service definition. These facilities are new. Our PROSPEX can be also used as a graphical simulator for distributed algorithms. Since PROSPEX supports Full LOTOS functionality, it can treat the abstract data types which are described in an algebraic specification language ACT ONE [16].

In Section 2, we explain the service definitions and protocol specifications and give the formal definition of the correctness of the protocol specification. In Section 3, the facilities of PROSPEX are summarized. In Section 4, we describe the details of the design and implementation of PROSPEX. In this paper, we assume that the readers are familiar with LOTOS. For the details of LOTOS, refer to Ref.[1], [2], [15].

2. Service Definition and Its Correct Protocol Specification

In this section, first, we explain the difference between a service definition and the corresponding protocol specification definitely. Then, we give the formal definition of the correctness of the protocol specification.

2.1 Service Definition

Even if several nodes cooperate to provide a service, we do not describe the exchange of the synchronization messages and data values in a service definition.

For explanation, we will use Janken Game whose rule is described in Appendix A. Suppose that we play Janken Game in a distributed system (Fig.1).

<table>
<thead>
<tr>
<th>Board J</th>
<th>Player A</th>
<th>Player B</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

There are a Board J and two players A and B. In Fig.1, we assume that there are three nodes 1, 2 and 3. These nodes correspond to the Board J and two players A and B, respectively. They have the gates “j”, “a” and “b”, respectively.

First, the node 1 shows the “start” of the game to the Board J(gate “j”). Then two players give their choices at the gates “a” and “b”, respectively. The system decides the winner and shows it to the Board J. If the game is a tie, then the string “tie” is shown to the Board J and the game is carried out again. If either two players A or B wins the game, then the strings “win” and “loss” are shown to the gates of the winner and loser, respectively. In Appendix B, a service definition of this game is described in LOTOS.

2.2 Protocol Specification

On the level of the protocol specification, the communication medium is considered explicitly. If some nodes must cooperate to provide a service, they must synchronize each other in order to keep the temporal ordering of the execution of the events.

LOTOS has rendezvous communication mechanisms among more than two nodes. However, we treat only asynchronous communication between two nodes. For this restricted class, some techniques synthesizing protocol specifications have been proposed [6],[7],[8],[9]. Also many practical protocols can be described in this class [8]. Therefore, we think the class is still useful. We have easily implemented the communication mechanisms of this class using inter-process communication(IPC) of UNIX system. Hereafter, we assume asynchronous communication.

For example, in order to execute an event “a” at a node “i” and then to execute an event “b” at another node “j”, the node “i” must send a synchronization message to the node “j” after “a” is executed. The node “j” must execute “b” after the node “j” receives the synchronization message. If a node uses the data values which it does not know, the node must receive the values from the nodes which know them. For example, if a value of a variable “x” is inputted at a node “i” and another node “j” needs the value, then the value must be transmitted from the node “i” to the node “j”. So, we must determine what kinds of the synchronization messages and data values should be sent and/or received in each node. We must also determine the timing of the sending/receiving actions and the destinations of the sending/receiving messages and data.

We assume that there is a reliable asynchronous communication channel from each node “i” to any other node “j”. That is, we assume that each message sent from the node “i” is eventually received by the node “j”. Each channel does not lose, duplicate, nor insert messages. At each node “i”, the sending action of a synchronization message “m” to a node “j” is described as the special event “sij(m)”. And the receiving action of a message “m” from a node “j” is described as “rj(m)”. There are two types of messages : synchronization messages and data values. We use the integers as the varieties of the synchronization messages. Let “k” be an integer and let “x” be an variable. The events sij(k) and sij(x) represent the transmission of the synchro-
In Appendix C, we will give a protocol specification which provides the service described in Appendix B. An execution process of the protocol specification in Appendix C is illustrated in Fig.2.

![Fig.2 An Execution Process of Protocol Specification](image)

First, the node 1 executes the event j"start" and sends the synchronization messages “1” to the nodes 2 and 3. If the nodes 2 and 3 receive the messages, then they read the choices of the players A and B and inform them to the node 1. Then, the node 1 decides the winner and shows it to the Board. If the winner is A (or B), then it sends the synchronization messages “2” (or “3”) to the nodes 2 and 3. The nodes 2 and 3 know the winner by receiving these messages. If they receive the synchronization message “4”, then the processes Game_2 and Game_3 are invoked again.

