Protocol synthesis and re-synthesis with optimal allocation of resources based on extended Petri nets

Hirozumi Yamaguchi[1], Khaled El-Fakih[2], Gregor von Bochmann[3], Teruo Higashino[1]

[1] Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
(e-mail: {h-yamagu,higashino}@ist.osaka-u.ac.jp) [*]
[2] Department of Computer Science, American University of Sharjah, Sharjah, United Arab Emirates
(e-mail: kelfakih@aus.ac.ae) [**]
[3] School of Information Technology and Engineering, University of Ottawa, Ottawa, Canada
(e-mail: bochmann@site.uottawa.ca)

Summary. Protocol synthesis is used to derive a protocol specification, that is, the specification of a set of application components running in a distributed system of networked computers, from a specification of services (called the service specification) to be provided by the distributed application to its users. Protocol synthesis reduces design costs and errors by specifying the message exchanges between the application components, as defined by the protocol specification. In general, maintaining such a distributed application involves applying frequent minor modifications to the service specification due to changes in the user requirements. Deriving the protocol specification after each modification using the existing synthesis methods is considered expensive and time consuming. Moreover, we cannot identify what changes we should make to the protocol specification in correspondence to the changes in the service specification. In this paper, we present a new synthesis method to re-synthesize only those parts of the protocol specification that must be modified in order to satisfy the changes in the service specification. The method consists of a set of simple rules that are applied to the protocol specification written in an extended Petri net model. An application example is given along with some experimental results.

Key words: Distributed system, Protocol derivation, Petri net

1 Introduction

Synthesis methods have been used (for surveys see [7, 8]) to derive an implementation level's specification of a distributed system (hereafter called *protocol specification*) automatically from a given specification of services to be provided by the distributed system to its users (called *service specification*). The service specification is written as a program of a centralized system, and does not contain any message exchange between different physical locations. However, the implementation level's specification of the cooperating programs, called *protocol entities* (*PE's*), includes the message exchanges between these entities. Therefore, protocol synthesis methods have been used to specify and derive such complex message exchanges automatically in order to reduce the design costs and errors that may occur when manual methods are used.

A number of protocol synthesis strategies have been described in the literature. The first strategy aims at implementing complex control-flows using different computational models such as CCS based models [5,6], LOTOS [10,11], Petri nets [16,17,20] and FSM/EFSM [12, 14]. The second strategy, [21–23, 25–27], aims at satisfying the timing constraints specified by a given service specification in the derived protocol specification. This strategy deals with real-time distributed systems. The last strategy, [9,13,18,19,24,28,29,31], deals with the management of distributed resources such as files and databases. The objective is to determine how the values of these distributed resources are updated or exchanged among PE's for a given resource allocation.

Some methods in the last strategy, especially in our previous research work [29], consider an efficient implementation of a given service specification by deriving the corresponding protocol specification with minimum communication costs and optimal allocation of resources. This work considers an optimal resource allocation to reduce the costs of message exchanges when we derive protocol specifications. As an example, we considered a software development process using a Computer Supported Cooperative Work (CSCW) environment. This process is carried out cooperatively by multiple engineers (developers, designers, managers and others). Each engi-

neer has his/her own workstation (PE) and participates in the development process using specific distributed resources (*e.g.* drafts, source codes, object codes, multimedia video and audio files, and others) which may be placed on different computers. Considering the need for managing such a process in the distributed environment, we describe the whole software development process (service specification) and derive the set of all the engineers' sub-processes (protocol specification). We also determine an optimal allocation of resources that would minimize the communication costs (such as file transfer costs).

In realistic applications, maintaining such a system involves modifying its specification as a result of changes of the user requirements. Moreover, developers usually implement incrementally the given specification. Synthesizing the whole system again after each minor modification is considered expensive and time consuming, especially for large-scale distributed systems with large number of users distributed over multiple sites.

In this paper, we propose a new technique for system maintenance called *protocol re-synthesis*. For a given service specification S, its corresponding protocol specification P and a modified service specification S', our re-synthesis algorithm produces the changes in the protocol specification ΔP corresponding to the changes ΔS $(= S' - S)$ in the service specification, and derives a modified protocol specification P' $(= P + \Delta P)$ corresponding to S'. A computer supported cooperative software development process is used again as an example to show that the method reduces the cost of deriving the revised protocol specification after each change in the service specification.

The primary goal of our approach is to save maintenance costs of distributed systems, and to provide a way to specify requirement changes in a natural manner. To this end, our re-synthesis algorithm decomposes the changes ΔS into a sequence of atomic changes and sequentially applies their corresponding *re-synthesis rules* to P to derive P'. Since each re-synthesis rule is designed to modify as small a part of P as possible, ΔP can be small and deriving P' is simple enough compared with normal protocol synthesis methods, which often consume much computational resources to derive P' directly from S'. Moreover, in contrast to Ref. [32] that considers requirement changes at the protocol level (in one of the protocol entities), our re-synthesis method is a service-based approach.

In our previous work presented in [30], we have stated the basic principle of our re-synthesis method where only simple modification cases are considered. In this paper, the method has been considerably extended so that we can deal with more realistic service modifications that need changing the allocation of resources, insertion or deletion of tasks and so on.

This paper is organized as follows. Section 2 gives examples of service specifications and protocol specifications. Section 3 describes the overview of our original protocol synthesis method. Based on this method, we propose a re-synthesis method in Section 4. Some application examples are given in Section 5. Section 6
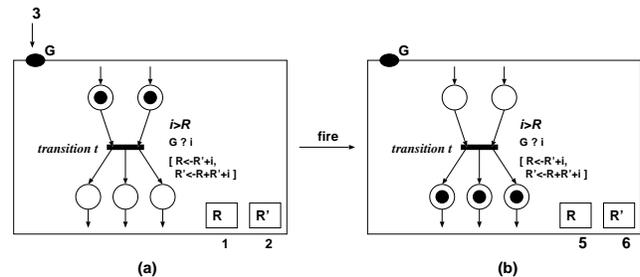
Fig. 1. Register values and token location before and after firing transition in PNR

concludes this paper and includes our insights for future research.

2 Service specifications and protocol specifications

2.1 Petri net model with registers

We use an extended Petri net model called *Petri Net with Registers* (*PNR* in short) [18] to describe both service specifications and protocol specifications of distributed systems. In this model, the service access points between the users and the system are modeled as gates, and the variables used inside the system, such as databases and files, are modeled as registers. Each transition t in a PNR has a label $\langle \mathcal{C}(t), \mathcal{E}(t), \mathcal{S}(t) \rangle$, where $\mathcal{C}(t)$ is a precondition (the firing condition of t), $\mathcal{E}(t)$ is an I/O event and $\mathcal{S}(t)$ is a set of assignment statements (which represent parallel updates of register values).

A transition t may fire if (a) each of its input places has a token, (b) the value of $\mathcal{C}(t)$ is true and (c) an input value is given through the gate in $\mathcal{E}(t)$ if $\mathcal{E}(t)$ is an input event. If t fires, the corresponding I/O event is executed, and the new values of registers are calculated and substituted in parallel as defined by $\mathcal{S}(t)$.

Consider, for example, transition t of Fig. 1(a) where $\mathcal{C}(t) =$"$i > R$", $\mathcal{E}(t) =$"$G?i$" and $\mathcal{S}(t) =$"$\{R \leftarrow R' + i, R' \leftarrow R + R' + i\}$". Here, i denotes an input variable which holds an input value. The input value can be referred to only in this transition t. R and R' denote registers which contain values, and their values may be used and updated by all the transitions of the PNR. This means that registers are treated like global variables. G is a gate, that is, a service access point (interaction point) between users and the system. Note that "?" or "!" in $\mathcal{E}(t)$ indicate that $\mathcal{E}(t)$ is an input or output event, respectively.

Assume that an integer of value 3 has been given as input through gate G, and the current values of the registers R and R' are 1 and 2, respectively. In this case, since the value of the pre-condition "$i > R$" is true, the transition may fire. If it fires, event "$G?i$" is executed and the input value 3 is assigned to the input variable i. Then the assignments "$R \leftarrow R' + i$" and "$R' \leftarrow R + R' + i$" are executed in parallel. After the firing, the tokens are
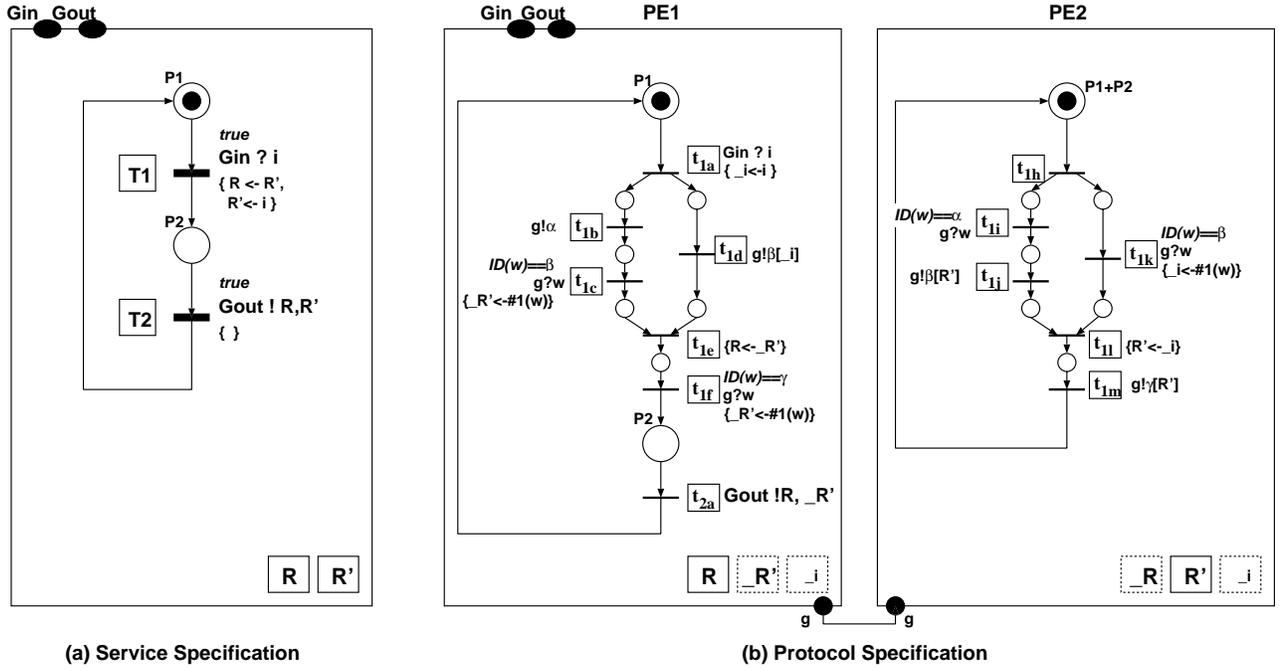
Fig. 2. Service specification and protocol specification

moved, and the values of the registers R and R' are 5 ($= 2 + 3$) and 6 ($= 1 + 2 + 3$), respectively (Fig. 1(b)).

