Protocol Re-synthesis with Optimal Allocation of Resources
   Based on Extended Petri Nets

**Hirozumi Yamaguchi**†
Graduate School of Engineering Science
Osaka University
1-3 Machikaneyamacho, Toyonaka, Osaka 560-8531, JAPAN

**Khaled El-Fakih**
School of Information Technology and Engineering
University of Ottawa
150 Louis Pasteur, Ottawa, Ontario, K1N 6N5, Canada

**Gregor v. Bochmann**
School of Information Technology and Engineering
University of Ottawa
Colonnel By Hall (A617), P.O.Box 450 Stn A, Ottawa, Ontario, K1N 6N5, Canada

**Teruo Higashino**
Graduate School of Engineering Science
Osaka University
1-3 Machikaneyamacho, Toyonaka, Osaka 560-8531, JAPAN

† Contact Person
E-mail: h-yamagu@ics.es.osaka-u.ac.jp
FAX: +81-6-6850-6609
Abstract

Protocol synthesis is used to derive a specification of a distributed system called a protocol specification (a set of programs of cooperative computers) from a specification of services (called a service specification) to be provided by the distributed system to its users. It reduces design costs and errors in specifying communications between computers in the protocol specification. In general, maintaining such a distributed system involves applying frequent minor modifications to the service specification due to changes in the user requirements. Deriving a 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 for the modification to the service specification. In order to reduce the maintenance cost of such a system, we present a new synthesis method to re-synthesize only the corresponding part of the current protocol specification after modifications to 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.

Keywords


1 Introduction

Synthesis methods have been used (for surveys see [6, 7]) 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 cooperative computers (called protocol entities (PE's)) contains 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 by using manual methods.

A number of protocol synthesis strategies have been described in the literature. The first strategy aims at implementing complex control-flows using several computational models such as CCS based models[4, 5],
LOTO$\text{S}$[9, 10], Petri nets[15, 18] and FSM/EFSM[11, 13]. The second strategy, [21, 19, 20, 23, 24, 25], 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, [8, 12, 16, 17, 22, 26, 27, 29], 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[27], 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 consider a software development process in Computer Supported Cooperative Work (CSCW). This process is carried out cooperatively by multiple engineers (developers, designers, managers and others). Each engineer has his/her own machine (PE) and participates in the development process using the specified distributed resources (drafts, source codes, object codes, multimedia video and audio files, and others) which may be placed on different machines. 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 decide 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 users’ requirements. Moreover, developers usually implement incrementally the given specification. Synthesizing the completely whole system after each minor modification (or increment) is considered expensive and time consuming, especially for operational large size distributed systems with large number of users distributed over multiple sites. Therefore, in order to reduce the maintenance cost of such a system, which was reported to account for as much as two-thirds of the cost of the software production[33], it is important to modify only a part of the existing protocol specification depending on the modification of
the service specification (protocol re-synthesis). This would also identify the changes that have to be made to the protocol specification. Unfortunately, none of existing methods considers re-synthesis of protocol specifications according to the changes in the service specifications.

In this paper, we propose a new protocol synthesis method for re-synthesizing an existing protocol specification from a modification of the corresponding service specification. The method consists of a set of rules that would be applied to only parts of the protocol specification to be modified. The protocol specification corresponding to the unmodified parts of the service specification is preserved intact. A computer supported cooperative software development process is used as an example to show that the method reduces the cost of synthesizing the whole software development process after each simple modification.

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 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 \textit{Petri Net with Registers (PNR in short)} \cite{16} to describe both service specifications and protocol specifications of distributed systems. In this model, the service access points between the users and 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 \( PNR \) has a label \( \{ \mathcal{C}(t), \mathcal{E}(t), \mathcal{S}(t) \} \), where \( \mathcal{C}(t) \) is a pre-condition (the firing condition of \( t \)), \( \mathcal{E}(t) \) is an I/O event and \( \mathcal{S}(t) \) is a set of substitution statements (which represents parallel update of the registers’ values).

A transition \( t \) may fire if (a) each of its input places has one 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.

Consider, for example, transition \( t \) of Fig. 1 where \( \mathcal{C}(t) = \{i > R_1\} \), \( \mathcal{E}(t) = \{G_1 \leftarrow i\} \) and \( \mathcal{S}(t) = \{R_1 \leftarrow R_2 + i, R_2 \leftarrow R_1 + R_2 + i\} \). Here, \( i \) denotes an input variable which holds an input value. The input value can be referred only in its transition \( t \). \( R_1 \) and \( R_2 \) denote registers, which hold the assigned values until new values are assigned, and their values may be referred and updated by all the transitions in \( PNR \). This means that registers are treated as global variables. \( G_1 \) is a gate, that is, a service access point (interaction point) between the users and system. Note that “?” and “!” in \( \mathcal{E}(t) \) mean that \( \mathcal{E}(t) \) is an input event and an output event, respectively.

Assume that an integer of value 3 has been given through gate \( G_1 \), and the current values of registers \( R_1 \) and \( R_2 \) are 1 and 2, respectively. In this case, since the value of pre-condition “\( i > R_1 \)” is true, the transition may fire. If it fires, event “\( G_1 \leftarrow i \)” is executed and the input value 3 is assigned to input variable \( i \). Then “\( R_1 \leftarrow R_2 + i \)” and “\( R_2 \leftarrow R_1 + R_2 + i \)” are executed in parallel. After the firing, the tokens are moved, and the values of registers \( R_1 \) and \( R_2 \) are changed to five (\( = 2 + 3 \)) and six (\( = 1 + 2 + 3 \)), respectively (Fig. 1(b)).