2.3 Correctness of Protocol Specifications

In this section, we will define the correctness of protocol specifications formally. Let P and Q be processes written in LOTOS. If the processes P and Q are observational equivalent [1], then we describe “P=Q”. Let P_k be a specification of the node “k”. For a service definition P_S and a protocol specification <P_1,...,P_N> with N nodes, we say that the protocol specification is correct with respect to the service definition P_S if the following relation holds.

P_S \approx \text{hide } G \text{ in } (P_1 || P_2 || \cdots || P_N)[[ G ]] \text{ Comm}

Here,

G = \{ s_{ij}, r_{ij} \mid 1 \leq i, j \leq N, i \neq j \}

Comm = Comm_{12} || \cdots || Comm_{ij} || \cdots || Comm_{N-1 N}

Comm_{ij} = s_{ij} ; r_{ij} ; [Comm_{ij}]

“Hide H in Q” represents that the events of the process Q belonging to H are treated as internal events. Internal events are not observed from the external environments. And “P || [F] Q” denotes that two processes P and Q are executable in parallel, and that the events of P and Q belonging to F must be executed simultaneously. The expression “(P_1 || \cdots || P_N)[[ G ]] Comm” requires that a receiving action should not occur prior to a sending action in each communication channel, and that if a sending action is executed, then the corresponding receiving action must be executed eventually. And “\text{hide } G” represents that the sending/receiving actions are treated as internal events.

For example, since the service definition Janken in Appendix B and the protocol specification <Janken_1, Janken_2, Janken_3> in Appendix C satisfy the following relation, the protocol specification is correct.

Janken[j,a,b] \approx \text{hide } G \text{ in } (Janken_1[j,s12,s13,r21,r31] || Janken_2[a,r12,s21] || Janken_3[b,r13,s31] ) [[G]]

Comm

Here, G = \{ s12,s13,r21,r31,s21,r12,s31,r13 \}

Comm = Comm_{12} || Comm_{13} || Comm_{21}

Comm_{ij} = s_{ij} ; r_{ij} ; [Comm_{ij}]

We can show that the above protocol specification is correct by using the technique in Ref. [5].

3. Facilities of PROSPEX

As we mentioned in Section 1, to use support tools for the simulation and analysis of protocol specifications is useful to develop correct protocol specifications. In this section, we explain the main facilities of our PROSPEX.

3.1 Simulation of Protocol Specifications

In PROSPEX, if a service definition P_S and a protocol specification <P_1,P_2,...,P_N> with N nodes are given, then PROSPEX generates N simulators S_1, ..., S_{N-1} and S_N and a monitor S_0. We use each simulator interactively. Each simulator S_k gets the specification P_k of the k-th node as the current behavior expression B_k when it starts the simulation. Here, the current behavior expression represents the event sequences which the node can execute in the future. The simulator S_k shows which events are executable for the current behavior expression B_k. The user chooses one executable event e from the candidates which the simulator shows. Then, the simulator executes the event e and then computes a new behaviour expression B_{k+1} after e is executed. After that, it shows which events are executable for B_{k+1}. The simulation is carried out by repeating these steps.

For example, suppose that the service definition in Appendix B and the protocol specification in Appendix C are given. PROSPEX generates three simulators S_1, S_2 and S_3 and one monitor S_0. The monitor draws the syntax tree of the behavior expression of the service definition “Janken”. Each simulator S_i draws the syntax tree of the behavior expression of the specification “Janken_k” on a display (see Fig.3(a)).
The event "start" is executable.

The events a?x:finger and b?y:finger are executable.

The player A inputs his choice.

The Board displays the winner.

