Synthesis of Protocol Entities' Specifications from Service Specifications in a Petri Net Model with Registers

Hirozumi Yamaguchi Kozo Okano Teruo Higashino Kenichi Taniguchi

Department of Information and Computer Sciences, Osaka University
Machikaneyama 1-3, Toyonaka, Osaka 560, JAPAN
e-mail : {h-yamagu, okano, higashino, taniguchi}@ics.es.osaka-u.ac.jp

Abstract

In general, the services of a distributed system are provided by some cooperative protocol entities. The protocol entities must exchange some data values and synchronization messages in order to ensure the temporal ordering of the events which are described in a service specification of the distributed system. It is desirable that a correct protocol entity specification for each node can be derived automatically from a given service specification. In this paper, we propose an algorithm which synthesizes a correct protocol entity specification automatically from a service specification in a Petri Net model with Registers called PNR model. In our model, parallel events and selective operations can be described naturally. The control flow of a service specification must be described as a free-choice net in order to simplify the derivation algorithm, however, many practical systems can be described in this class. In our approach, since each protocol entity specification is also described in our PNR model, we can easily understand what events can be executed in parallel at each protocol entity.

1 Introduction

At a high level of abstraction, a distributed system can be treated as a service provider that offers some specified services to some service users. A description of the temporal ordering of the service primitives occurred at the SAP's (Service Access Points) can be treated as a service specification for the system[2]. At this level, the distributed system can be seen as a black box. At a lower level of abstraction, the services are provided by some cooperative protocol entities (or nodes) that exchange some messages each other in order to synchronize and/or exchange data values. At this level, the behaviors of each protocol entity are described as a protocol entity specification, and the set of all protocol entities' specifications is called a protocol specification[2].

In order to get a correct protocol specification satisfying a given service specification, it is desirable that the designer only describes a service specification and that a correct protocol specification can be derived automatically from the service specification. Some synthetic approaches have been proposed such as LOTOS based approaches[3], FSM/EFSM based approaches [4][6] and so on (for survey, see [2]). In LOTOS based approaches, parallel events can be described in a service specification, however, efficient distributed control methods have not been proposed for the case that the state variables are allocated to some protocol entities separately. On the other hand, in FSM/EFSM based approaches, parallel events cannot be treated.

In our previous paper [6], we have proposed a derivation technique for an EFSM model where some registers (state variables) can be allocated to some protocol entities. And a distributed control method for changing the registers' values in a given distributed environment efficiently is proposed. In general, for specifying distributed systems naturally, it is desirable that both parallel events and state variables can be treated in a service specification. However, for such a class, useful algorithms for deriving protocol specifications have not been proposed yet.

In this paper, we define a Petri Net model with Registers (PNR model). In our PNR model, the control flow is described as a Petri net and the model may have a finite number of registers. The registers represent resources in the distributed system. At each transition, one I/O event is executed and the next registers' values are calculated from the current registers' values and input data. For each transition, we can give a guard to control the firability of the transition. Using the guards, selective operations can be described. For this PNR model, we propose a technique for deriving a correct protocol specification from a given service specification and an allocation of I/O gates and registers to protocol entities. The service specifications and derived protocol entities' specifications are described

in our PNR model. For simplifying the derivation algorithm, we restrict the class of service specifications where underlying nets must be treated as free-choice nets[1] (FC nets). Although we restrict the class, parallel events and non-deterministic selections can be described naturally in this class, and the class has considerable power for specifying practical distributed systems. In the proposed method, each protocol entity specification is obtained from a given service specification by replacing each transition with a sub-Petri net. The sub-Petri net cooperates with other protocol entities' sub-Petri nets for simulating the transition.

The paper is structured as follows. In Section 2, we give a definition of our PNR model. In Section 3, an example of a service specification is explained. In Section 4, a derivation problem treated in this paper is formally defined. The derivation algorithm is described in Section 5.

2 Basic Definitions

2.1 Petri Nets

Definition1(Petri Net) A Petri net[1] is denoted by a 4-tuple $PN = (P, T, F, M_0)$ (or simply (N, M_0) where N denotes (P, T, F)). P is a finite set of places and T is a finite set of transitions satisfying both $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. $F \subseteq P \times T \cup T \times P$ is a set of arcs and $M_0 : P \rightarrow \{0, 1, 2, \ldots\}$ is the initial marking. We say that a place p is marked with k tokens if nonnegative integer k is assigned as a marking for p.

For $u, v \in P \cup T$, we denote the preset $\{v | (v, u) \in F\}$ of u by $\bullet u$ and the postset $\{v | (u, v) \in F\}$ of u by $u\bullet$. Similarly we denote the union of all presets of u in a set S by $\bullet S$ and the union of all postsets of u in S by $S\bullet$. A transition t is said to be enabled if each input place p of t is marked with at least one token.