Formally, \( \mathcal{E}(t) \) is one of the following three events: “\( G_s \leftarrow exp \)”, “\( G_s \leftarrow i \)” or “\( \tau \)”. Here, “\( G_s \leftarrow exp \)” is an
output event and it means that the value of expression \( \text{"exp"} \) is output through gate \( G_s \) where all the
arguments in \( \text{"exp"} \) are registers. \( G_s \ ? \text{"i"} \) is an input event and it means that the value given through \( G_s \)
is assigned to the input variable \( \text{"i"} \). The event \( \tau \) is an internal event, which is executed in the system,
and it is unobservable from the users.

\( S(t) \) is a set of substitution statements, each of which has the form \( R_w \leftarrow \text{exp}_w \) where \( R_w \) is a
register and \( \text{exp}_w \) is an expression whose arguments are from the input variable in \( E(t) \) and/or registers.

2.2 Service Specifications

In the abstract level, a distributed system is regarded as a centralized system which works and provides
services as a single “virtual” machine. The number of actual PE’s and communication channels between
them are hidden. A specification of distributed systems at this level is called a service specification and denoted by $S_{spec}$ in this paper. Actual resources of a distributed system may be located on some physical machines, called protocol entities. However, only one virtual machine is assumed at this level. For better readability and understanding, we use a simple (and somewhat trivial) example of $S_{spec}$ in Fig. 2(a) hereafter. A practical example is given in Section 5.

The example describes a system to find the maximum value from a given sequence of values, using two gates $G_{in}$ and $G_{out}$ and three registers $R_{cut}$, $R_{max}$ and $R_{cy}$. It receives a value (input variable $i$) through gate $G_{in}$, and checks the values of pre-conditions $C(T_1)$ and $C(T_2)$. If the value of input is not greater than the maximum value in the precedence value sequence (we assume that the maximum value is stored in $R_{max}$), $T_1$ fires and the system just increments the counter (register $R_{cut}$). Otherwise $T_2$ fires and the input is stored to $R_{max}$ as the new maximum value and the counter is also incremented. Finally, the system outputs the pair of the maximum value and the value of counter through $G_{out}$ on $T_3$, and returns to the initial state. Note that on $T_3$, the counter $R_{cut}$ is reset to zero if it reaches the value of $R_{cy}$ (the upper limit of the counter).

2.3 Protocol Specifications

A distributed system can be considered as a communication system which consists of $p$ protocol entities $PE_1$, $PE_2$, ..., and $PE_p$. We assume a duplex and reliable communication channel between any pair of $PE$'s ($PE_i$ and $PE_j$) with infinite capacity buffers at the both $PE$'s. 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 resources (registers and gates) 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 as $M_{ij}[data]$ that consists of a message ID part (header) $M_{ij}$ and a data part $[data]$ (payload) where “data” is a set of “label:value”. The ID $M_{ij}$ means that this message is sent from $PE_i$ to $PE_j$ and is associated with transition $T_x$ of $S_{spec}$. “M” is one of the following three message types ($\alpha$, $\beta$ or $\gamma$) explained later. We assume that if $PE_i$ executes an output event $g_{ij}!M_{ij}[data]$ on a transition, the
message $M^e_{ij}[data]$ is sent to $PE_j$ through gate $g_{ij}$ via communication channel and written into the buffer at $PE_j$’s end. If $PE_j$ executes an input event \( g_{ij}?w \) with pre-condition \( ID(M^e_{ij}, w) \) on a transition, $PE_j$ removes a message \( M^e_{ij}[data] \) from the buffer and the message is kept in input variable \( w \). Note that after taking the message from the buffer, each value in the data part can be referred using its label as suffix such as \( w.R_z \). For example, if the data part contains “$R_z:4$” in the message, \( w.R_z = 4 \).

In order to implement a distributed system which consists of $p$ 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 $p$ protocol entity specifications \( \{ Pspec_1, ..., Pspec_p \} \) is called a protocol specification and denoted by $Pspec^{(1:p)}$. We need a protocol specification in order to implement a given service specification.

As an example, let us assume that there are only 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 gate $G_{in}$ and register $R_{out}$, and $PE_2$ has gate $G_{out}$ and registers $R_{reg}$ and $R_{max}$. Note that in addition to these registers, we assume that each $PE_i$ has an additional register $R_{tmp}$ to hold the values of input variables and registers in the received messages and input variables on $PE_i$ itself. $R_{tmp}$ may contain multiple values, and they are distinguished by suffix such as $R_{tmp_i}, R_{w}$ and $R_{tmp_i,z}$. 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 in Fig 2(b), internal events “$\tau$”, pre-conditions “true” and empty sets of substitution statement are omitted.