Formally, an I/O event $\mathcal{E}(t)$ is one of the following types: "$G\ !exp$", "$G\ ?i$", or "τ". "$G\ !exp$" is an output event, and it means that the value of the expression "exp" is output through the gate G (all the arguments in "exp" must be registers). "$G\ ?i$" is an input event and it means that the value given through G is assigned to the input variable "i". The event "τ" means that no external I/O event is associated with this transition. $\mathcal{S}(t)$ is a set of assignment statements, each of which has the form "$R \leftarrow exp$" where R is a register and exp is an expression whose arguments may be the input variable of $\mathcal{E}(t)$ or registers.

PNR is defined as a tuple $\langle T,P,A,M_0,G,R,I,\mathcal{C},\mathcal{E},\mathcal{S},R_0\rangle$ where $\langle T,P,A,M_0\rangle$ is a Petri net, G is a set of gates, R is a set of registers, and I is a set of input variables. $\mathcal{C}$, $\mathcal{E}$ and $\mathcal{S}$ define the labels of transitions as explained above, and R_0 defines the initial values of the registers.

2.2 Service specifications

At an abstract level, a distributed system is regarded as a non-distributed system which provides services as a single "virtual" machine. The number of actual PE's and communication channels between them are hidden. A specification of a distributed system at this level is called a *service specification* and denoted by *Sspec* in this paper. Although the actual resources of a distributed system may be located on different physical machines, called protocol entities, the service specification, at this level, considers only one virtual machine.

For better readability and understanding, hereafter, we use the simple example of *Sspec* shown in Fig. 2(a). A larger practical example is given in Section 5. *Sspec* in Fig. 2(a) uses two gates G_{in} and G_{out} and two registers R and R'. At the initial marking, one token is assigned to place P_1, and therefore T_1 can fire if an input is given through G_{in}. When T_1 fires, the system updates the values of the registers R and R' simultaneously using the current values of register R' and the input i, respectively[1]. Then T_2 fires, and the system outputs the updated values of R and R' through gate G_{out} and returns to the initial marking.

2.3 Protocol specifications

A distributed system may be considered as a communication system which consists of n protocol entities PE_1, PE_2, ... and PE_n. We assume a duplex and reliable communication channel between any pair of PE's (PE_i and PE_j). The PE_i and PE_j sides of the communication channel are represented as gates g_{ij} and g_{ji}, respectively. Moreover, we assume that all the registers and the gates for communication with the users are allocated to certain PE's in the distributed system.

Two PE's communicate with each other asynchronously by exchanging messages. A message is denoted by "M[list of values]" where "M" is one of the following three message types (α, β or γ) explained later. We assume that if PE_i executes an output event "$g_{ij}!M$[list of values]" on a transition, this message is sent through gate g_{ij} to the peer protocol entity PE_j.

---

[1] At the first firing of T_1, the initial value of R' is used to update R.
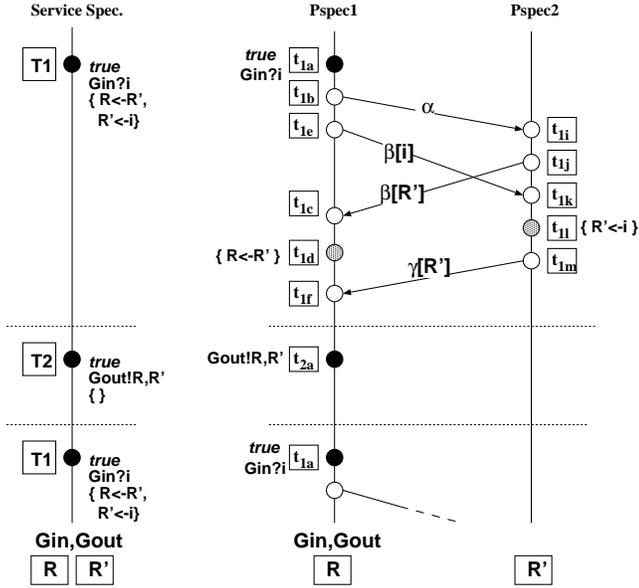
Fig. 3. Service specification and protocol Specification : timing charts

On reception, it is written into PE_j's receive buffer. If PE_j executes an input event "$g_{ji}?w$" with pre-condition "$ID(w) == M$" on a transition, PE_j removes the received message from its buffer and the message is kept in the input variable w. Note that the i-th value of the list included in the received message w will be denoted by $\#i(w)$.

In order to implement a distributed system which consists of n PE's, we must specify the behavior of these PE's. A behavior specification of PE_k is called a *protocol entity specification* and denoted by $Pspec_k$. A set of n protocol entity specifications is called a *protocol specification* and denoted as $Pspec_{1..n}$. We need a protocol specification in order to implement a given service specification.

Let us assume that there are two PE's (PE_1 and PE_2) in order to implement the service specification of Fig. 2(a). We also assume that PE_1 has both user gates G_{in} and G_{out} and register R, while PE_2 has register R'. Fig. 2(b) shows an example of $Pspec_{1..2}$ which provides the services of Fig. 2(a), based on the above allocation of resources. Note that some additional registers, called *temporary registers*, are used in this protocol specification to temporary keep values received in messages. If PE_i receives the value of register R from some other PE via a message, we assume a temporary register "$\_R$" (represented as a dotted box in the figure) to keep the value on PE_i. In Fig. 2(b), for simplicity of notations, both g_{12} and g_{21} are denoted as g, and internal events "τ", pre-conditions "true" and empty sets of assignment statements are omitted.

According to the protocol specification of Fig. 2(b) and its corresponding timing charts in Fig. 3, PE_1 first receives an input through G_{in} and checks the values of the firing condition $\mathcal{C}(T_1)$ on t_{1a}. Since it is always true, PE_1 executes $\mathcal{E}(T_1)$ on t_{1a} and keeps the received input

i in the temporary register $\_i$. Then it sends a message "α" during transition t_{1b} in order to ask PE_2 to send the current value of R' which is necessary to update the value of R. PE_2 receives the message during transition t_{1i} and sends a message "$\beta[R']$" during transition t_{1j}. PE_1 receives the message during t_{1c} and now knows the value of R'. In parallel with the sending of the α-message, PE_1 sends the message "$\beta[\_i]$" during t_{1d} thereby sending the value of the input "i" to PE_2. PE_2 receives the message during t_{1k} and now knows the value of the input. After sending/receiving the β-messages, PE_1 and PE_2 know that now they can execute $\mathcal{S}(T_1)$ using the received values. They independently execute "$R \leftarrow R'$" and "$R' \leftarrow i$" during the transitions t_{1e} and t_{1l}, respectively. After the firing of t_{1e} and t_{1l}, the system is in a state where the service specification should check whether transition T_2 would be executed. For that purpose, PE_2 sends a message "$\gamma[R']$" in order to send the (updated) value of R' and let PE_1 know that the execution of "$R' \leftarrow i$" had been completed. When receiving the γ-message, PE_1 is ready to start the execution of T_2. After executing $\mathcal{E}(T_2)$ on t_{2a}, both PE_1 and PE_2 are back in their initial markings.

As shown in the above example, two PE's cooperate with each other in order to provide the same event sequences (including values) at the user gates G_{in} and G_{out} as specified in $Sspec$. Moreover, our synthesis method described below guarantees that the values of a register in $Sspec$ and $Pspec_{1..2}$ are identical and the buffers of all the communication channels are empty at corresponding markings. For example, the marking of $Sspec$ where place P_2 has a token and the marking of $Pspec_{1..2}$ where place P_1 of PE_1 and place "$P_1 + P_2$" of PE_2 have tokens are such corresponding markings. This is because our implementation never starts the execution of a transition unless the execution of all the previous transitions have been completed, and it allows us to easily keep consistency between $Sspec$ and $Pspec_{1..2}$. It also allows us to use receive buffers of finite capacity for the communication channels, since we can determine the maximum number of messages that may be in transmit between any pair of protocol entities.

3 Synthesis overview

We have presented in our previous work, especially in [29], the protocol synthesis method which is the basis for and thus highly relevant to the re-synthesis method presented in this paper. In order for the complete understanding of the re-synthesis method with the conviction of correctness, it is important to present our synthesis method designed to be suitable to the re-synthesis method.

The synthesis method derives a protocol specification from a given service specification and is based on a set of *synthesis rules* that specify how to execute each transition $T = \langle \mathcal{C}(T), \mathcal{E}(T), \mathcal{S}(T) \rangle$ of the service specification by the corresponding PE's in the protocol specification. Based on these rules, the behavior of all PE's and an optimal allocation of resources (registers and user gates)

Table 1. Notation

Notation	
$\mathcal{C}(T), \mathcal{E}(T), \mathcal{S}(T)$	pre-condition, event and set of assignment statements of T
$PEstart(T)$	the PE where the gate used in $\mathcal{E}(T)$ is located
$PEsubst(T)$	the set of those PE's that contain a register updated by $\mathcal{S}(T)$
$PEstart(T \bullet \bullet)$	the set of the PE's that start the execution of the next transitions after T (i.e. $\bigcup_{T' \in T \bullet \bullet} PEstart(T')$ where $T \bullet \bullet$ is the set of the next transitions after T)

for minimum communication costs can be determined. This leads to the specifications of all the PE's (protocol entity specifications) written in the same PNR model formalism.

3.1 Synthesis rules

For executing a transition $T = \langle \mathcal{C}(T), \mathcal{E}(T), \mathcal{S}(T) \rangle$ of the service specification by a set of transitions of the PE's in the protocol specification, we use the following algorithm. Fig. 4 shows how our algorithm is applied to transition T_1 of $Sspec$ of Fig. 2(a). Note that the notation used in the following algorithm is summarized in Table 1.

- The PE that has the gate G used in $\mathcal{E}(T)$ (which we denote by $PEstart(T)$) decides to start the execution of T by checking the value of the pre-condition $\mathcal{C}(T)$. If it is true, $PEstart(T)$ executes the event $\mathcal{E}(T)$.

  In Fig. 4, $PEstart(T_1)$ is PE_1 since PE_1 has gate G_{in}. PE_1 checks the value of $\mathcal{C}(T_1)$="$true$" (always true) and then executes $\mathcal{E}(T_1)$="$G_{in}?i$".