Definition2(Free-Choice Net) A free-choice net[1] (FC net) is a Petri net such that for all $p \in P$, $|p\bullet| \leq 1$ or $\bullet\{p\bullet\} = \{p\}$.

Definition 2 indicates that if transitions share an input place p in a FC net, then each of the output transitions of p has exactly one input place p. In other words, if a place p has a selective structure, p can determine independently which output transition of p can fire (see an example in Fig. 1). In the sequel, a FC net is a Petri net such that a place p with a selective structure can select its selection itself.

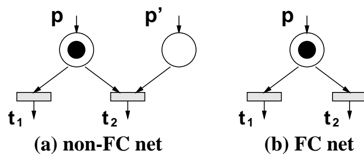

Figure 1: Selective Structures.

2.2 Petri Net with Registers

In this paper, we introduce a Petri Net model with Registers (PNR model).

Definition3(Petri Net with Registers) A Petri Net with Registers (PNR) is denoted by a pair $PNR = (PN, \Sigma)$, where Σ is defined as a 7-tuple $\Sigma = (G_s, \mathcal{A}, G, R, C, \delta, Init)$. PN is a Petri net which does not contain isolate transitions nor places. G_s is a finite set of gate symbols. $\mathcal{A}$ is a finite set of events whose gates are the elements in G_s. G, R, C are finite sets of guards, registers, register definition statements, respectively. $\delta : T \rightarrow G \times \mathcal{A} \times C$ is a function representing the contents of transitions and $Init$ is a function specifying the initial values of registers. Here the net PN is called the underlying net of the PNR.

A PNR may have some registers R_1,...,R_n. Each R_i is called a register variable. Each transition in a PNR has a label of 3-tuple [a guard, an event, a register definition statement]. An event in the set $\mathcal{A}$ must have one of the the following three forms : $a?x$, $a!E(\ldots)$ and i. The $a?x$ denotes an input event and the variable x represents an input value from the gate a (if more than one input values are given, it is denoted like $a?x_1, x_2, x_3, \ldots$). The $a!E(\ldots)$ denotes an output event and the value of the expression E is emitted from the gate a. E is an expression which may contain register variables. The event i is an internal event which does not execute any input/output. A guard in the set G is a predicate which may contain the register variables and/or input variables. A register definition statement in the set C has the form $R_{h_1} \leftarrow f_1(\ldots)$, ..., $R_{h_i} \leftarrow f_i(\ldots)$, where each f_j $(1 \leq j \leq i)$ is a function which may contain the register variables and/or input variables.

A transition t is enabled in a $PNR = (PN, \Sigma)$ iff t is enabled in the PN and the value of the guard of t is true. If an enabled transition t fires, the event of t is executed, and then the values of registers are changed concurrently based on its register definition statement (by preserving the current values of registers before the calculation, this concurrent calculation can be done even on the single CPU system). For an enabled transition which executes an input event, we assume that it cannot fire until input data are given.

If (1) the event of a transition t is the internal event i, (2) the guard of t is "true" and (3) the register definition statement of t is empty, then we call the transition t an ε-transition.

3 An Example of Service Specification

The service specifications must be described in the PNR model that satisfies some restrictions. The restrictions are described in Section 4.

Fig. 2 shows an example of a service specification SS in the PNR model. The specification describes a simple distributed database system with a back up mechanism. There are four registers R_1, R_2, R_3 and R_4.

The system works as follows. First, the transition t_1 fires and an input data x is given from the gate a. The data x is stored in the register R_4. After that, either transition t_2 or t_3 fires. The values of their guards decide which transition can fire. The guards of t_2 and t_3 are Man(R_4) and Woman(R_4), respectively, and they calculate an attribute from the value of the register R_4. If the value of the attribute is "male", then the value of the guard Man(R_4) becomes true, and the transition t_2 fires. Then, the input data (the value of the register R_4) is appended to the register R_1 (database for men). If the value of the attribute is "female", then the value of the guard Woman(R_4) becomes true, and the transition t_3 fires. Then, the input data is appended to the register R_2 (database for women). On the other hand, the transition t_4 can fire in parallel with the transition t_2 or t_3. And the value of the register R_4 (input data) is appended to the register R_3 (database for back up). After those, the transition t_5 fires and the size (volume) of the back up database (register R_3) is output from the gate c. At this moment, the marking is the same as the initial marking.