According to the protocol specification of Fig. 2(b), $PE_1$ first receives an input through $G_{in}$ and calculates the values of $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$ in order to check the executability of $t_{1a}$ and $t_{2a}$. If the value of $\mathcal{C}(t_{1a})$ is true, $PE_1$ executes $t_{1a}$ and keeps the input value in input variable $i$. Then it increments the value of its own register $R_{out}$ on transition $t_{1b}$. After that, $PE_1$ returns to the initial marking and waits for an input. On the other hand, if the value of $\mathcal{C}(t_{2a})$ is true, $PE_1$ executes $t_{2a}$, keeps the value of input in input variable $i$ and stores it to $R_{tmp_i}$. Then it sends a message \( \gamma^2_{12}[i : R_{tmp_1,i}] \) on transition $t_{2b}$ to send the input value to $PE_2$, since $PE_2$ needs the value of $i$ to change the value of $R_{max}$. In parallel with $t_{2b}$, $PE_1$ changes the value of $R_{out}$ to $R_{out} + 1$ on transition $t_{2c}$ and sends a message \( \gamma^2_{12}[R_{out} : R_{out}] \) to $PE_2$. 

8
on transition \( t_{2d} \) in order to let \( PE_2 \) know that \( PE_1 \) finished the execution of “\( R_{cut} \leftarrow R_{cut} + 1 \)” with the new value of \( R_{cut} \). \( PE_2 \) receives these two independent messages. If \( PE_2 \) receives “\( \beta_2^{R_{tmp}}[i : R_{tmp1}.i] \)”, it stores the received value of \( i \) to \( R_{tmp2}.i \) on transition \( t_{2g} \) and uses the value to change the value of register \( R_{max} \) on the next transition \( t_{2h} \). If \( PE_2 \) receives “\( \gamma_2^{R_{cut}}[R_{cut} : R_{cut}] \)”, it stores the received value of \( R_{cut} \) to \( R_{tmp2}.R_{cut} \) on transition \( t_{2i} \). After the firing of \( t_{2k} \) and \( t_{2l} \), \( PE_2 \) knows that the substitution statements of \( T_2 \) of \( Spec \) had been executed and starts the execution of \( t_{3e} \). \( PE_3 \) outputs the values of \( R_{max} \) and \( R_{tmp2}.R_{cut} \) on transition \( t_{3e} \) and sends two messages “\( \beta_2^{R_{cut}}[R_{cut} : R_{cut}] \)” and “\( \gamma_2^{R_{max}}[R_{max} : R_{max}] \)” to \( PE_1 \) on transitions \( t_{3f} \) and \( t_{3g} \), respectively. The former message is sent to inform the value of register \( R_{cut} \) necessary for changing the value of register \( R_{cut} \), and the latter one is sent to inform the value of register \( R_{max} \) necessary for evaluating the pre-conditions of \( t_{1a} \) and \( t_{2a} \) (two message types “\( \beta \)” and “\( \gamma \)” are arranged for these different purposes in our synthesis method). If \( PE_1 \) receives the \( \beta \) message, it stores the received value of \( R_{cut} \) to \( R_{tmp1}.R_{cut} \) on transition \( t_{3b} \) and changes the value of register \( R_{cut} \) using it on transition \( t_{3c} \). If \( PE_1 \) receives the \( \gamma \) message, it stores the received value of \( R_{max} \) to \( R_{tmp1}.R_{max} \) on transition \( t_{3d} \) so that \( PE_1 \) can use the value to evaluate the pre-conditions of \( t_{1a} \) and \( t_{2a} \). Finally, these two \( PE \)'s (\( PE_1 \) and \( PE_2 \)) return to their initial markings.

As shown in the above discussion, \( PE \)'s cooperate with each other by exchanging messages. The communication between different \( PE \)'s may be quite complex, and it is difficult to design correct protocol specifications. Therefore we would like to derive a protocol specification automatically, which provides the same services as a given service specification.

3 Synthesis Overview

A method for deriving a protocol specification from a given service specification is presented in this section. This method is based on a set of synthesis rules that specify how to execute each transition \( T_x = \langle C(T_x), E(T_x), S(T_x) \rangle \) of the service specification by the corresponding \( PE \)'s in the protocol specification. Based on these rules, the behavior of each \( PE \) and an optimal allocation of resources (registers and gates) for minimum communication costs is determined. Then the specifications of \( PE \)'s (protocol entity
specifications) written in our PNR model are derived.

3.1 Synthesis Rules

For executing a transition $T_x = (C(T_x), E(T_x), S(T_x))$ of the service specification by the set of transitions of PEs in the protocol specification, we use the following algorithm.

- The PE that has gate $G_s$ used in $E(T_x)$ (this PE is denoted as $P(E_{\text{start}}(T_x))$) decides to start the execution of $T_x$ by checking the value of pre-condition $C(T_x)$ and then executes the event $E(T_x)$.

  For example, for transition $T_2$ of the service specification in Fig. 2(a), $P(E_{\text{start}}(T_2))$ is $PE_1$ since $PE_1$ has gate $G_{in}$. $PE_1$ checks the value of $C(T_2) = \text{"i > R_{\text{max}}"}$ and if it is true, $PE_1$ executes $E(T_2) = \text{"G_{in}?i"}$ on transition $t_{2a}$.

- Then, $P(E_{\text{start}}(T_x))$ sends messages called $\alpha$-messages to those PEs that have the registers used to execute the substitution statements in $S(T_x)$. $\alpha$-messages let those PEs send these values to certain PEs. Hereafter, an $\alpha$-message sent from $PE_i$ to $PE_j$ in the execution of $T_x$ is denoted as $\alpha_{ij}^x$.