- Then, $PEstart(T)$ sends synchronization messages called α-messages to those PE's that have the registers used to execute the assignment statements in $\mathcal{S}(T)$. On reception, those PE's send the register values to the PE's that execute assignment statements whose expressions require those register values. Note that α-messages are also sent to some other PE's. This is explained later.

  In Fig. 4, we assume that "$R \leftarrow R'$" and "$R' \leftarrow i$" are executed by PE_1 and PE_2 respectively, since PE_1 has R and PE_2 has R' (see Section 3.2 for the discussion of this allocation problem). For this execution, PE_1 needs the value of R' and PE_2 needs the value of i. Here, since PE_1 does not have "R'", the value must be sent from PE_2. An α-message is sent from $PEstart(T_1) = PE_1$ to PE_2 in order to let PE_2 send the value of R'. Also, since PE_2 does not know the value of i, it must be sent from PE_1 to PE_2. Here, since PE_1 is itself $PEstart(T_1)$, it knows the timing to send the value of "i" (just after the execution of $\mathcal{E}(T_1)$). Therefore no α-message is sent from PE_1 to itself.

- On the reception of an α-message, the PE sends the values of registers to those PE's that need those values for the execution of part of assignment statements in $\mathcal{S}(T)$ (the set of these PE's is denoted as $PEsubst(T)$). These messages are called β-messages. Using those register values, each PE in $PEsubst(T)$ executes the assignment statements. Note that if a PE in $PEsubst(T)$ does not need any register value for the execution of the assignment statements, $PEstart(T)$ directly sends a β-message to the PE.

  In the example of Fig. 4, we have $PEsubst(T_1) = \{PE_1, PE_2\}$ since PE_1 has register R and PE_2 has R'. PE_1 sends a β-message to PE_2 with the value of i, which is used by PE_2 to execute "$R' \leftarrow i$". Moreover, PE_2 sends a β-message to PE_1 with the value of R', which is used by PE_1 to execute "$R \leftarrow R'$". Then PE_1 and PE_2 execute "$R \leftarrow R'$" and "$R' \leftarrow i$", respectively.

- After all assignment statements in $\mathcal{S}(T)$ are executed, each PE in $PEsubst(T)$ sends so-called γ-messages to those PE's that will start the execution of the next transitions. The set of those PE's is denoted as $PEstart(T \bullet \bullet)$. These messages confirm the completion of the assignment statements and also contain the values of registers necessary to start the execution of next transitions. Note that the PE's that do not belong to $PEsubst(T)$ may also need to send some values of registers to the PE's in $PEstart(T \bullet \bullet)$. These values are also sent as γ-messages. In this case, α-messsages are sent to these PE's to initiate the sending of γ-messages.

  In our example, PE_2 sends a γ-message to $PEstart(T_1 \bullet \bullet) = \{PE_1\}$. PE_1 then knows the value of R' and the fact that the execution of $\mathcal{S}(T_1)$ on PE_2 has been completed.

The above algorithm is presented as a set of rules called *synthesis rules* (see Appendix A). The synthesis rules are classified into action rules and message rules. The action rules specify which PE's should check the pre-condition, execute the I/O event and assignment statements of T. The message rules specify which PE's should exchange messages. The contents and types of these messages are also specified.

Consequently, three types of messages are exchanged for the execution of a transition T:

- α-**messages** are sent from the PE that starts the execution of T (*i.e.* $PEstart(T)$) to the PE's that send β-messages (and PE's that send γ-messages and are not in $PEsubst(T)$). Their reception leads to the sending of β-messages and/or γ-messages. An α-message does not contain any register value.
- β-**messages** are sent from PE's that have registers to be used to execute assignment statements of $\mathcal{S}(T)$, to those PE's that execute these assignment statements. The latter PE's form the set $PEsubst(T)$. Note that for the PE's that need no register values for the execution of the assignment statements, β-messages are sent from $PEstart(T)$ for synchronization. The reception of β-messages leads to the execution of the assignment statements.
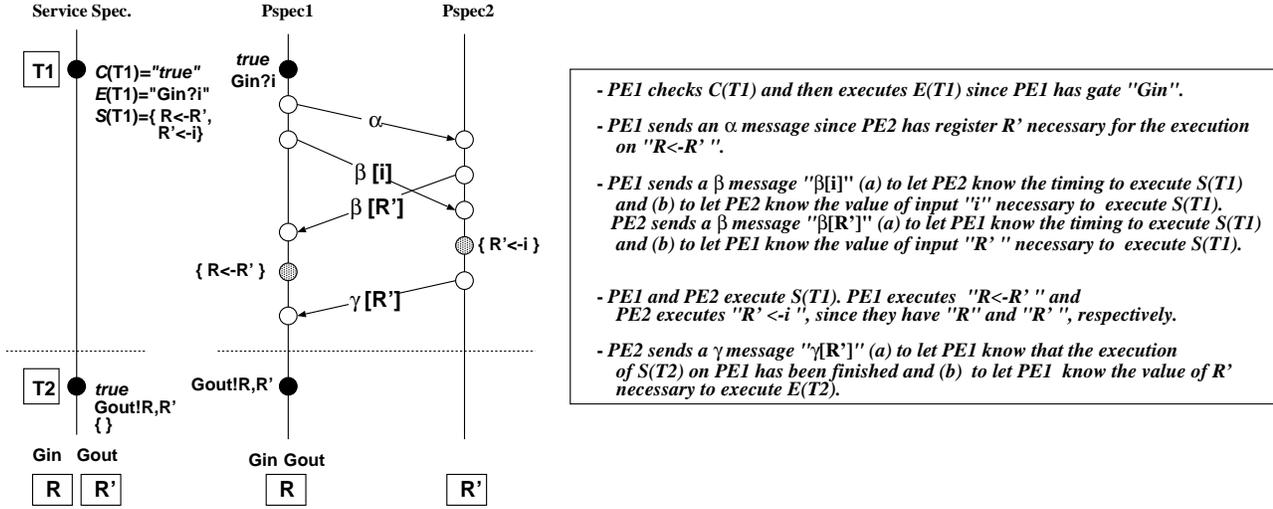
Fig. 4. Applying synthesis rules to T_1

- γ-**messages** are sent from the PE's in $PEsubst(T)$ and PE's that have registers to be used to check/execute $\mathcal{C}(T')/\mathcal{E}(T')$ to PE's in $PEstart(T\bullet\bullet)$, where T' is a next transition after T. They let the PE's in $PEstart(T\bullet\bullet)$ know the values of registers required for $\mathcal{C}(T')/\mathcal{E}(T')$ and the timing for executing the next transition.

Our synthesis method assumes that the Petri net of the service specification is a live and safe *free-choice net*[1,2]. A free-choice net is a sub-class of Petri nets which has simple choice structures. It is known that a live and safe free-choice net can be decomposed into a set of finite state machines [1,2] and this property is used in our algorithm. In addition, we assume that for two transitions T and T' of $Sspec$ in a choice structure, $PEstart(T) = PEstart(T')$ (*i.e.* the gates in $\mathcal{E}(T)$ and $\mathcal{E}(T')$ are allocated to the same PE). This guarantees that a single PE makes the decision to select the next transition in the choice structure. Otherwise an agreement would be needed among several PE's to make this decision. This would be done by implementing a leader election algorithm as the one shown in [4]. Finally, it is assumed that for two transitions T and T' of $Sspec$ that may be executed in parallel, there is no register that is updated by one and referred or also updated by another. This assumption is used to prevent the inconsistency that may result in having multiple accesses to the same register. This assumption may also be relaxed by implementing a mutual exclusion algorithm (see for instance [4]).

3.2 Optimal resource allocation for minimum communication costs

The above synthesis rules assume that an allocation of user gates and registers to PE's is given. However, the communication costs (especially the number of messages) depend on this allocation. Therefore we may carefully design this allocation in order to minimize the communication costs.

As a simple example, let us consider the timing charts in Fig. 5(b). This chart is the same as in Fig. 4 and obtained when R and R' are allocated to PE_1 and PE_2, respectively. If we use another allocation where both R and R' are allocated to PE_2, we obtain a different protocol specification whose timing chart is shown in Fig. 5(c). We note that the allocation of the user gates are usually fixed by the nature of the application, and therefore cannot be changed freely. These examples show that the resource allocation affects the communication costs of the protocol specifications and that it is not easy to find an optimal allocation, given the complex message exchanges between the PE's.

We can formulate this optimal resource allocation problem as an Integer Linear Programming (ILP) problem. For this purpose, we introduce 0-1 integer (boolean) variables, which represent the fact that (a) a message (of type α, β or γ) is sent from one PE to another, (b) a message contains the value of a given register, or (c) a user gate or a register is allocated to a given PE. For example, a 0-1 integer variable "$\alpha_{i,j}^x$" is introduced for PE_i, PE_j and transition T_x of $Sspec$, whose value is one *iff* an α-message is sent from PE_i to PE_j during the execution of transition T_x, otherwise zero. A 0-1 integer variable "$ALC_i(R_w)$" is introduced for register R_w and entity PE_i, whose value is one *iff* register R_w is allocated to PE_i, otherwise zero.

Using these variables, we define an objective function that minimizes the communication cost (*e.g.* the number of messages) and linear inequalities that represent the synthesis rules. For example, an inequality $\alpha_{i,j}^x \geq \beta_{j,k}^x$ represents the fact that if the value of $\beta_{j,k}^x$ is one, the value of $\alpha_{i,j}^x$ must be one, which corresponds to the synthesis rule "a PE that sends β-messages must receive an α-message". Then an optimal resource allocation is obtained as the solution of the ILP problem. Note that a general ILP problem is known to be a hard problem of