In Fig. 2, if the guards are "true", or if the register definition statements are empty, then they are omitted. Here, we do not describe the details of the functions used in Fig. 2 such as **Man**, **Woman**, **append** and

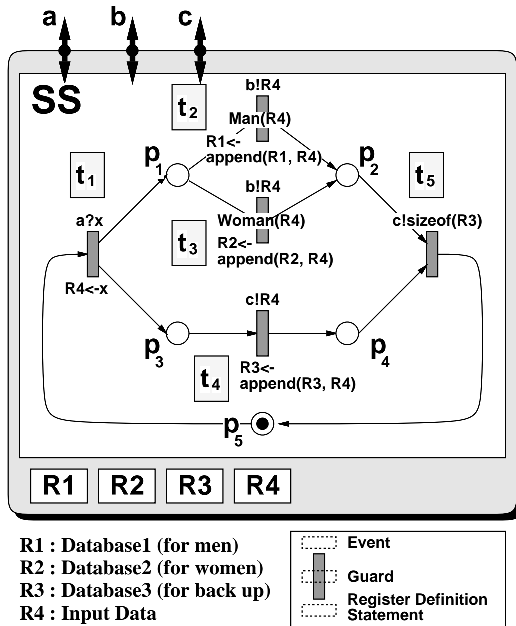

Figure 2: An Example of a Service Specification SS.

sizeof. We can define the details freely and they do not influence on the derivation algorithm.

4 Derivation Problem and an Example of Protocol Specification

In this section, we will give a formal definition of the problem to implement a given service specification SS in a distributed system with p protocol entities. In this paper, we assume that each protocol entity specification is also described as the PNR model.

Let PE_k denote a protocol entity specification for the protocol entity k, and let $\langle PE_1, \ldots, PE_p \rangle$ (or, simply $PE^{\langle 1,p \rangle}$) denote a protocol specification with p protocol entities. Here, we assume that each communication channel from protocol entity i to protocol entity j is modeled as a FIFO queue (queue$_{ij}$) whose capacity is infinite. We call both sides of the channel the gate g_{ij}. If the protocol entity i executes an output event "$g_{ij}!d$", then the data d is enqueued to the queue$_{ij}$. If the protocol entity j executes an input event "$g_{ij}?x$" and the first element of the queue$_{ij}$ is d, then the data d is dequeued from the queue$_{ij}$ and the value of d is assigned to the input variable x. If there are no elements in the queue$_{ij}$, then we assume that protocol entity j cannot execute the input event $g_{ij}?x$.

4.1 Allocation of Gates and Registers

We assume that each gate must belong to one of p protocol entities, and that each register must be allocated to more than one protocol entities. This means a distributed allocation of resources. Let Θ denote such a resource allocation. Fig. 3 denotes an example of such a resource allocation Θ.

Since we assume that each gate must belong to one of p protocol entities, the protocol entity which executes the event of each transition t in SS can be determined uniquely. We call such a protocol entity a responsible protocol entity of the transition t, and denote it by RPE(t). Also, the responsible protocol entities of all transitions in $t\bullet\bullet$ are called next responsible protocol entities, and we denote a set of such protocol entities by RPE($t \bullet \bullet$).

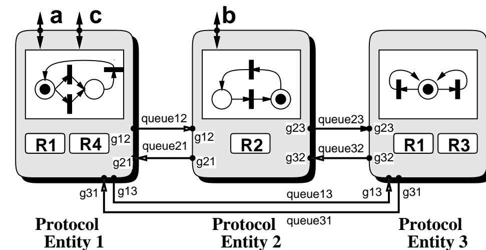

Figure 3: An Allocation of Registers and Gates.