  In the example above, the values of $R_{\text{cut}}$ and $i$ are used to execute $S(T_2) = \text{"\{R_{\text{max}} \leftarrow i, R_{\text{cut}} \leftarrow R_{\text{cut}} + 1\"\}"}$. Since $PE_1$ has both of these values and $P(E_{\text{start}}(T_2)) = PE_1$, only $\alpha_{11}^2$ is needed. However, any message to itself “$M_{ii}^x$” can be omitted and consequently no $\alpha$-message is sent in this case.

- In response, these PEs that have received $\alpha$-messages send the values of registers to those PEs that have the registers to be updated in $S(T_x)$ as messages called $\beta$-messages. $P(E_{\text{subst}}(T_x))$ denotes the set of those PEs. Hereafter, a $\beta$-message sent from $PE_j$ to $PE_k$ in the execution of $T_x$ is denoted as $\beta_{jk}^x$. Using those register values, each PE in $P(E_{\text{subst}}(T_x))$ changes the values of their own registers according to $S(T_x)$. $S_k(T_x)$ denotes the subset of $S(T_x)$ that $PE_k$ executes.

  Using the same example, $P(E_{\text{subst}}(T_x)) = \{PE_1, PE_2\}$ ($PE_1$ has register $R_{\text{cut}}$ and $PE_2$ has $R_{\text{max}}$). Since $PE_1$ receives $\alpha_{11}^2$, which is actually omitted in the specification, $PE_1$ sends a $\beta$-message $\beta_{12}^x$.
to PE₂ on transition $t_{2b}$ with the value of $i$ used by PE₂ to execute $S_2(T_2) = \{ R_{max} \leftarrow i \}$. PE₂ receives it on transition $t_{2g}$. Both PE₁ and PE₂ execute their own subsets of $S(T_2)$ on transitions $t_{2c}$ and $t_{2h}$, respectively.

- After all $S_i(T_k)$ are executed, notification messages called $\gamma$-messages are sent to those PE’s which will start the execution of the next transitions. The set of those PE’s is denoted as $PE_{start}(T_x \bullet \bullet)$. Hereafter, a $\gamma$-message sent from $PE_i$ to $PE_m$ in the execution of $T_x$ is denoted as $\gamma_{im}^\gamma$.

$PE_1$ sends $\gamma$-message $\gamma_{12}^\gamma$ on transition $t_{2d}$ to $PE_{start}(T_3) = PE_2$. $PE_2$ knows that the execution of $S(T_x)$ had been finished on PE’s in $PE_{subst}(T_x)$ and starts the execution of $t_{3c}$. Note that $PE_2$ does not know the new value of $R_{cnt}$ used in $E(T_3)$. Therefore $PE_1$ includes the value of $R_{cnt}$ in $\gamma_{12}^\gamma$.

The above algorithm is presented as a set of rules in Fig. 3, called synthesis rules[27]. The synthesis rules are classified into the action rules and message rules. The notation used by these rules is summarized in Table 1. The action rules specify which PE’s should check the pre-condition and execute the event and substitution statements of $T_x$. The message rules specify which PE’s should exchange messages. The contents and types of these messages are also specified.
### Action Rules

(SA1) PEi that has the gate Gs used in \( \mathcal{E}(T_x) \) checks that

1. the value of \( \mathcal{E}(T_x) \) is true,
2. the execution of the previous transitions of \( T_x \) (\( \bullet \bullet \) \( T_x \) denotes the set of those transitions) has been finished and
3. an input has been given through \( G_s \) if \( \mathcal{E}(T_x) \) is an input event.

Then PEi executes \( \mathcal{E}(T_x) \). This PEi is denoted by \( PEstart(T_x) \).

(SA2) After (SA1), each PE (say PEk) executes \( S_k(T_x) \), a subset of substitution statements of \( S(T_x) \) that updates only the registers allocated to PEk. The set of PEk where \( S_k(T_x) \) is not empty is denoted by \( PESubst(T_x) \).

### Message Rules

(Mo) After (SA1), the only PE that can send \( \alpha \)-message is \( PEstart(T_x) \).

(M1) Each \( PE_k \in PESubst(T_x) \) must receive at least one \( \beta \)-message from some PE's (each called \( PE_j \) in order to know the timing to execute \( S_k(T_x) \). This message also lets \( PE_k \) know the values of registers used in \( S_k(T_x) \) (see (M2)).

(M2) For each register \( R_k \in RSubst_k(T_x) \), a register used by \( PE_k \) to execute \( S_k(T_x) \), \( PE_k \) must receive its value through a \( \beta \)-message if \( R_k \) is not allocated to \( PE_k \).

(M3) Each \( PE_j \) that sends a \( \beta \)-message to \( PE_k \in PESubst(T_x) \) knows the timing to send the message by receiving an \( \alpha \)-message from \( PEstart(T_x) \).

(M4) Each \( PE_m \in PEstart(T_x \bullet \bullet ) \), where \( T_x \bullet \bullet \) is the set of the next transitions of \( T_x \), must receive a \( \gamma \)-message from each \( PE_k \in PESubst(T_x) \) after (SA2). This let \( PE_m \) know that the execution of \( S_k(T_x) \) had been finished.