(a) Service Specification **(b) Protocol Specification on a Resource Allocation** **(c) Protocol Specification on another Resource Allocation**
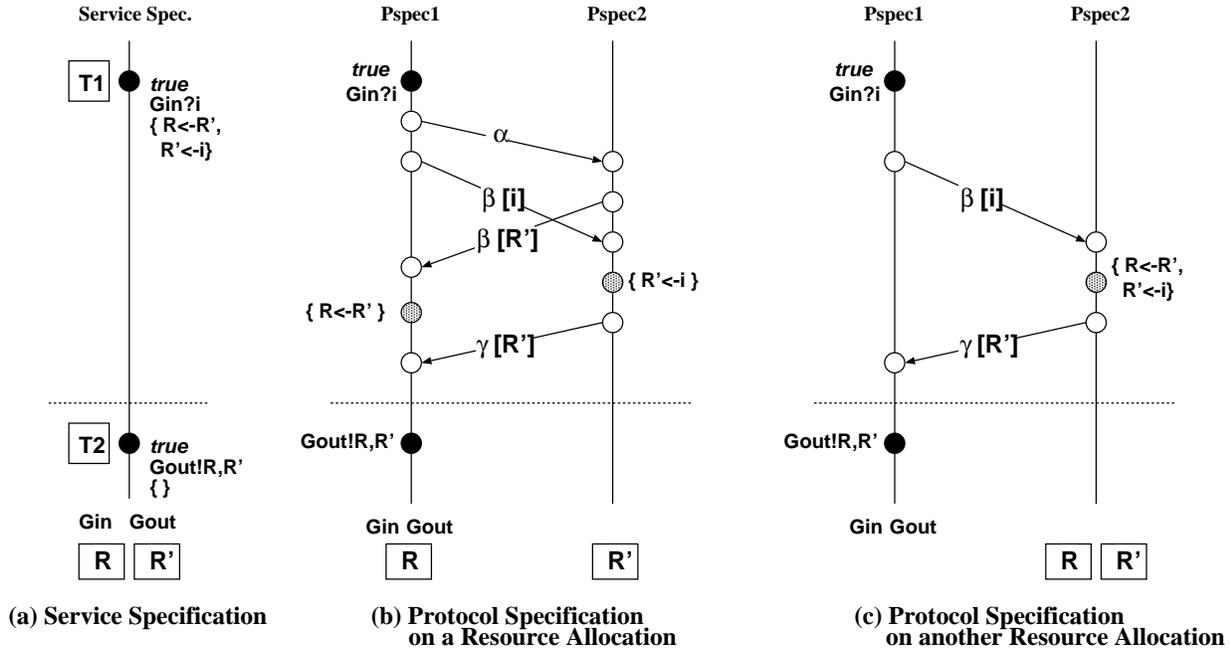
Fig. 5. Two protocol specifications: they provide the same service as the service specification, however their resource allocations are different

exponential complexity, and therefore a solution to our optimization problem is not readily available for large-scale systems in terms of the numbers of transition, resources and PE's. There are some possibilities to tackle this complexity: (i) we may adopt partially fixed allocations to reduce the computation time. If the sizes of the values of some registers are small enough and these registers are rarely accessed, fixing their allocation may not cause significant performance degradation. (ii) We may use heuristic algorithms, for instance genetic algorithms as in [28]. This is part of our future work.

We note that different optimization criteria may be considered by adopting corresponding objective functions. Instead of considering the number of messages, we may also consider the size and/or the cost of sending messages over particular communication channels. The reader may refer to [29] for the details of the problem formulation and cost criteria.

3.3 Synthesis of protocol specifications

We derive a protocol specification using the following three steps.

[Step1] Based on the synthesis rules, a set of actions and message exchanges to be executed on each PE is defined for each transition T of $Sspec$. Then these actions and message exchanges are represented, for each PE, as a set of transitions where two transitions are connected through a place if a temporal ordering between the transitions is specified in the synthesis rules (*e.g.* an α-message must be received before the corresponding β-messages are sent). As a result, for each transition T in $Sspec$, a set of sub-PNR's $SPnet_1(T)$, ..., $SPnet_n(T)$

are produced for the n protocol entities. For example, for transition T_1 of $Sspec$ in Fig.2(a), $SPnet_1(T_1)$ consists of $t_{1a},...,t_{1f}$, and $SPnet_2(T_1)$ consists of $t_{1h},...,t_{1m}$ in Fig.2(b). Note that $SPnet_i(T)$ may be an empty net in the case that PE_i has no action and no message exchange related to the execution of T. In this case, we suppose that the sub-PNR has only a single ε-transition with a label $\{true,``\tau",\{\}\}$ where τ is an internal event. For example, $SPnet_2(T_2)$ is an empty net and therefore it only has one ε-transition.

[Step2] An intermediate protocol entity specification of PE_i (denoted as $\overline{Pspec_i}$) is derived by connecting all sub-PNR's $SPnet_i(T)$ in the same way as the transitions T are connected in $Sspec$. More specifically, $Pspec_i$ is obtained using the net structure of $Sspec$, by replacing each T with the corresponding sub-PNR $SPnet_i(T)$. Note that if $SPnet_i(T)$ has more than one head (or tail) transition, an ε-transition is attached as its head (or tail) transition so that the sub-PNR can be treated like a single transition. $\overline{Pspec_2}$ is shown in Fig. 6(a).

[Step3] Finally, a protocol entity specification of PE_i ($Pspec_i$) is derived by removing ε-transitions from $\overline{Pspec_i}$. The removing technique is based on the well-known technique to remove ε-moves in finite automata. In order to apply this technique to our PNR model containing parallel synchronization, we use the fact that a live and safe free-choice net can be decomposed into a set of live and safe finite state machines (FSM's) [1,2]. This simplification algorithm proceeds as follows. First, for each ε-transition that has u input places and v output places, $(LCM(u,v)/u)-1$ copies of each input place are produced and $(LCM(u,v)/v)-1$ copies of each output place are produced, where $LCM(u,v)$ is the least common multiple of u and v. This replacement never
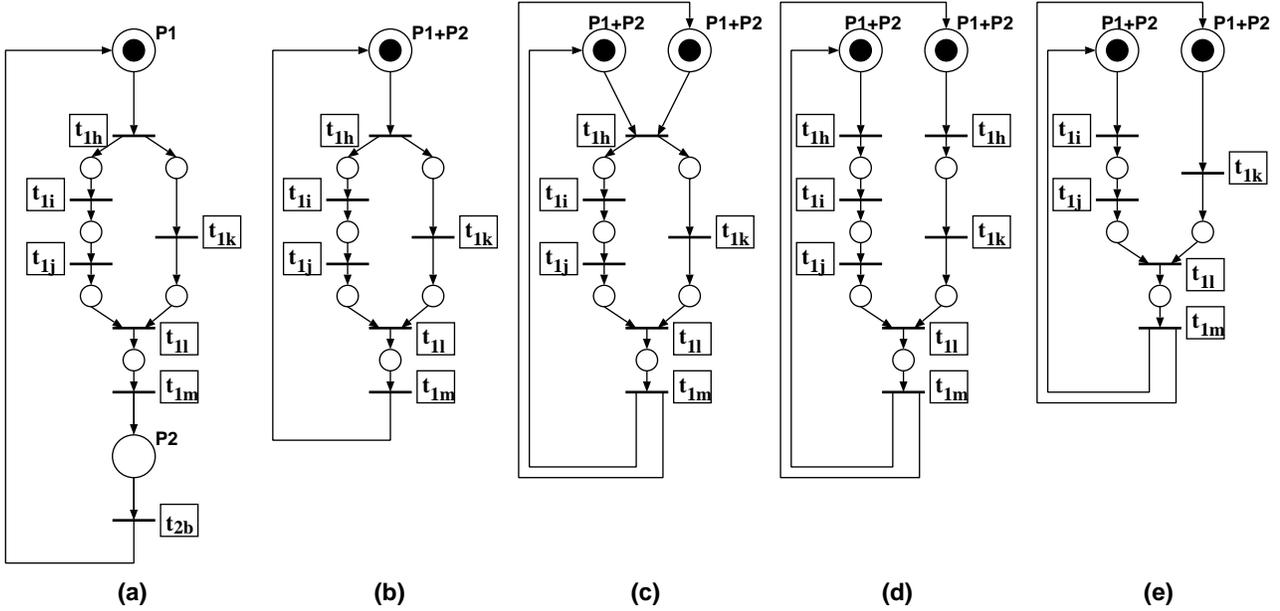
Fig. 6. Removing ε-transitions in $Pspec_2$

changes the behavior of the net and shows that the number of FSM's which synchronize on the ε-transition is $LCM(u, v)$. Then we split these FSM's by splitting the ε-transition and at this moment the ε-transition is no longer a synchronization point. Then the ε-transition is removed using the technique to remove ε-moves in FSM's. For example, in the intermediate specification of PE_2 in Fig. 6(a), t_{1h} and t_{2b} are ε-transitions. Here, t_{2b} can be easily removed by merging places P_1 and P_2 into a single place "$P_1 + P_2$" as shown in Fig. 6(b). Then the input place "$P_1 + P_2$" of the ε-transition t_{1h} is copied as shown in Fig. 6(c). This means that two FSM's synchronize on t_{1h}. Then t_{1h} is split into two transitions (Fig. 6(d)) and removed (Fig. 6(e)). Note that the specification of Fig. 2(b) includes t_{1h} in order to make the correspondence with the service specification more direct.

4 Protocol re-synthesis

In this section, we present our new method for re-synthesizing a protocol specification. Given a service specification $Sspec$, a corresponding protocol specification $Pspec_{1..n}$ and modified service specification (denoted by $Sspec'$), our method derives the corresponding modified protocol specification (denoted by $Pspec'_{1..n}$) by making changes to $Pspec_{1..n}$ only as much as required by the difference between $Sspec$ and $Sspec'$. The advantage of our re-synthesis method is to avoid the applications of the complete protocol synthesis algorithm for each minor change of the requirements which frequently occurs in software maintenance. Even in the case of a minor change on the service specification, the application of the protocol synthesis algorithm described in Section 3 may lead to a revised protocol specification that is different from the original protocol specification in very

many aspects and the resulting maintenance cost would be high. With our re-synthesis method only the minimal sequence of changes will be introduced to the protocol specification.

Requirement changes on service specifications may generally be complex and of wide variety. In our re-synthesis method, we present a technique to decompose requirement changes into sequence of *atomic changes*, and present *re-synthesis rules* for the atomic changes. This approach allows us to treat a variety of requirement changes in a simple manner. Moreover, since the re-synthesis rules are designed to modify as small a part of the protocol specifications as possible, changes on the protocol specifications will remain limited.

4.1 Atomic changes and re-synthesis rules

We consider the following four types of atomic changes (AC_{1+}), (AC_{2+}), (AC_{3+}) and (AC_{4+}), and their corresponding re-synthesis rules (RS_{1+}), (RS_{2+}), (RS_{3+}) and (RS_{4+}). As an example, we use the specifications $Sspec$ and $Pspec_{1..3}$ shown in Fig. 7(a) as timing charts.

Note that for each atomic change (AC_{i+}), we also define its "inverse" atomic change (AC_{i-}) and the corresponding re-synthesis rule (RS_{i-}). We note that since re-synthesis rule (RS_{i-}) can be understood as the inverse of (RS_{i+}), the detailed explanation is omitted in this paper. For details, see the formal description of all the atomic changes and re-synthesis rules given in Appendix B.

Atomic Change $\boldsymbol{AC_{1+}}$: A PE (say PE_k) becomes a new member of $PEsubst(T)$.