Each leaf corresponds to either an event, a sending/receiving action or a process name. In Fig. 3, the sending/receiving actions "s_{ij}(m)" and "r_{ij}(m)" at each node "i" are abbreviated as "s_{ij}(m)" and "r_{ij}(m)", respectively. All executable events are shown by the dotted rectangles. In Fig. 3(a), only the event "start" at the node 1 is executable. If the user clicks "start" at the node 1, then the simulator S_1 executes the event. After "start" is executed, a new behavior expression, say B', at the node 1 is obtained. For the new expression B', the sending actions "s_{11}(x)" and "s_{12}(x)" are executable. The simulator S_1 executes these sending actions automatically without interactions from the user. Then, the receiving actions "r_{12}(1)" at the node 2 and "r_{13}(1)" at the node 3 become executable. The simulators S_2 and S_3 execute these receiving actions automatically (see Fig. 3(b)). In Fig. 3(b), the events "a?x:finger" at the node 2 and "b?y:finger" at the node 3 are executable. From Fig. 3(b), we can know that both "a?x:finger" and "b?y:finger" are also executable in the service definition. Fig. 3(c) denotes that the player A chooses paper and inputs it (the input data is assigned on a small window). By executing sending/receiving actions s_{21}(x) and r_{21}(x), the node 1 knows that the player A chooses paper and waits for the data from the node 3 (the player B). If the node 3 sends the choice of player B to the node 1, then the node 1 calculates the winner and shows it on a small window (see Fig. 3(d)). The simulation is continued by repeating the similar interactions.

Here, our PROSPEX doesn't draw the labeled transition system (LTS) [1] but the syntax tree of a given behavior expression. By drawing the syntax tree, it is easy to understand the structure of the behavior expression such as the nesting information of parallel and choice operators. We can easily know what kinds of synchronization messages must be received to execute an event. But, if we want to prove the equivalence of two specifications, to draw their LTSs is more suitable. We have developed a test system for LOTOS expressions [23]. In the test system, we have implemented the facility to draw the LTS of a given behavior expression. Now, we are planning to add the facility to PROSPEX.

3.2 Monitoring Service Definition

PROSPEX has the facility for monitoring a service definition. The monitor S_0 checks a status of the simulation of the protocol specifications against the service definition. If an event on a node is executed and it is not executable for the service definition, PROSPEX indicates to the users that the event cannot be executed. In this case, the protocol specification is not observational equivalent to the service definition. If the events on the service definition are executable, then they are executed automatically according to the information from the corresponding simulator. If the service definition contains nondeterminism, then there may be some problems when an event on a protocol specification is executed. For instance, suppose that the current behavior ex-
pression on a service definition is “a:b [] a;c”. When the event “a” is executed on a node, the monitor cannot determine to execute which side of event “a” is executed. In such a case, PROSPEX indicates the candidates of the corresponding events, and make the users choose one of them. For each execution step, the users can check whether all executable events in the protocol specification are also executable in the service definition. This facility helps the users to develop correct protocol specifications.

4. Design and Implementation of PROSPEX

In this section, we pick up some problems which occurs when we design and implement the support tools such as PROSPEX, and describe how we have solved such problems when we have developed PROSPEX.

4.1 Convenient Notation for Describing LOTOS Specifications

In a LOTOS specification, the temporal ordering of the execution of the events is described as a behaviour expression, and the abstract data types used in the specification are described in an algebraic specification language ACT ONE [16]. Hereafter, we call the above two parts “the behaviour expression part” and “the data type part”, respectively. Our PROSPEX supports Full LOTOS functionality although some special operators in Full LOTOS are not supported for simplicity [14].

In the behaviour expression part, we can use the following operators: “;”, “[]”, “||”,” ”,” “>”,” ”,” ”,” “,” ”,” and “hide”. But, we do not support the “let”, “choice”, “par”, “accept” and “any” operators. In the data type part, we can use the “type”, “library”, “sorts”, “opns” and “eqns” operators. We do not support the “formalsorts”, “formalopns”, “formaleqns”, “sortnames” and “opnnames” operators. The axioms in the data type part are treated as a term rewriting system. Therefore, the variables in the right side of each axiom must be appeared in the left side of the axiom. We do not support conditional axioms. In LOTOS, all the abstract data types used in a specification must be described by the users as an algebraic specification even if the abstract data types are primitive. Although some library texts are prepared, some LOTOS simulators cannot use the normal mathematical notations such as decimal arithmetic. This makes LOTOS specifications unreadable. For solving this problem, PROSPEX supports the normal mathematical notations.