(M5) If \( PESubst(T_x) \) is empty, each \( PE_m \in PEstart(T_x \bullet \bullet ) \) must receive at least one \( \gamma \)-message from some PEi in order to know that the execution of \( T_x \) had been finished. \( \gamma \)-messages also lets \( PE_m \) know the values of registers used in the pre-conditions and/or events of next transitions (see (M6)).

(M6) Each \( PE_k \) that sends a \( \gamma \)-message to \( PE_m \in PEstart(T_x \bullet \bullet ) \) must be in \( PESubst(T_x) \) (see (M4)) or must receive an \( \alpha \)-message from \( PEstart(T_x) \) in order to know the timing to send the \( \gamma \)-message to \( PE_m \).

(M7) For each \( R_k \in Rstart_m(T_x \bullet \bullet ) \), a register used by \( PE_m \) to start the execution of the next transitions of \( T_x \), \( PE_m \) must receive its value through a \( \gamma \)-message if \( R_k \) is not allocated to \( PE_m \).

Figure 3: Synthesis Rules for Transition \( T_x \)
Consequently, three types of messages are exchanged for the execution of \( T_x \). (1) \( \alpha \)-messages are sent by the PE that starts the execution of \( T_x \) \( (i.e. \ P E\text{start}(T_x)) \) to let those PE’s send their registers’ values to other PE’s. An \( \alpha \)-message does not contain values of registers. (2) \( \beta \)-messages are sent in order to let each PE in \( P E\text{subst}(T_x) \) know the values of registers needed to execute the subset of substitution statements of \( T_x \). (3) \( \gamma \)-messages are sent in order to let each \( P E_m \in P E\text{start}(T_x \cdot \cdot) \) know that the execution of \( T_x \) had been finished. It also informs \( P E_m \) of the values of registers used to start the execution of the next transitions.

The above synthesis rules assume that a resource allocation of gates and registers to PE’s is given. In Section 3.2, we extend our synthesis method so that an optimal resource allocation that minimizes the communication costs among these PE’s can be determined.

Our synthesis method assumes that the Petri net of the service specification is a live and safe \textit{free-choice net}[1]. 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] and this property is used in our algorithm. Also, it assumes that for two transitions of \( S\text{spec} T_x \) and \( T_y \) in a choice structure, \( P E\text{start}(T_x) = P E\text{start}(T_y) \) \( (i.e. \ the \ gates \ in \ E(T_x) \ and \ E(T_y) \ are \ allocated \ to \ the \ same \ PE) \). This guarantees that only one PE makes the decision to select one of two (or more) transitions in a choice structure, otherwise an agreement is needed among the PE’s to make this decision. This can be done by implementing a leader election algorithm as the one shown in [3]. Finally, it is assumed that for two transitions \( T_u \) and \( T_v \) of \( S\text{spec} \) that may be executed in parallel, they do not update/refer the same register. 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 as the one shown in [3].

### 3.2 Optimal Resource Allocation for Minimum Communication Costs

Based on the synthesis rules of Section 3.1, we have proposed three different models that would deal with the allocation of resources.
Our first model \cite{16, 26} assumes that the allocation of the resources to PEs is given in advance (i.e. fixed). Even though the allocation of resources is fixed, there is still possibility to select the best way to exchange messages between PEs. For example, let us assume that there are eight PEs as in Fig. 4(a). Registers $R_5$, $R_6$ and $R_7$ are allocated to $PE_6$, $PE_7$ and $PE_8$, respectively. According to the synthesis rules, these PEs should update these registers, however they do not have the registers used for updating. Here, in our synthesis method, a register can be allocated to more than one PE, and all the copies of the register are updated by those PEs. This means that any of those PEs can send the value of the register to other PEs. In this case, these values can be sent by $PE_2$, $PE_3$, $PE_4$ and $PE_5$ and there are many ways to send these values. Fig. 4(a) shows the optimal way that uses five messages and Fig. 4(b) shows another way that uses six messages. In \cite{16}, the optimization of the total number of messages is formulated and a genetic algorithm is proposed in \cite{26} for solving it efficiently. Our second model \cite{29} extends the first one so that an optimal allocation of registers is determined under a given allocation of gates, which further reduces the communication cost than the first model. A distributed development process is a good application example of such a model. In such a process, different engineers participate in the development process through their own gates using registers (resources, i.e. files and databases).
allocated to their machines. The third model [27] can determine an optimal resource allocation that minimizes the communication cost.

Here we explain the third optimization model, which is the most generalized model. The following 0-1 integer variables are introduced.

- $\alpha_{p,q}^x$ ($\beta_{p,q}^x$, $\gamma_{p,q}^x$): its value is one iff an $\alpha$-message ($\beta$-message, $\gamma$-message) is sent from PE$_p$ to PE$_q$ in the execution of transition $t_x$; otherwise zero. Each of these variables indicates whether a message is sent or not.

- $\beta_{p,q}^R (\gamma_{p,q}^R)$: its value is one iff the $\beta$-message ($\gamma$-message) sent from PE$_p$ to PE$_q$ contains the value of register $R_w$; otherwise zero. Each of these variables indicates whether a message contains a register value or not.

- $ALC_p[G_4]$ ($ALC_p[R_w]$): its value is one iff gate $G_4$ (register $R_w$) is allocated to PE$_p$; otherwise zero.