As an example, let us assume that $\mathcal{S}(T) =$ "$\{R \leftarrow 0\}$" is modified to "$\{R \leftarrow 0, \ R' \leftarrow 0\}$" as shown in Fig. 7(b). In this case, PE_3 must execute "$R' \leftarrow 0$"
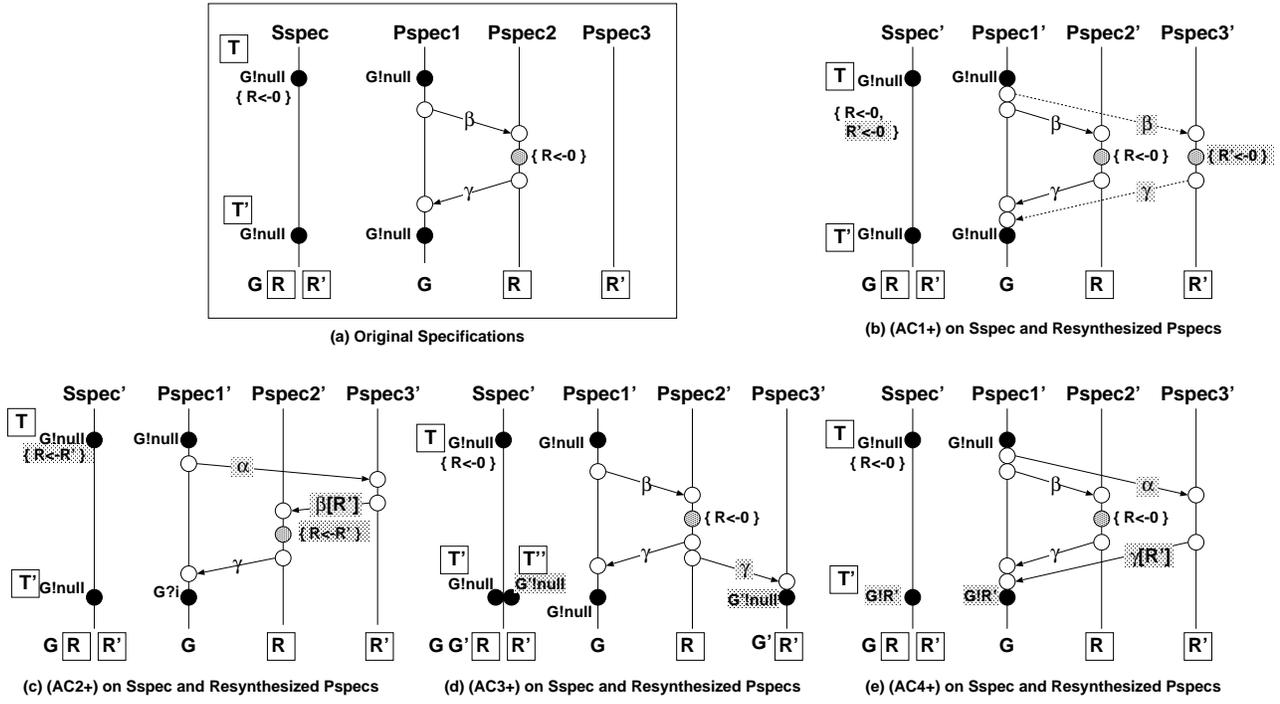
Fig. 7. Atomic changes on $Sspec$ and re-synthesized $Pspec_{1..3}$: timing charts.

(since PE_3 has R'), and thus PE_3 is now a member of $PEsubst(T)$. In general, PE_k must know when to execute $\mathcal{S}(T)$ and then let some PE's know when to start the execution of the next transitions after T. Therefore, in the corresponding re-synthesis rule (RS_{1+}), one additional β-message is sent to PE_k by $PEstart(T)$. Moreover, γ-messages are sent by PE_k to the PE's that start the execution of the next transitions.

In the above example, a new β-message is sent to PE_3 by PE_1, and after the execution of "$R' \leftarrow 0$" a new γ-message is sent to PE_1 by PE_3 .

Atomic Change $\boldsymbol{AC_{2+}}$: An additional register (say R_h) is needed by a PE (say PE_k) in order to execute $\mathcal{S}(T)$.

For example, assume that $\mathcal{S}(T) =$"$\{R \leftarrow 0\}$" is modified to "$\{R \leftarrow R'\}$" as shown in Fig. 7(c). In this case, the value of R' is now needed by PE_2 to execute $\mathcal{S}(T)$. In general, PE_k must receive the value of R_h. Therefore, in the corresponding re-synthesis rule (RS_{2+}), if there exists already a β-message sent to PE_k from a PE which has R_h, then the value of R_h is included in this message. Otherwise, a new β-message including the value of R_h will be sent to PE_k by the PE that has R_h (in this case, some existing β-messages may be removed). Note that adding this new β-message may need a new α-message to be sent from $PEstart(T)$ to the PE that sends the new β-message in order to let this PE know when to send the β-message.

In the example above, a new β-message including the value of R' is sent from PE_3 (and the existing β-message from PE_1 is removed since it has no role

now). Moreover, a new α-message sent from PE_1 to PE_3 is added in order to let PE_3 know when to send the new β-message.

Atomic Change $\boldsymbol{AC_{3+}}$: A PE (say PE_m) becomes a new member of $PEstart(T \bullet \bullet)$, the set of PE's that start the next transitions after T:

For example, let us assume that a new transition T'' with $\mathcal{E}(T'') = $ "$G'!null$" is added as a next transition after T where G' is allocated to PE_3 as shown in Fig. 7(d). In this case, PE_3 is now a new member of $PEstart(T \bullet \bullet)$. In general, the PE's that execute $\mathcal{S}(T)$ must let PE_m know that the execution of $\mathcal{S}(T)$ had been completed. Therefore, in the corresponding re-synthesis rule (RS_{3+}), new γ-messages sent from those PE's to PE_m are added. If $\mathcal{S}(T)$ is empty, $PEstart(T)$ sends a new γ-message to PE_m. In the example above, a new γ-message sent from PE_2 to PE_3 is added in order to let PE_3 know that the execution of $\mathcal{S}(T)$ had been completed.

Atomic Change $\boldsymbol{AC_{4+}}$: An additional register (say R_h) is needed by a PE (say PE_m) to execute $\mathcal{E}(T')$ and/or to check $\mathcal{C}(T')$ where T' is a next transition after T.
For example, assume that $\mathcal{E}(T') =$"$G!null$" is modified to "$G!R'$" as shown in Fig. 7(e). In this case, the value of R' is needed by PE_1 in order to execute $\mathcal{E}(T')$. In general, the value of R_h must be sent to PE_m. Therefore, in the corresponding re-synthesis rule (RS_{4+}), if there exists a γ-message sent to PE_m from the PE_j that has R_h, the value of R_h is included in the message. Otherwise, a new γ-message including the value of R_h is sent from PE_j to PE_m (in this case, some existing γ-messages may be removed). Note that adding this new γ-message may

need an α-message sent from $PEstart(T)$ to PE_j in order to let PE_j know when to send the γ-message, in case that PE_j does not execute $\mathcal{S}(T)$.

In the example above, a new γ-message including the value of R' is sent from PE_3 which has R' (the existing γ-message is left since it has a role to let PE_1 know that the execution of $\mathcal{S}(T)$ had been completed). Moreover, a new α-message is sent from PE_1 to PE_3 in order to let PE_3 know when to send the new γ-message.

According to the above re-synthesis rules, we can determine which messages or actions should be add/removed or modified in the protocol specification. Adding a new message requires two new transitions that send and receive the message, while adding an action requires a single transition to be inserted. They are inserted so that they satisfy the temporal relations between the existing transitions, as specified in the synthesis rules. On the other had, for removing an existing message sent from PE_i to PE_j, the two transitions that send and receive the message are replaced with ε-transitions and then removed. Removing an action can be done in a similar way.

4.2 Decomposing service modifications into atomic changes

In this section, we consider general changes in the service specification and discuss how they can be realized by a sequence of atomic changes, as discussed above. If the sequence of atomic changes is determined, we can then apply the corresponding re-synthesis rules to the protocol specification in the same sequence and thus obtain the modified protocol specification corresponding to the modified service specification.

We consider three types of general changes by which (a) the nature of a transition in a service specification is changed (event, pre-condition or set of assignment statements), (b) a new transition is added or an existing transition is deleted in the service specification, and (c) a register allocation is changed.

(a) changing the nature of a transition: We assume that the event $\mathcal{E}(T)$, pre-condition $\mathcal{C}(T)$ and the assignment statements $\mathcal{S}(T)$ of T are changed. This modification can be realized by the following atomic changes.

- For each transition T' preceding T, $PEstart(T' \bullet \bullet)$ may be changed if $PEstart(T)$ is changed from PE_m to $PE_{m'}$. (PE_m may no longer be a member of $PEstart(T' \bullet \bullet)$ and $PE_{m'}$ may be a new member of $PEstart(T' \bullet \bullet)$). This is represented by atomic changes (AC_{3-}) for the pair of PE_m and T', and (AC_{3+}) for the pair of $PE_{m'}$ and T'.
- In addition, some registers (say R_h) may no longer be needed by PE_m if PE_m is no longer $PEstart(T)$. Instead, some other registers (say $R_{h'}$) may now be needed by $PE_{m'}$ if $PE_{m'}$ is now $PEstart(T)$. This is represented by atomic changes (AC_{4-}) for the pair

of R_h and T', and (AC_{4+}) for the pair of $R_{h'}$ and T', where T' is any transition preceding T.
- Since the set of registers updated in $\mathcal{S}(T)$ may be changed, some PE's (say PE_k) may no longer be members of $PEsubst(T)$, and some other PE's (say $PE_{k'}$) may be new members of $PEsubst(T)$. This is represented by (AC_{1-}) for the pair of PE_k and T and (AC_{1+}) for the pair of $PE_{k'}$ and T.
- In accordance, some registers (say R_h) are no longer needed by some PE_k to execute $\mathcal{S}(T)$. Instead, some other registers (say $R_{h'}$) are now needed by some $PE_{k'}$ to execute $\mathcal{S}(T)$. This is represented by (AC_{2-}) for the pair of PE_k and R_h, and (AC_{2+}) for the pair of $PE_{k'}$ and $R_{h'}$.

(b) a transition is inserted or removed If a new transition T is inserted in $Sspec$, we synthesize its corresponding sub-PNR's according to the synthesis rules described in Section 3.1. Then we insert each sub-PNR of T in the intermediate specification of PE_k ($\overline{Pspec_k}$), and obtain $Pspec_k$ by removing ε-transitions from $\overline{Pspec_k}$. On the other hand, if an existing transition T is removed from $Sspec$, we replace each transition in sub-PNR's of T with ε-transitions and remove them. However, both cases cause some additional modifications which are presented by atomic changes as follows.