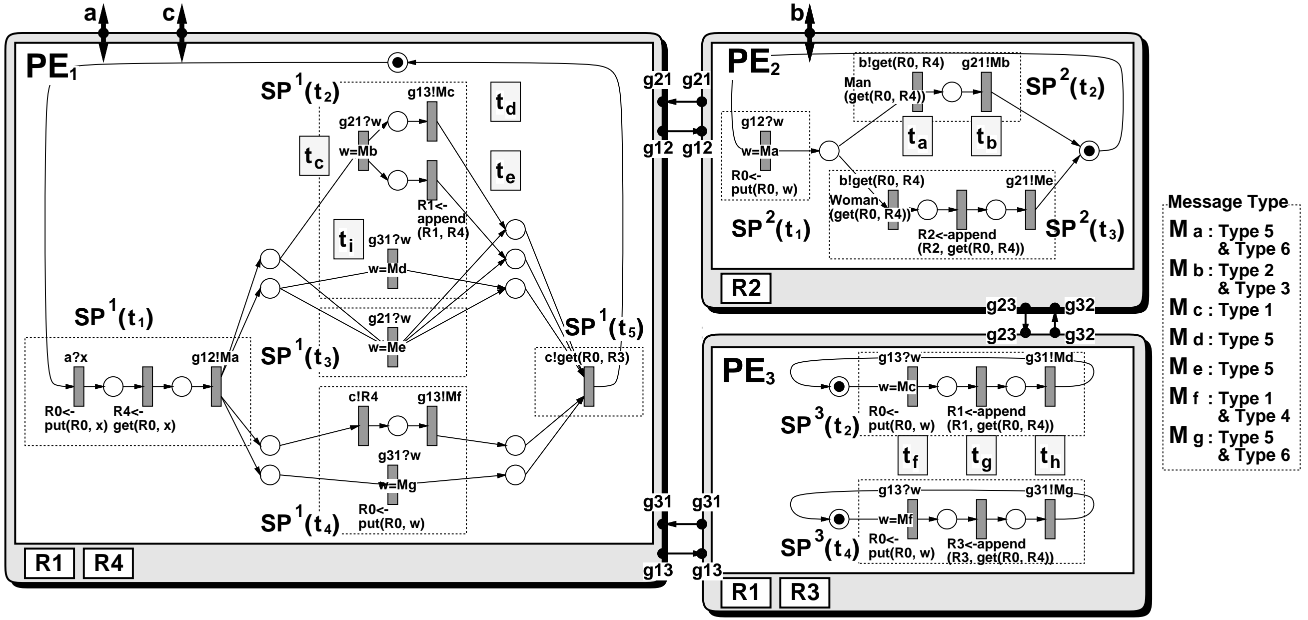

Figure 4: PE1, PE2 and PE3.

4.2 Derivation Problem

For a protocol specification $PE^{\langle 1,p\rangle}$, we define its initial state as follows. If (1) each $PE_k (1 \leq k \leq p)$ is at its initial marking, (2) its registers are set to the initial values and (3) all FIFO queues (communication channels) are empty, then such a state is called the initial state of the $PE^{\langle 1,p\rangle}$.

We define the equivalence between a service specification SS and a protocol specification $PE^{\langle 1,p\rangle}$ as follows.

Definition4(Equivalence) Suppose that all I/O events $g_{ij}?x$ and $g_{ij}!E(\ldots)$ for the communication channels and internal events i are treated as unobservable and the other I/O events are treated as observable. If a service specification SS and a protocol specification $PE^{\langle 1,p\rangle}$ are observational congruent [5], then we say that SS and $PE^{\langle 1,p\rangle}$ are equivalent. Such $PE^{\langle 1,p\rangle}$ is called a correct protocol specification for the service specification SS.

If SS and $PE^{\langle 1,p\rangle}$ are observational congruent, then all observational event sequences in SS ($PE^{\langle 1,p\rangle}$) must be also executable in $PE^{\langle 1,p\rangle}$ (SS). Also at the state after executing each observational event sequence, executable observational events for SS and $PE^{\langle 1,p\rangle}$ must be the same.

Here, we define the conflict of registers in SS as follows.

Definition5(Conflict of Registers) If there are two transitions t_i and t_j ($t_i \neq t_j$) which can fire simultaneously and if the transition t_i (or t_j) has a register definition statement that changes the value of a register R and the another transition t_j (or t_i) has a register definition statement or an output event that uses the register R, then we say that there is a conflict of registers for the pair of the transitions t_i and t_j.

From Definition 5, the problem to find possibilities of conflicts of registers for a given service specification can be reduced to the concurrent firability problem of its underlying net and it is known that the concurrent firability problem is decidable[1].

Definition6(Derivation Problem) For a given SS, a set of p protocol entities and a resource allocation Θ, we will consider the problem to derive a correct protocol specification $PE^{\langle 1,p\rangle}$ in this paper.

Here, for simplicity of discussion, we will give the following five restrictions for SS and Θ.

1. SS must be modeled as a $PNR = (PN, \Sigma)$ and the PN must be a live and safe FC net (if the PN is not connected, then each connected net must be a live and safe FC net).
2. The responsible protocol entities for all transitions sharing one input place must be the same, i.e., for any $t_1, t_2 \in T$, $RPE(t_1) = RPE(t_2)$ if $t_1, t_2 \in p\bullet$.
3. There are no conflicts of registers in SS.
4. There are no internal events i in SS.
5. If a transition t in SS can fire at the initial state, then the registers used in its guard and/or output event must be allocated to the protocol entity $RPE(t)$.