- $PES_{start}^x_p (PES_{subst}^x_p)$: its value is one iff PE$_p$ is $PES_{start}(t_x)$ ($PE_p$ belongs to $PES_{subst}(T_x)$); otherwise zero.

Then the problem to determine an optimization resource allocation is formulated as an Integer Linear Programming (ILP) problem where an objective function minimizes the communication cost (the number of messages in this case) and linear inequalities represent the synthesis rules in Fig. 3. For example, according to synthesis rule $(S_{M,31})$, inequality $(S_{M,31})$ in Fig. 5 indicates that if PE$_k$ belongs to $PES_{subst}(T_x)$ (i.e., if the value of $PES_{subst}^x_k$ is one), then PE$_k$ must receive at least one $\beta$-message (i.e., the value of $\sum_{j} \beta_{j,k}^x$ must be one).

Moreover, we can consider many communication cost criteria in addition to the number of messages (e.g., the size of messages, and the number of messages depending on the channel utilization) by using corresponding objective functions. The reader may refer to [27] for the detail of the cost criteria.
Objective Function:

\[
\text{Min} : \sum_{x} \sum_{p} \sum_{q \neq q} (\alpha_{p,q}^x + \beta_{p,q}^x + \gamma_{p,q}^x)
\]

Constraints:

(S\textsubscript{A1}):

\[PE\text{start}^x_i - ALC_i[G_s] = 0\]

(S\textsubscript{A2}):

\[PE\text{subst}^x_k - ALC_k[R_w] \geq 0\]

\[\sum_{w} ALC_k[R_w] - PE\text{subst}^x_k \geq 0\]

(S\textsubscript{M1}):

\[PE\text{start}^y_i - \alpha_{i,j}^x \geq 0\]

(S\textsubscript{M31}):

\[\sum_{j} \beta_{j,k}^x - PE\text{subst}^x_k \geq 0\]

(S\textsubscript{M32}):

\[\sum_{j} \beta_{j,k}^x[R_e] + ALC_k[R_e] - ALC_k[R_w] \geq 0\]

(if \(R_e\) is used to change the value of \(R_w\) in \(S(T_e)\))

(S\textsubscript{M33}):

\[\sum_{x} \alpha_{i,j}^x - \beta_{j,k}^x \geq 0\]

(S\textsubscript{M7}):

\[\gamma_{k,m}^x - PE\text{subst}^x_k - PE\text{start}^y_m \geq -1\]

(\(\text{if } T_y \in \mathcal{E} \bullet \bullet \))

(S\textsubscript{M11}):

\[\sum_{i} \alpha_{i,j}^x - PE\text{subst}^x_i - \gamma_{i,m}^x \geq 0\]

(\(\text{if } T_y \in \mathcal{E} \bullet \bullet \))

(S\textsubscript{M71}):

\[\sum_{i} ALC_i[G_s] = 1; \sum_{i} ALC_i[R_w] \geq 1\]

\[\beta_{j,k}^x + ALC_j[R_w] - 2\beta_{j,k}^x[R_w] \geq 0\]

\[\gamma_{i,m}^x + ALC_m[R_w] - 2\gamma_{i,m}^x[R_w] \geq 0\]

(\(\text{Others}\))

Figure 5: Integer Linear Programming Model for Determining Optimal Resource Allocation
Figure 6: (a) Behavior of $PE_1$ and $PE_2$ in Execution of Sequence “$T_1; T_2; T_3; T_4$” of $Spec$ and (b) Sub-PNR’s for $T_1$, $T_2$ and $T_3$

3.3 Synthesis of Protocol Specifications

We derive a protocol specification in the following three steps.

[step 1] Based on the synthesis rules, a set of actions and message exchanges to be executed on each PE is defined for each $T_x$ of $Spec$. Then these actions and message exchanges are represented as a set of transitions where two transitions are connected with a place if a temporal ordering between them is specified in the synthesis rules (e.g. an $\alpha$-message must be received before the corresponding $\beta$-messages are sent). As a result, a subnet is produced for each pair of $T_x$ of $Spec$ and $PE_i$ (called a sub-PNR and denoted as $SPnet^i_x$). Note that $SPnet^i_x$ may be an empty net in the case that $PE_i$ has no action and no
message exchange in the execution of \( T_x \). In this case, we suppose that the sub-PNR has only a single \( \varepsilon \)-transition with a label \{true, \( \tau \), {}\} where \( \tau \) is an internal event.

The behavior of \( PE_1 \) and \( PE_2 \) in Fig. 2(b) in the execution of a transition sequence “\( T_1; T_2; T_3; T_1 \)” of \( S_{spec} \) in Fig. 2(a) is shown as a timing chart in Fig. 6(a). In Fig. 6(a), a temporal ordering between two actions is represented as a dotted arrow. According to those actions, message exchanges and temporal orderings, the sub-PNR’s are produced as shown in Fig. 6(b). Note that the sub-PNR for the pair of \( T_1 \) and \( PE_1 \) is an \( \varepsilon \)-transition.