- Since the set of next transitions after T' may be changed by the insertion/removal of T, where T' is any transition preceding T, some PE's (say PE_m) may no longer be members of $PEstart(T' \bullet \bullet)$ and some other PE's (say $PE_{m'}$) may become new members of $PEstart(T' \bullet \bullet)$. This is represented by (AC_{3-}) for the pair of PE_m and T', and by (AC_{3+}) for the pair of $PE_{m'}$ and T'.
- In accordance, some registers (say R_h) may no longer be used by some PE's (say PE_m) and some other registers (say $R_{h'}$) may be used in addition by some PE's (say $PE_{m'}$) to start the execution of the next transitions after T'. This is represented by (AC_{4-}) for the PE_m, R_h and T', and by (AC_{4+}) for the $PE_{m'}$, $R_{h'}$ and T'.

Note that the assumptions made on $Sspec$ as presented in Section 3.1 must also hold on the modified $Sspec$, i.e., (a) the Petri net of the modified $Sspec$ must be a live and safe free-choice net, (b) only one PE is involved in the decision to select the next transition in any choice structure, and (c) two parallel transitions of the modified $Sspec$ must not refer/update the value of the same register. For (a), a set of rules is presented in [1, 2] to transform a free-choice net keeping its liveness and safeness properties. We assume that users follow these rules for the insertion/deletion of transitions to/from service specifications. Checking (b) is trivial and (c) can be checked easily using the decompositionality of live and safe free-choice net.

(c) the register allocation is changed Our synthesis method allows copies of a single register to be allocated
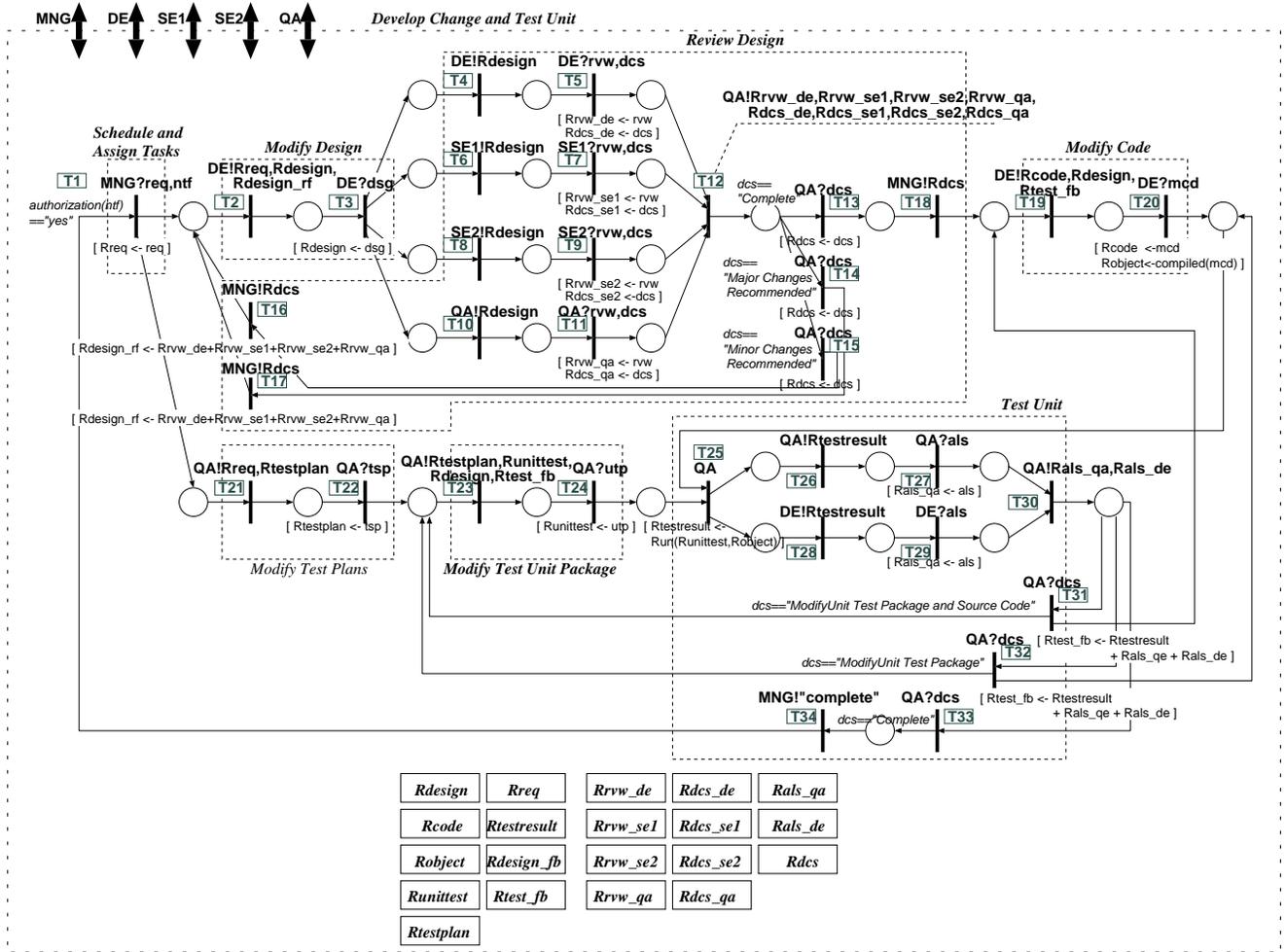
Fig. 8. Modeling the ISPW-6 core problem

to multiple PE's. In our synthesis method, the values of all these copies on different PE's are updated to keep a consistent state. This idea is very useful in some application areas. For example, in distributed databases, adding copies of an existing register to some PE's increases the robustness to faults, balances the load, and may reduce the communication costs.

Now we assume that an existing register R_h, already allocated to some PE(s), is now to be allocated to an additional PE (say PE_k). This causes the following changes.

- For each transition T which has an assignment statement in $\mathcal{S}(T)$ that updates R_h, PE_k becomes a new member of $PEsubst(T)$. This is represented by (AC_{1+}) for the transition T.
- For the same PE_k, some registers (say R_q) are needed to execute $\mathcal{S}(T)$. This is represented by (AC_{2+}) for the pair of R_q and T.
- The value of R_h needs no longer be transferred to PE_k since now PE_k has its own copy of R_h. This is represented by (AC_{2-}) and (AC_{4-}) for the transition T where R_h is used.

On the other hand, if register R_h allocated to more than one PE is removed from one of them, the opposite occurrence of the above case is applied.

We note that sometimes one may want to change the allocation of a register R_h from one PE (say PE_x) to another PE (say PE_y). This modification may be obtained by first allocating R_h to PE_y and then removing R_h from PE_x.

5 Experimental results

5.1 Modeling the ISPW-6 example

Protocol synthesis methods have been applied to many applications such as communication protocols, factory manufacturing systems [16], distributed cooperative work management [15] and so on.

We apply our synthesis and re-synthesis methods to the distributed development of software that involves five engineers (project manager, quality assurance, design, and two software engineers). Each engineer has his/her own machine connected through a network, and participates in the development through a gate (interfaces) of

Table 2. Optimal allocation of resources for engineers' machines

	PE_{mng}	PE_{de}	PE_{se1}	PE_{se2}	PE_{qa}
Gate	MNG	DE	$SE1$	$SE2$	QA
Register	R_{dcs} ($\mathbf{R_{code}}$ is added in case 4)	R_{req} R_{design} R_{design_fb} R_{code} R_{test_fb} R_{object}	($\mathbf{R_{code2}}$ is added in case 1)	($\mathbf{R_{unittest2}}$ is added in case 2)	R_{rvw_de} R_{dcs_de} R_{rvw_se1} R_{dcs_se1} R_{rvw_se2} R_{dcs_se2} R_{rvw_qa} R_{dcs_qa} R_{code} R_{object} $R_{testplan}$ $R_{unittest}$ $R_{testresult}$ R_{alc_qa} R_{als_de}

this machine, using distributed resources placed on this machine. This distributed development process includes tasks for scheduling and assigning tasks, design modification, design review, code modification, test plan modification, modification of unit test packages, unit testing, and progress monitoring. The engineers cooperate with each other to finish these sub-sequential tasks in a suitable order. The reader may refer to *ISPW-6 Core Problem* [33] for a complete description of this process, which was provided as an example to help the understanding and comparison of various approaches to process modeling.

Fig. 8 shows a workflow model of the above development process using *PNR*, where the engineers and resources needed to accomplish the tasks are indicated. We note that for convenience, we do not show the progress monitoring tasks in Fig. 8.

5.2 Experimental results

We regard this workflow as the service specification, and we derived the corresponding protocol specification according to the synthesis method described in Section 3 using the tool described in [29]. The tool lp_solve[34] was used to solve the ILP problem to determine the optimal allocation of resources as described in Section 3.2. The resulting allocation is shown in Table 2. The derived protocol specification includes 29 messages and is not included here due to space limitations.

In this section, we show the effectiveness of our re-synthesis method by comparing the time it takes to derive the complete protocol specification again after each minor modification with the time it takes to re-synthesize a modified protocol specification.

We consider the following three cases, each corresponding to one of the general changes described in Section 4.2.

Case1: QA needs to read the present design to modify the test plans. For this purpose, the value of R_{design} is emitted to QA in transition T_{21}.

Case2: MNG needs to check the test feedback (register R_{test_fb}) before it is shown to DE (that is, between the execution of T_{31} and T_{19}) in order to know how DE and QA proceed the code development process. For this purpose, a new transition T_{35} is added between T_{31} and T_{19} where the value of R_{test_fb} is emitted to MNG.

Case3: For resource accessibility, robustness against storage faults (or failure of machine), and reduction of communication costs, the source code R_{code} is duplicated and a new copy is placed on MNG's machine.

After each case, we have used the program developed in [29] to measure the time (in seconds) to synthesize from scratch a new protocol specification. Moreover, we have also measured the time to re-synthesize a modified protocol specification using a program that we have developed for this purpose. Both programs are written in perl, and the experiments have been performed on a Linux PC (with an Athlon 750 MHz CPU and 256MB memory).

Table 3(a) shows synthesis/re-synthesis times and the number of messages (mes) in the synthesized/re-synthesized protocol specifications for the above three cases. The reader can clearly see that the re-synthesis time is much less than that of a complete synthesis. This is mainly due to the fact that by using the re-synthesis rules, we do not have to re-derive the whole protocol specification. Moreover, we do not re-optimize the number of messages sent between different PE's. Nevertheless, the resulting protocol specifications still have optimal or near-optimal solutions as shown in Table 3(a).

We also consider the following two cases, each of which consists of more than one general change.