By the restrictions 1 and 2, if selective transitions are described in a service specification, such a selection can be done by the responsible protocol entity for those transitions. Also, the restriction 1 has good effects for

the reformation of protocol entities' specifications as described later. A service specification satisfying the restriction 3 is called a conflict-free service specification. In our model, more than one registers' values may be changed in each transition and such a change is executed by some cooperative protocol entities. If a conflict of registers occurs, during simulation of one transition, another transition must be blocked to avoid inconsistency of the registers' values among the protocol entities. For this purpose, many mutual exclusion algorithms are useful where distributed mutual exclusion controls for accessing critical sections are given. We can apply one of them to such transitions to control the conflicts of registers. For the details, see [8]. The restriction 4 is used for simplifying the proof of equivalence. This restriction is not essential. By the restriction 5, the responsible protocol entity of the transitions which can fire at the initial state can decide which transition can fire by itself and start the simulation without asking registers' values to other protocol entities. This restriction is not also essential. By inserting a dummy transition with an empty register definition statement and a "true" guard before those transitions and treating it as the initial transition, this restriction can be removed.

4.3 An Example of Protocol Specification

Fig. 4 shows $PE^{\langle 1,3 \rangle}$ derived from the service specification SS in Fig. 2 and the resource allocation Θ in Fig. 3.

Here, we assume that each protocol entity has an additional local working register R_0, which has a value table. The values of registers and input data sent through the communication channels can be kept in the table at each protocol entity. We use the following functions as primitive functions. The function $put(R_0, u)$ stores the registers' values and input data in the input u to the register R_0. The functions $get(R_0, v)$ and $get(R_0, R_h)$ return the latest values of the input variable v and the register R_h stored in R_0, respectively.

In Fig. 4, if the guards are "true", or if the register definition statements are empty, then they are omitted.

5 Derivation of Protocol Specifications

5.1 Basic Idea

Basically, we construct the protocol specification as follows. For each transition t in a given SS, we construct a sub-Petri net $SP^k(t)$ for each PE_k that simulates the transition t. A protocol entity specification PE_k is constructed by replacing each transition t in SS with the corresponding sub-Petri net $SP^k(t)$.

In [6], a method to simulate a transition t in SS described as an EFSM model is proposed where the number of exchanged messages is minimized. We use the similar method to construct $SP^k(t)$. In [6], since parallel events are not permitted, they are executed sequentially. However, in the PNR model, those events can be executed in parallel. In Section 5.2, we explain the details of this simulation method.

In general, in order to derive protocol specifications in the PNR model, we have to solve the following problems.

P1. Many transitions in SS may fire in parallel and several messages simulating those transitions may be sent to a protocol entity at the same time through the same communication channel. Those messages should be distinguished.

P2. Complex non-deterministic selective structures can be described in a general PNR model. At the protocol specification level, many protocol entities must participate to make a non-deterministic selection. It is difficult to implement such a complex selective structure in the distributed environment. Therefore, we have to find a suitable sub-class of the PNR model that provides a simple selective structure in the distributed environment.

P3. In order to keep the consistency of the temporal ordering of execution of the transitions, the sub-Petri net $SP^k(t')$ simulating the next transition t' should start to fire after all of the transitions in the sub-Petri net $SP^k(t)$ simulating the current transitions t have finished to fire. Otherwise, the derived protocol specification may not work correctly.

P4. If a protocol entity k is not concerned with a transition t in SS, then t will be replaced by an ε-transition in the PE_k to simplify the derivation and to apply the method in [6] for our method. The derived protocol specification is equivalent to a given service specification if the firing rule of ε-transitions is defined in the same as that of ε-moves in the Non-deterministic Finite state Automata (NFA). However, as defined in Section 2.2, an ε-transition can be enabled if it is enabled in its underlying net (because its guard is "true"). Therefore, we must remove such ε-transitions that have wrong effects on the equivalence. In the PNR model, those ε-transitions cannot be removed from the derived protocol specifications easily because those ε-transitions may play synchronous points.

To solve the problem P1, we assume that each message used for simulating a transition t has its identifier. Using the identifier, each protocol entity can know which transition should be simulated by a received message. For the problem P2, we give the restriction 1 in Section 4 (that is, the underlying net of SS must be a FC net). The selection can be always done in one place. Therefore, at the protocol specification level, only one protocol entity can select for each selection. In Section 5.3 and Section 5.4, we give solutions for the problem P3 and P4, respectively.

Table 1: Contents of Messages exchanged in SML.

Types	Contents
1	the values of registers (or input variables)
2	a request for asking to send the values of registers
3	a request for asking to change the registers' values
4	a request for asking to send the values of registers necessary for evaluating the guard or executing the output event of each next responsible protocol entity
5	a notice which informs that the change of registers' values has been finished
6	the values of registers necessary for evaluating the guard or executing the output event of each next responsible protocol entity
7	a notice which informs that the responsible protocol entity has been changed

5.2 Simulation of Each Transition

In this section, we give how each transition in SS is implemented with $PE^{\langle 1,p \rangle}$. Here, we denote the simulation principle by **SML**.