[step2] Intermediate protocol entity specification of \( PE_1 \) (denoted as \( P_{spec} \)) is derived by connecting all sub-PNR’s \( SPnet_i \) for \( PE_i \) for \( x = 1, 2, . . . , m \) as \( T_1, T_2, . . . \) and \( T_m \) are connected in \( S_{spec} \). More simply, it is obtained using the net structure of \( S_{spec} \), by replacing \( T_x \) with the corresponding sub-PNR \( SPnet_i \) for \( x = 1, 2, . . . , m \). Note that if \( SPnet_i \) has more than one head (or tail) transition, an \( \varepsilon \)-transition is attached as its head (or tail) transition so that the sub-PNR can be treated like a single transition.
$P_{spec2}$ is shown in Fig. 7(a).

**[step 3]** Finally, protocol entity specification of $PE_i$ ($P_{spec_i}$) is derived by removing $\varepsilon$-transitions from $P_{spec_i}$. Note that since $\varepsilon$-transitions have internal events and no substitution statements, they do not affect the behavior of $P_{spec_i}$ except in the case that their input places have choice structure. For example, if an input place of an $\varepsilon$-transition has another output transition $t$ in $P_{spec_i}$ and if $t$ is a transition that receives a message from an other PE, the firing of the $\varepsilon$-transition disables $t$. Thus the PE cannot receive the message if it should be received (this means deadlock). Therefore, we introduce a procedure for removing $\varepsilon$-transitions.

The removing technique is based on the well-known technique to remove $\varepsilon$-moves in finite automata. In order to apply this technique to our PNR model containing parallel synchronization, we use the property of a live and safe Free-choice net where it can be decomposed into a set of live and safe finite state machines (FSM's) [1]. First, for each $\varepsilon$-transition that has $u$ input places and $v$ output places, $(LCM(u, v)/u) - 1$ copies of each input place is produced and $(LCM(u, v)/v) - 1$ copies of each output place is produced where $LCM(u, v)$ is the least common multiple of $u$ and $v$. This procedure never changes the behavior of the net and shows that the number of FSM's that synchronize on the $\varepsilon$-transition is $LCM(u, v)$. Then we split these FSM's by splitting the $\varepsilon$-transition and at this moment the $\varepsilon$-transition is no longer a synchronization point. Then the $\varepsilon$-transition is removed using the technique to remove $\varepsilon$-moves. For example, for the intermediate specification of $PE_2$ in Fig. 7(a), the $\varepsilon$-loop by $\varepsilon$-transition $\varepsilon_1$ can be obviously removed and then the input place $P_1$ of $\varepsilon$-transition $\varepsilon_2$ is copied to $P_{1_a}$ and $P_{1_b}$ (Fig. 7(b)). This means that two FSM's synchronize on $\varepsilon_2$. Then $\varepsilon_2$ is split into two transitions (Fig. 7(c)) and removed. Note that in $P_{spec^{1,2}}$ of Fig. 2(b), $\varepsilon$-transitions $t_{2e}$, $t_{2f}$, $t_{3j}$, $t_{3a}$, $t_{3c}$ and $t_{3h}$ remain for readability of specifications, since their input places do not have choice structure.

The set of all the above protocol entity specifications corresponds to the protocol specification $P_{spec^{1, P}}$. 

19
4 Protocol Re-synthesis

In this section, we present our new method for re-synthesizing the protocol specification from a modified service specification. For given modified $Spec$ and the unmodified $Spec^{(1,P)}$, our protocol re-synthesis method modifies only the corresponding part of $Spec^{(1,P)}$ so that it is equivalent to the modified $Spec$.

The advantage of our re-synthesis method is to avoid a number of applications of complete protocol synthesis for a sequence of minor modifications which often happen in software maintenance. Even for a minor modification, complete protocol synthesis may change most part of the protocol specification, and it makes the maintenance cost much more expensive.

As a simple example of a minor modification, the service specification $Spec$ of Fig. 2(a) is modified from $S(T_1) = \{R_{cnt} ← R_{cnt} + 1\}$ to $S(T_1) = \{R_{cnt} ← (R_{cnt} + 1) \mod R_{cy} \}$ so that the value of $R_{cnt}$
never exceeds $R_{cyc}$ due to the iteration of the loop of $T_1$ depending on the values of inputs. The modified service specification is shown in Fig. 8(a) where the modified part is highlighted. This modification indicates that $PE_1$ needs the value of $R_{cyc}$ to execute $S'(T_1) = \{R_{cnt} \leftarrow (R_{cnt} + 1) \mod R_{cyc}\}$. Then according to our re-synthesis method, new transitions $t_{1c}$, $t_{1d}$, $t_{1e}$ and $t_{1f}$ are inserted and $t_{1b}$'s substitution statement is modified to the new one. The new protocol specification is shown in Fig. 8(b) where the added and modified parts are highlighted.

4.1 Atomic Changes and Re-synthesis Rules

We define a set of simple changes (called atomic changes) for sets of PE's $PESubst(T_x)$ and $PEstart(T_x \bullet \bullet)$, and sets of registers $RSubst_k(T_x)$ and $Rstart(T_x \bullet \bullet)$ (see Table 1). Then we introduce re-synthesis rules applied to $Pspec^{(1, r)}$ corresponding to the atomic changes. The idea is that by presenting modifications to service specifications and resource allocations as sequences of the atomic changes, we deal with the modifications in the set of simple re-synthesis rules.

We consider the following four types of atomic changes.