Case4: A second version of the source code (register R_{code2}) is placed on the machine of the software engineer 1 (SE1), and the design engineer (DE) modifies and compiles it as well as R_{code}, in "Modify Code" (transitions T_{19} and T_{20}). This modification is treated as two general changes of type (a) on T_{19} and T_{20}. The general change on T_{19} changes the set of registers used to start the execution of the succeeding transitions of T_{18} and T_{31} and

Table 3. Experimental results

(a) each case consists of a single general change on $Sspec$

Case	Changes on $Sspec$	Re-synthesis Rules	Re-synth.		Synth.	
			time	mes.	time	mes.
1	The label of T_{21} is modified (a change of type (a)). $\mathcal{E}'(T_{21}) =$ "$QA!R_{req}, \mathbf{R_{design}}, R_{testplan}$"	(RS_{4+}) for T_1 is applied.	4s	29	520s	29
2	T_{35} is inserted between T_{31} and T_{19} (a change of type (b)). $\mathcal{E}(T_{35}) =$ "$MNG!R_{test\_fb}$"	Sub-PNR's for T_{35} are synthesized. (RS_{4-}) for T_{31}, (RS_{4-}) for T_{31}, (RS_{3-}) for T_{31} and (RS_{3+}) for T_{31} are applied.	24s	31	840s	31
3	R_{code} is also placed on PE_{mng} (a change of type (c)).	(RS_{1+}) and (RS_{2+}) for T_{20} are applied.	14s	31	520s	29

(b) each case consists of multiple (general) changes on $Sspec$

Case	Changes on $Sspec$	Re-synthesis Rules	Re-synth.		Synth.	
			time	mes.	time	mes.
4	The labels of T_{19} and T_{20} are modified (two changes of type (a)). $\mathcal{E}'(T_{19}) =$ "$DE!R_{code}, \mathbf{R_{code2}}, R_{design}, R_{test\_fb}$" $\mathcal{E}'(T_{20}) =$ "$DE?mcd, \mathbf{mcd2}$" $\mathcal{S}'(T_{20}) =$ "$[R_{code} \leftarrow mcd, \mathbf{R_{code2}} \leftarrow \mathbf{mcd2},$ $R_{object} \leftarrow compiled(mcd, \mathbf{mcd2})]$"	(RS_{4+}) for T_{18}, (RS_{4+}) for T_{31}, (RS_{1+}) for T_{20}, (RS_{2+}) for T_{20} and (RS_{2+}) for T_{20} are applied.	29s	35	507s	29
5	The labels of T_{23}, T_{24} and T_{25} are modified (three changes of type (a)). $\mathcal{E}'(T_{23}) =$ "$QA!R_{testplan}, R_{unittest}, \mathbf{R_{unittest2}},$ R_{design}, R_{test_fb}" $\mathcal{E}'(T_{24}) =$ "$QA?utp, \mathbf{utp2}$" $\mathcal{S}'(T_{24}) =$ "$[R_{unittest} \leftarrow utp, \mathbf{R_{unittest2}} \leftarrow \mathbf{utp2}]$" $\mathcal{S}'(T_{25}) =$ "$[R_{testplan} \leftarrow Run(R_{unittest}, \mathbf{R_{unittest2}}, R_{object})]$"	(RS_{4+}) for T_{22}, (RS_{4+}) for T_{31}, (RS_{4+}) for T_{32}, (RS_{1+}) for T_{24}, (RS_{2+}) for T_{24} and (RS_{2+}) for T_{25} are applied.	35s	39	742s	30

the general change on T_{20} changes $PEsubst(T_{20})$ and the set of registers used to execute $\mathcal{S}(T_{20})$ on PE_{de}.

Case5: An additional new unit test (register $R_{unittest2}$) is placed on the machine of the software engineer 2 (SE2), and the QA engineer (QA) modifies it as well as $R_{unittest}$, in "Modify Test Unit Package" (T_{23} and T_{24}). Moreover, an additional test is done using the unit test in "Test Unit" (T_{25}).

Table 3(b) shows the experimental results for the above two cases. The re-synthesis time is still much less than the time for a complete synthesis. Regarding the number of messages, since the present (thus fixed) allocation is used in the re-synthesis method, some deviation from the optimal solution is found for these cases. However, a closer look at the protocol specification obtained by synthesis from the modified service specification indicates that the structure of this protocol specification is quite different from the original protocol specification, which is due to several changes in the register allocation which, in turn, is due to the re-optimization of the resource allocation as described in Section 3.2. We think that this results is a much larger maintenance cost as compared with the re-synthesized protocol specification which has a structure similar to the original one.

Note that, as shown above, there exists a trade-off between the optimality and maintenance cost. Re-optimization of the resource allocation may be applied after several applications of re-synthesis, as the need arises depending on each application's cost criteria [29].

6 Conclusion

We have proposed a synthesis method to derive a protocol specification of a distributed system from a given service specification. The method involves the optimization of register (storage) allocation that minimizes communication costs of the distributed system. We have also proposed a method to re-synthesize the modified protocol specification when some changes of the user requirements have given rise to a modified service specification. The method consists of a set of simple rules that are applied to the original protocol specification. The rules correspond to the changes in the service specification, and are designed to modify only small parts of the protocol specification. Therefore, the resulting modifications on the protocol specification are small compared with the changes that result from the application of the normal protocol synthesis method on the modified service specification. The experimental results have shown that our re-synthesis method could save the maintenance costs, compared with the normal synthesis method.

We are planning to develop an integrated development environment for distributed systems, including tool support for specifying requirements of service

(service specifications) through a graphical interface, synthesizing/re-synthesizing protocol specifications, and Java code generation from the protocol specifications. This is part of our future work.

References

1. T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. of IEEE*, Vol. 77, No. 4, pp. 541–580, 1989.

2. E. Best, "Structure Theory of Petri Nets: the Free Choice Hiatus," *Advances in Petri Nets Part I: Petri Nets, Central Models and Their Properties, in Lecture Notes in Computer Science*, Vol. 254, pp. 168-205, 1986.

3. R. Milner, "Communication and Concurrency," *Prentice-Hall*, 1989.

4. N. A. Lynch, "Distributed Algorithm," *Morgan Kaufmann Publishers*, 1996.

5. V. Carchiolo, A. Faro and D. Giordano, "Formal Description Techniques and Automated Protocol Synthesis," *Journal of Information and Software Technology*, Vol. 34, No. 8, pp. 513–421, 1992.

6. H. Erdogmus and R. Johnston, "On the Specification and Synthesis of Communicating Processes," *IEEE Trans. on Software Engineering*, Vol. SE-16, No. 12, 1990.

7. R. Probert and K. Saleh, "Synthesis of Communication Protocols: Survey and Assessment," *IEEE Trans. on Computers*, Vol. 40, No. 4, pp. 468–476, 1991.

8. K. Saleh, "Synthesis of Communication Protocols: an Annotated Bibliography," *ACM SIGCOMM Computer Communication Review*, Vol. 26, No. 5, pp. 40–59, 1996.

9. R. Gotzhein and G. v. Bochmann, "Deriving Protocol Specifications from Service Specifications Including Parameters," *ACM Trans. on Computer Systems*, Vol. 8, No. 4, pp. 255–283, 1990.

10. R. Langerak, "Decomposition of Functionality; a Correctness-Preserving LOTOS Transformation," *Proc. of 10th IFIP WG6.1 Symp. on Protocol Specification, Testing and Verification (PSTV-10)*, pp. 229–242, 1990.

11. C. Kant, T. Higashino and G. v. Bochmann, "Deriving Protocol Specifications from Service Specifications Written in LOTOS," *Distributed Computing*, Vol. 10, No. 1, pp. 29–47, 1996.

12. P. -Y. M. Chu and M. T. Liu, "Protocol Synthesis in a State-transition Model," *Proc. of COMPSAC '88*, pp. 505–512, 1988.

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

14. M. Nakamura, Y. Kakuda and T. Kikuno, "Component-based Protocol Synthesis from Service Specifications," *Computer Communications Journal*, Vol. 19, No. 14, pp.1200-1215, Dec. 1996.

15. K. Yasumoto, T. Higashino and K. Taniguchi, "Software Process Description Using LOTOS and its Enaction," *Proc. of 16th Int. Conf. on Software Engineering (ICSE-16)*, pp. 169-179, 1994.

16. D. Y. Chao and D. T. Wang, "A Synthesis Technique of General Petri Nets," *Journal of System Integration*, Vol. 4, pp. 67–102, 1994.

17. F.-Y. Wang, K. Gildea, H. Jungnitz and D.D. Chen, "Protocol Design and Performance Analysis for Manufacturing Message Specification: A Petri Net Approach," *IEEE Trans. on Industrial Electronics*, Vol. 41, No. 6, pp. 641–653, 1994.

18. H. Yamaguchi, K. Okano, T. Higashino and K. Taniguchi, "Synthesis of Protocol Entities' Specifications from Service Specifications in a Petri Net Model with Registers," *Proc. of 15th Int. Conf. on Distributed Computing Systems (ICDCS-15)*, pp. 510–517, 1995.

19. H. Kahlouche and J. J. Girardot, "A Stepwise Requirement Based Approach for Synthesizing Protocol Specifications in an Interpreted Petri Net Model," *Proc. of INFOCOM '96*, pp. 1165–1173, 1996.

20. A. Al-Dallal and K. Saleh, "Protocol Synthesis Using the Petri Net Model," *Prof. of 9th Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'97)*, 1997.

21. A. Khoumsi and K. Saleh, "Two Formal Methods for the Synthesis of Discrete Event Systems," *Computer Networks and ISDN Systems*, Vol. 29, No. 7, pp. 759–780, 1997.

22. M. Kapus-Koler, "Deriving Protocol Specifications from Service Specifications with Heterogeneous Timing Requirements," *Proc. of 1991 Int. Conf. on Software Engineering for Real Time Systems*, pp. 266–270, 1991.

23. A. Khoumsi, G. v. Bochmann and R. Dssouli, "On Specifying Services and Synthesizing Protocols for Real-time Applications," *Proc. of 14th IFIP WG6.1 Symp. on Protocol Specification, Testing and Verification (PSTV-14)*, pp. 185–200, 1994.

24. A. Khoumsi and G. v. Bochmann, "Protocol Synthesis Using Basic LOTOS and Global Variables," *Proc. of 1995 Int. Conf. on Network Protocols (ICNP'95)*, 1995.

25. A. Nakata, T. Higashino and K. Taniguchi, "Protocol Synthesis from Timed and Structured Specifications," *Proc. of 1995 Int. Conf. on Network Protocols (ICNP'95)*, pp. 74–81, 1995.

26. H. Yamaguchi, K. Okano, T. Higashino and K. Taniguchi, "Protocol Synthesis from Time Petri Net Based Service Specifications," *Proc. of 1997 Int. Conf. on Parallel and Distributed Systems (ICPADS'97)*, pp. 236–243, 1997.

27. J. -C. Park and R. E. Miller, "Synthesizing Protocol Specifications from Service Specifications in Timed Extended Finite State Machines," *Proc. of 17th Int. Conf. on Distributed Computing Systems (ICDCS-17)*, 1997.