In **SML**, we assume that the responsible protocol entity of a transition t (RPE(t)) knows all values of registers used in the guard and/or output event of t (if it is an output event) when the responsible protocol entity starts to simulate the transition t. Since we assume that this assumption holds when each simulation starts, at the end of simulation of t, all information (the values of registers used in guards and output events) necessary for the responsible protocol entities of $t \bullet \bullet$ is sent during the simulation of the current transition. Now, suppose that for the transitions enabled in the underlying net at the current marking M, the responsible protocol entity evaluates these guards, and then chooses non-determinately a transition t to be executed from the enabled transitions. Table 1 shows the types and contents of the messages exchanged in **SML**.

[Simulation Principle **SML**]

- The responsible protocol entity of a transition t (RPE(t)) executes its I/O event.
- The responsible protocol entity RPE(t) sends some messages to the related protocol entities as follows.
  - Each protocol entity with the registers whose values must be changed in the transition t has to know the values of the registers and inputs necessary for changing its registers' values. Those values are sent from RPE(t) if it has them. Those messages are called type 1 messages. If some of those values are not held in RPE(t), then RPE(t) sends the request messages (type 2 messages) to the protocol entities which have those registers.
  - Some protocol entities can change their registers' values by themselves. Type 3 messages are sent to those protocol entities from RPE(t).
  - The next responsible protocol entities of the transition t (the protocol entities in RPE($t \bullet \bullet$)) should know the values of the registers used in their guards and output events. RPE(t) sends type 4 messages to some protocol entities that have such values.
- Each protocol entity which received the type 2 message sends the type 1 messages (including the values of registers) to the protocol entities which need them.
- Each protocol entity which received the type 4 message sends type 6 messages (including the values of registers) to some of the next responsible protocol entities. If such protocol entities have to change the values of the registers to be transmitted, the type 6 messages must be sent after changing the values of registers.
- Each protocol entity which received the type 1 or type 3 messages changes its registers' values. RPE(t) should change its registers' values after executing the I/O event of the transition t.
- Each protocol entity that has changed its registers' values sends the type 5 messages to all of the next responsible protocol entities of t.
- If RPE(t) has never sent the type 1, ..., type 6 messages, then RPE(t) sends the type 7 messages to the next responsible protocol entities of t. These messages are used to inform that the responsible protocol entity has been changed.

The total number of exchanged messages necessary for simulating each transition may not be unique in the case that several messages can be merged into one message and a register's value can be sent by several nodes. Here, we will adopt the total number of exchanged messages necessary for simulating each transition as the communication cost (this cost measure is appropriate in high speed networks) and minimize it. It can be minimized as follows. We introduce some Boolean variables. For example, we introduce a Boolean variable $t_1_uv_R_h$ whose value is 1 iff a type 1 message including the value of the register R_h should be sent from the protocol entity u to the protocol entity v. Those Boolean variables are treated as integer variables whose values are either 0 (false) or 1 (true). We describe the constraints for those Boolean variables as the integer linear inequalities. For example, if a non-responsible protocol entity u must send a type 1 message containing the register R_h's value to a protocol entity v, the protocol entity u must receive a type 2 message from the responsible protocol entity, say w, for asking to send R_h's value. This constraint is described as an integer inequality such as $t_1_uv_R_h \leq t_2_wu$. We regard the total number of exchanged messages as the objective function of the 0-1 integer linear programing problem and calculate its minimum solution. For the details, see [6].

As an example, for the transition t_2 in the service specification SS in Fig. 2 and the resource allocation Θ

in Fig. 3, we will construct the sub-Petri nets $SP^1(t_2)$, $SP^2(t_2)$ and $SP^3(t_2)$ simulating the transition t_2 based on **SML** (see Fig.4). First, since the protocol entity 2 is the responsible protocol entity of t_2 (RPE(t_2)=protocol entity 2), it evaluates the value of the guard of t_2, and executes the event of t_2 (it corresponds to the transition t_a in Fig. 4). the protocol entities 1 and 3 should change the value of the register R_1. They need the value of R_4 for changing their registers' values. Although the protocol entity 1 has it, the protocol entity 3 does not have it. Therefore, the responsible protocol entity 2 sends the protocol entity 1 a message M_b for asking to send the value of R_4 to the the protocol entity 3 (t_b). If the protocol entity 1 receives M_b (t_c), then it sends a message M_c which includes the value of R_4 (t_d) and changes the values of R_1 (t_e) in parallel. If the protocol entity 3 receives M_c (t_f), then it changes the value of R_1 (t_g), and sends the protocol entity 1 (the next responsible protocol entity) a message M_d which informs that the change of the register's value has been finished (t_h). Finally, the next responsible protocol entity 1 receives M_d (t_i).