If the text “primitives” is described in the library statement, then primitive data types such as integer, character, string, list, array and tuple can be treated as the pre-defined data types. The primitive functions such as if-functions, “+”, “-”, “*”, “/”, “-”, “>”, “and”, “or”, “not”, “car” and “cdr” are also treated as the pre-defined functions. The users do not have to describe the axioms for these primitive functions when they use PROSPEX.

4.2 Parsing of LOTOS Specifications

Since, in LOTOS, the function names used in the data type part and their syntax are defined by the users, parsing of LOTOS specifications is not simple. Therefore, some LOTOS simulators do not check the syntax strictly. Here, we give a method to parse LOTOS specifications strictly.

We have defined an algebraic specification language ASL, and developed a support system, ASL system, to design and develop programs in ASL [17],[18]. A specification in ASL can be denoted by a pair $t = (G, AX)$ where $G$ is a context free grammar and $AX$ is a set of axioms. A set of terminal symbols $Term(t)$ generated by $G$ is treated as the terms used in this text. The congruence relation on $Term(t)$ is defined by the axioms $AX$. ASL system reads a text $t = (G, AX)$ and generates a parser $P(G)$ for $G$. The parser $P(G)$ examines whether a given term can be generated by $G$. By using this facility, ASL system examines whether each axiom in $AX$ can be generated by $G$. If all axioms in $AX$ are generated by $G$, ASL system generates an interpreter $R(AX)$ which regards each axiom in $AX$ as a rewrite rule, and calculates a normal form of a given term. The normal form corresponds to the value of the given term. Our interpreter $R(AX)$ treats the axioms $AX$ as a term rewriting system. In order to parse a LOTOS specification $t_L$, PROSPEX divides the LOTOS specification $t_L$ into two parts, the data type part $t_D$ and the behavior expression part $t_B$. Then, PROSPEX parses each of them.

(1) Parsing Data Type Part

The data type part $t_D$ consists of two sub-parts, the function definition part $t_{DF}$ and axiom part $t_{DA}$. The function definition part $t_{DF}$ declares the function names used in the axioms and defines their syntax. The axiom part $t_{DA}$ describes some axioms whose functions are all defined in the function definition part. The syntax of the function definition part $t_{DF}$ is defined based on the grammar $G_{ACT}$ of ACT ONE [1],[16]. Therefore, PROSPEX checks the syntax of the function definition part $t_{DF}$ by using the parser $P(G_{ACT})$. If the syntax of $t_{DF}$ is correct, then PROSPEX constructs a grammar $G_{t_{DF}}$ for generating the terms consisting of the functions which are defined in $t_{DF}$. By using $P(G_{t_{DF}})$, the syntax of the axiom part $t_{DA}$ is checked. If the syntax of both the function definition part $t_{DF}$ and axiom part $t_{DA}$ is correct, then an interpreter $R(t_{DA})$ is derived.

(2) Parsing Behaviour Expression Part

Let $G_B$ be a grammar generating behaviour expressions in LOTOS [1]. PROSPEX examines whether the syntax of a given behaviour expression is correct by using $P(G_B)$. The syntax of each guard in the behaviour expressions is parsed by $P(G_{t_{DF}})$.

The parser is developed based on Earley method, it takes $O(n^3)$ times to parse a LOTOS specification whose length is $n$. 
4.3 Execution of LOTOS Specifications

In this section, we describe the facilities for the execution of LOTOS specifications and their implementation.

(1) Finding Executable Events from Current Behaviour Expression

In order to find executable events, PROSPEX searches the syntax tree of the current behaviour expression from the root node to the leaf nodes. The way to search a given tree depends upon the operators on the internal nodes. For instance, if the operator "\(>>\)" is found while searching a tree, then the next search is continued to its left descendant node. On the other hand, if the operator "\([\)" is found, the search is continued to the both descendant nodes. Some guard expressions may be described in a behaviour expression. While searching tree, if a guard expression is found, PROSPEX calculates the value of the guard by using the interpreter R\((t_{DA})\) which was explained in Section 4.2. If the value is true, the search is continued. Otherwise, that is, if the value is false, then we do not search the guard expression.

Since the value of each guard is stored when the value is calculated, it is not calculated twice.