28. K. El-Fakih, H. Yamaguchi and G.v. Bochmann, "A Method and a Genetic Algorithm for Deriving Protocols for Distributed Applications with Minimum Communication Cost," *Proc. of 11th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'99)*, pp. 863–868, 1999.

29. H. Yamaguchi, K. El-Fakih, G.v. Bochmann and T. Higashino, "A Petri Net Based Method for Deriving Distributed Specification with Optimal Allocation of Resources," *Proc. of ASIC Int. Conf. on Software Engineering Applied to Networking and Parallel/ Distributed Computing (SNPD'00)*, pp. 19–26, 2000.

30. K. El-Fakih, H. Yamaguchi, G.v. Bochmann and T. Higashino, "Protocol Re-synthesis Based on Extended Petri Nets," *Proc. of Int. Workshop on Software Engineering and Petri Nets (SEPN-2000)*, pp. 173–188, 2000.

31. K. El-Fakih, H. Yamaguchi, G.v. Bochmann and T. Higashino, "Automatic Derivation of Petri Net Based Dis-

tributed Specification with Optimal Allocation of Resources," *Proc. of 15th IEEE Int. Conf. on Automated Software Engineering (ASE'2000)*, pp. 305–308, 2000.

32. B. B. Bista, K. Takahashi, H. Kaminaga and N. Shiratori, "A Flexible Protocol Synthesis Method for Adopting Requirement Changes," *Proc. of the 1996 Int. Conf. on Parallel and Distributed Systems (ICPADS'96)*, pp. 319-326, 1996.

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

34. "lp_solve," `ftp://ftp.ics.ele.tue.nl/pub/lp solve/`

35. S. Schach, "Software Engineering," *Boston:Aksen Assoc*, 1992

Hirozumi Yamaguchi received his B.E., M.E. and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan. Since 1999, he has been a research associate at Osaka University. His research interests are design and implementation of distributed systems and communication protocols, and resource management in multicast applications.

Khaled El-Fakih received his B.S. and M.S. degrees in Computer Science from the Lebanese American University. Currently, he is a Ph.D. student at the University of Ottawa and an instructor at the American University of Sharjah. His current research interests are in automating the design of distributed systems, test development from given specifications, and fault diagnosis of distributed systems.

Gregor von Bochmann is professor at the School of Information Technology and Engineering at the University of Ottawa since January 1998. Previously he was professor at the University of Montreal for 25 years. He is a fellow of the IEEE and ACM and a member of the Royal Society of Canada. He has worked in the area of programming languages, compiler design, communication protocols, and software engineering and has published many papers in these areas. He has also been actively involved in the standardization of formal description techniques for communication protocols and services. His present work is aimed at methodologies for the design, implementation and testing of communication protocols and distributed systems. Ongoing projects include distributed systems management and quality of service negotiation for distributed multimedia applications.

Teruo Higashino received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Osaka, Japan, in 1979, 1981 and 1984, respectively. He joined the faculty of Osaka University in 1984. Since 1999 he has been a Professor in the Department of Informatics and Mathematical Science at Osaka University. In 1990 and 1994, he was a Visiting Researcher of Dept. I.R.O. at University of Montreal, Canada. His current research interests include design and analysis of distributed systems, specification and verification of communication protocols, and formal approach of program design. He is a senior member of IEEE and a member of IFIP TC6/WG 6.1.

Appendix A: Synthesis rules

[action rules]

$(\mathbf{S}_{A1})$ PE_i that has the gate G used in $\mathcal{E}(T)$ checks that
  (1) the value of $\mathcal{C}(T)$ is true,
  (2) the execution of the previous transitions of T is completed
  (3) an input has been given through G, if $\mathcal{E}(T)$ is an input event.
  Then PE_i executes $\mathcal{E}(T)$. This PE_i is denoted $PEstart(T)$.

$(\mathbf{S}_{A2})$ After $(\mathbf{S}_{A1})$, each PE (say PE_k) executes the subset of the assignment statements of $\mathcal{S}(T)$ that update the registers allocated to PE_k. The set of these PE's is denoted by $PEsubst(T)$. These assignment statements are executed when the corresponding β-messages are received (see below).

[message rules]

$(\mathbf{S}_{M\alpha})$ After $(\mathbf{S}_{A1})$, $PEstart(T)$ only sends α-messages. The PE's to which α-messages are sent are determined in $(\mathbf{S}_{M\beta3})$ and $(\mathbf{S}_{M\gamma3})$.

$(\mathbf{S}_{M\beta1})$ Each $PE_k \in PEsubst(T)$ must receive at least one β-message from some PE's (each called PE_j) in order to know the timing to execute $\mathcal{S}(T)$. This message also lets PE_k know the values of registers used in $\mathcal{S}(T)$ (see $(\mathbf{S}_{M\beta2})$).

$(\mathbf{S}_{M\beta2})$ For each register R_h that is used to execute $\mathcal{S}(T)$ by PE_k, PE_k must receive its value through a β-message if R_h is not allocated to PE_k.

$(\mathbf{S}_{M\beta3})$ Each PE_j that sends a β-message to $PE_k \in PEsubst(T)$ knows the timing to send the message by receiving an α-message from $PEstart(T)$ unless PE_j is $PEstart(T)$.

$(\mathbf{S}_{M\gamma1})$ Each $PE_m \in PEstart(T\bullet\bullet)$, where $T\bullet\bullet$ is the set of the next transitions after T, must receive a γ-message from each $PE_k \in PEsubst(T)$ after $(\mathbf{S}_{A2})$. This lets PE_m know that the execution of $\mathcal{S}(T)$ had been completed on PE_k. If $PEsubst(T)$ is empty, PE_m must receive at least one γ-message from any PE in order to know that the execution of T had been completed. γ-messages also let PE_m know the values of registers used in the pre-conditions and/or events of next transitions (see $(\mathbf{S}_{M\gamma3})$).

$(\mathbf{S}_{M\gamma2})$ For each register R_h used by PE_m to start the execution of the next transitions of T, PE_m must receive its value through a γ-message if R_h is not allocated to PE_m.

$(\mathbf{S}_{M\gamma3})$ Each PE_l that sends a γ-message to $PE_m \in PEstart(T\bullet\bullet)$ must be in $PEsubst(T)$ (see $(\mathbf{S}_{M\gamma1})$), must receive an α-message from $PEstart(T)$ or must be $PEstart(T)$, in order to know the timing to send the γ-message to PE_m.

Appendix B: Re-synthesis rules

$PEsubst(T)$ has been changed:

(AC_{1+}) $PEsubst(T) = PEsubst(T) \cup \{PE_k\}$

(RS_{1+}) Add a β-message sent from $PE_i = PEstart(T)$ to PE_k in order to let PE_k know the timing to execute $\mathcal{S}(T)$. Also add γ-messages sent from PE_k to

$\forall PE_m \in PEstart(T \bullet \bullet)$ in order to let PE_m know that the execution of $\mathcal{S}(T)$ on PE_k had been completed.

(AC_{1-}) $PEsubst(T) = PEsubst(T) \setminus \{PE_k\}$

(RS_{1-}) Delete the β-messages sent to PE_k since PE_k no longer executes $\mathcal{S}(T)$. Also delete the γ-messages which include no value sent from PE_k. Finally, if at least one γ-message from PE_k still exists, add an α-message from PE_i to PE_k.

$Rsubst_k(T)$ has been changed: ($Rsubst_k(T)$ is the set of registers used to execute $\mathcal{S}(T)$ on PE_k)

(AC_{2+}) $Rsubst_k(T) = Rsubst_k(T) \cup \{R_h\}$

(RS_{2+}) Include the value of R_h in one of the existing β-messages sent from PE_j which has R_h to PE_k, since PE_k needs the value of R_h for the execution of $\mathcal{S}(T)$. If such a message does not exist, add a new β-message including the value of R_h, sent from PE_j which has R_h to PE_k. Also add an α-message from PE_i to PE_j ($PE_i = PEstart(T)$, $i \neq j$) if PE_j does not receive an α-message.

(AC_{2-}) $Rsubst_k(T) = Rsubst_k(T) \setminus \{R_h\}$

(RS_{2-}) Exclude the value of R_h from the existing β-messages sent to PE_k, since PE_k no longer needs the value of R_h for the execution of $\mathcal{S}(T)$. Then delete each of the β-messages only if (a) it has no register value and (b) another β-message sent to PE_k exists. Finally, delete the α-message sent to PE_j ($PE_i = PEstart(T)$) only if there is no β-message nor γ-message sent from PE_j.

$PEstart(T \bullet \bullet)$ has been changed:

(AC_{3+}) $PEstart(T \bullet \bullet) = PEstart(T \bullet \bullet) \cup \{PE_m\}$

(RS_{3+}) Add γ-messages from PE_k to $\forall PE_k \in PEsubst(T)$, since PE_m needs to know that the execution of $\mathcal{S}(T)$ had been completed. If $PEsubst(T)$ is empty, add a γ-message sent from PE_i to PE_m where $PE_i = PEstart(T)$.

(AC_{3-}) $PEstart(T \bullet \bullet) = PEstart(T \bullet \bullet) \setminus \{PE_m\}$

(RS_{3-}) Delete the existing γ-messages sent to PE_m, since PE_m no longer needs to know that the execution of $\mathcal{S}(T)$ had been completed.

$Rstart_m(T \bullet \bullet)$ has been changed: ($Rstart_m(T \bullet \bullet)$ is the set of registers used by PE_m to start the execution of next transitions)

(AC_{4+}) $Rstart_m(T \bullet \bullet) = Rstart_m(T \bullet \bullet) \cup \{R_h\}$

(RS_{4+}) Include the value of R_h in one of the existing γ-messages sent from PE_l which has R_h to PE_m, since PE_m needs the value of R_h for the execution of next transitions of T. If such a message does not exist, add a new γ-message including the value of R_h sent from PE_l which has R_h to PE_m. Also add an α-message sent from $PE_i = PEstart(T)$ to PE_l if PE_l does not receive an α-message and $PE_l \notin PEsubst(T_x)$.

(AC_{4-}) $Rstart_m(T \bullet \bullet) = Rstart_m(T \bullet \bullet) \setminus \{R_h\}$

(RS_{4-}) Exclude the value of R_h from γ-messages sent to PE_m since PE_m no longer needs the value of R_h for the execution of the next transitions of T. Then delete each of the γ-messages sent to PE_m only if (a) it is sent from $PE_l \notin PEsubst(T)$, (b) a γ-message sent to PE_m exists and (c) the γ-message has no register value. Finally, delete the α-message sent to

PE_l only if there is no β-message nor γ-message sent from PE_l.