5.3 Construction of Protocol Entity Specification

In this section, we derive each PE_k by replacing each transition t in SS with the corresponding sub-Petri net $SP^k(t)$ ($SP^k(t)$ may be an ε-transition). Fig. 5 shows that t_2 in SS is replaced by the sub-Petri net $SP^1(t_2)$ in PE_1. The details are as follows. Here, the sub-Petri net $SP^1(t_2)$ has two types of special transitions. The first type of transitions is called a source transition, which has no incoming arcs. The another type of transitions is called a sink transition, which has no outgoing arcs. $SP^1(t_2)$ has two source transitions and three sink transitions. First, for a given t_2 in SS, we make the copies of each input place p_{in} of the transition t_2 as many as the number of the source transitions (in this case, two places) like Fig. 5. We also make the copies of each output place p_{out} of the transition t_2 as many as the number of the sink transitions similarly. Then, we connect an arc from each p_{in} to each source transition. We also connect an arc from each sink transition to each p_{out}. We execute this process for all transitions in the service specification. In the obtained net, each sub-Petri net cannot start executing until all of its previous sub-Petri nets have completely finished executing. Also all source transitions in the sub-Petri net can be enabled simultaneously (from these properties, the problem P3 in Section 5.1 can be solved). However, if a given transition t is in a self-loop, this method does not work well since its input place p_1 and output place p_2 are identical (a place p). In this case, we introduce a new place p' and a dummy transition t' which has an internal event i and a guard "true". Then, we can remove all self-loops.

The initial marking for the derived PE_k is given as follows. If a place in SS is marked with a token, then we also mark each of the corresponding places (including the copied places) in PE_k with a token. Then, we can derive each protocol entity specification PE_k. Since we have made the copies of some places, the underlying net of each PE_k may not be a FC net. However, it still keeps the liveness and safeness properties.

5.4 Reformation of Protocol Specifications

In our method, if a protocol entity k is not related to the simulation of a transition t in the SS, then the transition t is replaced by an ε-transition. So each PE_k may contain ε-transitions.

As described in Section 5.1, the two rules for firing ε-transitions are considered, (1) an ε-transition t is enabled in the PNR model iff t is enabled in its underlying net and there exists a non-ε-transition behind t where the value of its guard is true and (2) an ε-transition t is enabled if t is enabled in its underlying net. For example, in Fig. 6, the ε-transition t cannot be enabled until the message M_a is given from the gate g_{32} (i.e., the value of the guard of t_a becomes true) if we adopt the rule (1). On the other hand, t can be enabled independent of the transition t_a and t_b if we adopt the rule

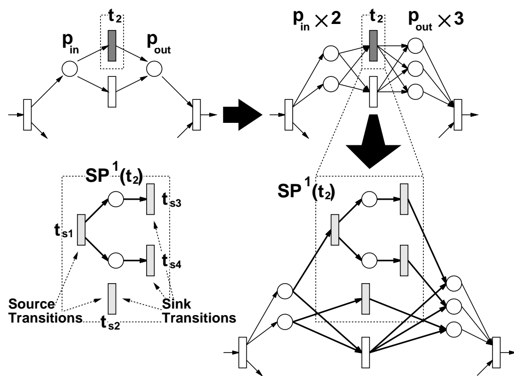

Figure 5: Replacement of a Transition.

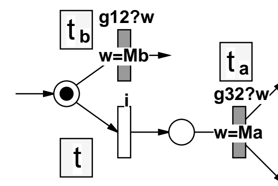

Figure 6: Firability of Each $\mathcal{E}$-Transition t.

(2). If we adopt the rule (1), then $PE^{\langle 1,p \rangle}$ can simulate SS correctly. However, the rule (2) is more natural as the firing rule for general Petri nets. Therefore, we will reform $PE^{\langle 1,p \rangle}$ by removing all ε-transitions so that we can adopt the rule (2).

Removal of ε-Transitions. In general, an ε-transition may represent a synchronous point in a system because more than one incoming arcs and/or outgoing arcs may be connected to an ε-transition. Here, since each sub-Petri net does not contain any ε-transition, we can treat it as a single transition like the original transition in the service specification.

Fig. 7 shows an example to remove the ε-transitions t_4 and t_5 in PE_2. First, we decompose PE_2 (Fig. 7(a)) into two finite state machines SM_1 and SM_2 (nets where each transition has one incoming arc and one outgoing arc) called SM components (Fig. 7(b)). Then, we recompose the SM components SM_1 and SM_2 into a net PE'_2 where the same name's non-ε-transitions in SM_1 and SM_2 are merged into one transition while all ε-transitions are not merged. In this case, only the transitions t_1 are merged. In this net, each ε-transition has one incoming arc and one outgoing arc. This means that all ε-transitions do not play synchronous points in PE'_2. Finally, in the recomposed PE'_2, we delete all ε-transitions t_4 and t_5 (Fig. 7(d)).