(2) Execution of Executable Event

The three types of events are supported in PROSPEX: (i) Input events and output events, (ii) Internal events, and (iii) Sending and receiving actions.

The input events and output events are executed by clicking the executable events on the display. When an input event such as "\(a?x:finger\)" is clicked, PROSPEX opens a small window. The user can give an input value on the small window (see Fig.3 (c)). When an output event such as "\(f!x:stop\)" is clicked, PROSPEX opens a small window and display it on the small window (see Fig.3 (d)). If a guarded event such as "\(f?y:int[x>2]\)" is executed, PROSPEX examines whether the input value satisfies the guard. If the input value does not satisfy the guard, then the execution of the event is canceled. The values of the outputs and guards are calculated by using the interpreter R\((t_{DA})\).

Internal events are the events which are not seen from the external environments. Usually each internal event is described as "\(i\)". The "\(exit\)" event and the events hidden by the "\(hide\)" operator are also treated as the internal events. PROSPEX executes these internal events as soon as they become executable.

When a sending (receiving) action is executable, PROSPEX sends (receives) a synchronization message or a data value to (from) the designated node. In order to help the designer to find some redundant synchronization messages, the sending/receiving actions can be also executed in manual. If manual mode is selected, then the sending/receiving actions are executed according to the guide from the user. By this facility, the designer can easily recognize how the nodes synchronize each other.

(3) Transformation of Current Behaviour Expression

In LOTOS, when an event \(e\) is executed for the current behaviour expression \(B\), it must be transformed into a new behaviour expression \(B'\) after \(e\) is executed. PROSPEX has the facility for the transformation which was explained in Section 3.

(4) Invocation of A Process

In the syntax tree of a behaviour expression which PROSPEX draws, each process is drawn as a node. When some events in the process become executable, the process is invoked. When a process is invoked, it is allowed to replace the gate names and parameter names. Therefore, PROSPEX can replace them. For instance, suppose that \("P[a,b](x):= a?y:int;b!x;stop\)". If \(P[f,g](2)\) is invoked, PROSPEX replaces the process \(P[f,g](2)\) by the behaviour expression "\(f?y:int[g];stop\)".

4.4 Displaying Syntax Tree of Current Behaviour Expression

Since nondeterminism and parallelism may be described in LOTOS specifications, it is difficult to understand the structure of the current behavior expression if it is displayed as a character string. Therefore, we show the syntax tree of the current behavior expression graphically on a display so that the users can understand its structure at a glance. Since the size of each LOTOS specification may become large, the simulator must be able to handle large trees. Then, we have developed a graph editor VTM which can handle large trees with thousands of nodes. VTM makes users easy to observe a whole tree and some specified parts of the tree simultaneously. VTM is a library program which can be used easily from any application program without the knowledge about X Window System. In order to monitor both the outline of a whole tree and the details of the specified part of the tree simultaneously on a display, VTM prepares two windows, GlobalView and Canvas (Fig.4).

On GlobalView a whole tree is illustrated where each node is represented by a dot without the label and a rectangle called AreaMark are displayed. On Canvas the area spec-
ified by AreaMark are zoomed up. Each node on Canvas is displayed by a rectangle in which its label is written. In order to observe several parts simultaneously, it is possible to open more than one canvases. Users can move and resize AreaMark arbitrarily by pushing the center and left mouse buttons, respectively, and dragging it to the appropriate directions. It takes less than 0.2 seconds for VTM to calculate the layout and display the whole tree which has 1000 nodes (on a DECstation 3100 with 12MB memory).

4.5 Performance of PROSPEX

In this section, we explain the capacity and the speed of PROSPEX. Generally speaking, the size of the specification PROSPEX can deal with depends on the memory size of the machine where it works. On a DECstation 3100 with 12MB memory, PROSPEX can draw a syntax tree which has thousands of nodes. The number of LOTOS specifications which can be simulated simultaneously depends on the number of file descriptors in UNIX system. About 30 LOTOS specifications can be simulated simultaneously on the DECstation 3100.