Formally, we remove ε-transitions by the following three steps. (S1) We decompose each PE_k into strongly-connected finite state machines (SM components) covering PE_k where each of the SM components has exactly one token at its initial marking M_0. It is known that a live and safe FC net can be always decomposed into such SM components[1]. Although the way to decompose the net is not unique, the algorithm works well when we use the same way for decomposing all protocol entities' specifications. (S2) We recompose the SM components into a net by merging only non-ε-transitions. (S3) Since each ε-transition in a PE'_k has exactly one incoming arc and one outgoing arc, it can be removed by merging its input place and output place into a single place.

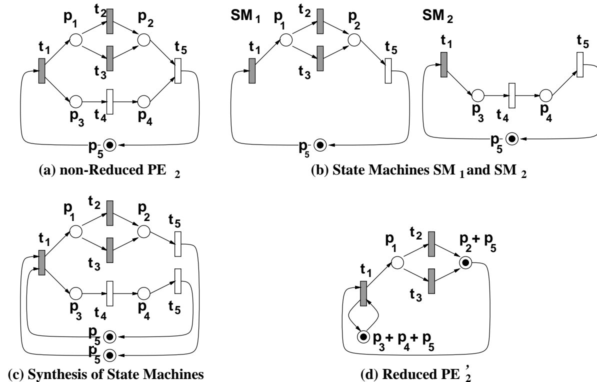

Figure 7: Reformation of PE2.

Removal of Redundant Places. By removing ε-transitions, each PE'_k may contain redundant places, and they can be removed. For example, the redundant place $p_3 + p_4 + p_5$ in Fig. 7(d) can be removed (and then by replacing each transition with the corresponding sub-Petri net, we can get PE_2 in Fig. 4). Some rules to remove such places can be considered, the details are omitted (see [8]).

6 Conclusion

In this paper we have proposed a method to derive protocol specifications from service specifications of distributed systems described in the Petri Net model with Registers (PNR model). Many service specifications of distributed systems can be described naturally in this model since the distributed allocation of resources is permitted and parallel events and non-deterministic selective operations can be described in this model. In [8], we give an example where our algorithm is applied to the "Software Process Modeling Example Problem" given by Marc Kellner in [7] and give the correctness proof of our derivation algorithm. In our algorithm, to simplify the derivation, we restrict the class of the underlying nets of the PNR model to live and safe free-choice nets. The underlying nets of the PNR model for the derived protocol specifications are not free-choice nets, however, selective and synchronous structures in the nets are so simple that the derived protocol specifications can be implemented easily. To show the usefulness of our approach, we are planning to implement our derivation algorithm.

Acknowledgments

The authors would like to thank T. Murata (Univ. of Illinois at Chicago) for his useful comments to this work.

References

[1] Murata, T. : "Petri Nets: Properties, Analysis and Applications," *Proc. of the IEEE*, Vol. 77, No. 4, pp. 541-580 (1989).

[2] Probert, R. and Saleh, K. : "Synthesis of Communication Protocols: Survey and Assessment," *IEEE Trans. on Comp.*, Vol. 40, No. 4, pp. 468-476 (1991).

[3] Gotzhein, R. and Bochmann, G. v. : "Deriving Protocol Specifications from Service Specifications Including Parameters," *ACM Trans. on Comp. Syst.*, Vol. 8, No. 4, pp. 255-283 (1990).

[4] Chu, P.-Y.M. and Liu, M.T. : "Protocol Synthesis in a State-Transition Model," *Proc. of COMPSAC '88*, pp. 505-512 (1988).

[5] Milner, R. : "Communication and Concurrency," *Prentice-Hall* (1989).

[6] Higashino, T., Okano, K., Imajo, H. and Taniguchi, K. : "Deriving Protocol Specifications from Service Specifications in Extended FSM Models," *Proc. of 13th IEEE Int. Conf. on Distributed Computing Systems (ICDCS-13)*, pp. 141-148 (1993).

[7] Kellner, M. et al. : "ISPW-6 Software Process Example," *Proc. of the 1st Int. Conf. on the Software Process*, pp. 176-186 (1991).

[8] Yamaguchi, H., Okano, K., Higashino, T. and Taniguchi, K. : "Synthesis of Protocol Entities' Specifications from Service Specifications in a Perti Net Model with Registers," *I.C.S Research Report*, 95-ICS-1 (1995).