We have measured the running time of PROSPEX. In order to execute an input/output event, PROSPEX must (1) check the syntax of the input/output data, (2) calculate a new behaviour expression after the event is executed, (3) find executable events for the new behaviour expression and (4) draw the syntax tree of the new behaviour expression, and so on. For a behaviour expression whose syntax tree has about 200 nodes, it takes about 0.5 - 2 seconds to execute the above four tasks. If we must calculate the values of the guard expressions, it takes much time. For Janken Game in Section 2, it takes about 6 - 10 seconds for each simulator to execute the LOTOS specification, communicate each other and finish one match.

For the current version of our PROSPEX, there are some limitations. For example, since LOTOSPHERE [12] supports conditional axioms, the inference using “narrowing” can be handled. But, our PROSPEX does not support it. Also, our PROSPEX assumes asynchronous communication and cannot handle rendezvous communication among more than two nodes.

5. Conclusion

In this paper, we describe the facilities and implementation of our LOTOS tool PROSPEX which has been developed to observe the execution processes of a protocol specification and to check whether the protocol specification executes the event sequences which satisfy the required service. Our PROSPEX supports to prove the correctness of the protocol specifications. But, in general, it becomes difficult to prove the correctness if the size of the specification becomes large.

We have been developing a tool to synthesize a correct protocol specification from a service definition based on the techniques in Ref.[9],[10]. By combining PROSPEX and the synthesizer, it becomes a more useful system to develop correct protocol specifications. To prove that the produced protocol specification is correct for a given service definition, we can use the well-known decision procedure in Ref.[4],[5]. Now we have been developing the decision procedure based on the technique in Ref.[5]. One of the future work is to give a practical example.

References


Rule of Janken Game

Janken Game is a finger-flashing game of paper-scissors-stone. Paper wins against stone. Stone wins against scissors. Scissors wins against paper. Each player chooses one of them. For instance, if the players A and B choose stone and scissors, respectively, then the player A wins the game. If they choose the same one, then they try the game again.

Service definition of Janken Game

specification Janken[j,a,b]:exit
behavior Game[j,a,b]
where
process Game[j,a,b] :=
  j!"start" ; ( a?x:finger ; exit || b?y:finger ; exit )
>> ( ( [result(x,y)="A"] -> j!result(x,y) ;
    (a!"win" ; exit || b!"loss" ; exit ) )
  [] ( [result(x,y)="B"] -> j!result(x,y) ;
    (a!"loss" ; exit || b!"win" ; exit ) )
  [] ( [result(x,y)="tie"] -> j!result(x,y) ; Game[j,a,b] )
) endproc
endspec

Here, "result(x,y)" is a function which calculates the winner of this game. It returns the string "A" (or "B") if "x" wins (loses) against "y". Otherwise, it returns the string "tie".

Protocol specification of Janken Game

(1) Node 1

specification Janken_1[j,s12,s13,r21,r31]:exit
behavior Game_1[j,s12,s13,r21,r31]
where
process Game_1[j,s12,s13,r21,r31] :=
  j!"start" ; ( s12(1) ; exit || s13(1) ; exit )
>> ( ( [result(x,y)="A"] -> j!result(x,y) ;
    (s12(2);exit || s13(2);exit ) )
  [] ( [result(x,y)="B"] -> j!result(x,y) ;
    (s12(3);exit || s13(3);exit ) )
  [] ( [result(x,y)="tie"] -> j!result(x,y) ;
    (s12(4);exit || s13(4);exit)
  >> Game_1[j,s12,s13,r21,r31] )
) endproc
endspec

(2) Node 2

specification Janken_2[a,r12,s21]:exit
behavior Game_2[a,r12,s21]
where

process Game_2[a,r12,s21]:=
  r12(1) ; a?x:finger ; s21(x) ;
  ( ( r12(2) ; a!"win" ; exit )
  [] ( r12(3) ; a!"loss"; exit )
  [] ( r12(4) ; Game_2[a,r12,s21] )
)
endproc
endspec

(3) Node 3

specification Janken_3[b,r13,s31]:exit
... (the description of the data types used in this program)
behavior Game_3[b,r13,s31]
where

process Game_3[b,r13,s31]:=
  r13(1) ; a?y:finger ; s31(y) ;
  ( ( r13(2) ; b!"loss"; exit )
  [] ( r13(3) ; b!"win" ; exit )
  [] ( r13(4) ; Game_3[b,r13,s31] )
)
endproc
endspec