1. $PESubst(T_x)$, the set of PE's that execute $S(T_x)$, has been changed to $PESubst(T_x) \cup \{PE_k\}$ (or $PESubst(T_x) \setminus \{PE_k\}$).

   * Assume that two registers $R_1$ and $R_2$ are allocated to $PE_1$ and $PE_2$, respectively. In this situation, if $S(T_1) = \{R_1 \leftarrow 1\}$ is modified to $\{R_1 \leftarrow 1, R_2 \leftarrow 2\}$, this modification changes $PESubst(T_1) = \{PE_1\}$ to $PESubst'(T_1) = PESubst(T_1) \cup \{PE_2\}$.

2. $RSubst_k(T_x)$, the set of registers used in $S_k(T_x)$, has been changed to $RSubst_k(T_x) \cup \{R_k\}$ (or $RSubst_k(T_x) \setminus \{R_k\}$).

   * In the same situation above, if $S(T_x) = \{R_1 \leftarrow 1\}$ is modified to $\{R_1 \leftarrow R_2 + 1\}$, this modification changes $RSubst_1(T_x) = \{\}$ to $RSubst'_1(T_x) = RSubst_1(T_x) \cup \{R_2\}$.

3. $PEstart(T_x \bullet \bullet)$, the set of PE's that start the execution of next transitions of $T_x$, has been changed to $PEstart(T_x \bullet \bullet) \cup \{PE_m\}$ (or $PEstart(T_x \bullet \bullet) \setminus \{PE_m\}$).

21
ex. Assume that the set of next transitions of $T_x$ is $\{T_1, T_2\}$ where $E(T_1) = "G_1!R"$ and $E(T_2) = "G_2!R"$. Also assume that $G_1$ and $G_2$ are allocated to $PE_1$ and $PE_2$, respectively. In this situation, if $E(T_1)$ is modified to $"G_2!R"$, this modification changes $PEstart(T_x \bullet \bullet) = \{PE_1, PE_2\}$ to $PEstart'(T_x \bullet \bullet) = PEstart(T_x \bullet \bullet) \setminus \{PE_1\}$.

4. $Restart_m(T_x \bullet \bullet)$, the set of registers used by $PE_m$ to start the execution of next transitions of $T_x$, has been changed to $Restart_m(T_x \bullet \bullet) \cup \{R_b\}$ (or $Restart(T_x \bullet \bullet) \setminus \{R_b\}$).

ex. In the same situation above, if $E(T_1) = "G_1!R"$ is modified to $"G_1!R, R'"$, this modification changes $Restart_1(T_x \bullet \bullet) = \{R\}$ to $Restart'_1(T_x \bullet \bullet) = Restart_m(T_x \bullet \bullet) \cup \{R\}$.

The above four types of atomic changes are represented as $(AC_i)$ ($i = 1, 4$) where each $(AC_i)$ is further classified into $(AC_{i+})$ and $(AC_{i-})$. “+” and “−” correspond to “∪” (addition) and “\” (subtraction), respectively. For atomic changes $(AC_{i+})$ and $(AC_{i-})$, we define their corresponding re-synthesis rules $(RS_{i+})$ and $(RS_{i-})$, respectively. They are given in Fig. 9.

If $(AC_{1+})$ occurs, $PE_k$ needs to know the timing to execute $S_k(T_x)$ and to let PE’s in $PEstart(T_x \bullet \bullet)$ know that the execution of $S_k(T_x)$ had been finished. $(RS_{1+})$ indicates that $PE_k$ must receive a $\beta$-message and send $\gamma$-messages to those PE’s in $PEstart(T_x \bullet \bullet)$.

If $(AC_{2+})$ occurs, $PE_k$ must receive the value of $R_h$ used to execute $S_k(T_x)$ if $PE_k$ does not have $R_h$. $(RS_{2+})$ indicates that $PE_k$ must receive the value of $R_h$ via an existing $\beta$-message from a PE that has $R_h$. If such a $\beta$-message does not exist, a new $\beta$-message including the value of $R_h$ sent from a PE (say $PE_j$) that has register $R_h$ should be added. Note that this new $\beta$-message may need an $\alpha$-message sent from $PEstart(T_x)$ to $PE_j$ in order to know the timing for sending the $\beta$-message.

If $(AC_{3+})$ occurs, each PE in $PE\text{subj}(T_x)$ (say $PE_k$) must let $PE_m$ know that the execution of $S_k(T_x)$ had been finished. $(RS_{3+})$ indicates that $PE_k$ must send a $\gamma$-message to $PE_m$. If $PE\text{subj}(T_x)$ is empty, $PEstart(T_x)$ sends this message.

If $(AC_{4+})$ occurs, $PE_m$ must receive the value of $R_h$ if $PE_m$ does not have $R_h$. $(RS_{4+})$ indicates that $PE_m$ must receive the value of $R_h$ via an existing $\gamma$-message from a PE that has $R_h$. If such a $\gamma$-message
$P E_{\text{subst}}(T_0)$ has been changed:

(AC$_{1+}$)  $P E_{\text{subst}}(T_0) = P E_{\text{subst}}(T_0) \cup \{ P E_i \}$

(RS$_{1+}$)  Add $\gamma_{r_i}^m$ where $P E_i$ = $P E_{\text{start}}(T_0)$ to let $P E_i$ know the timing to execute $S_i(T_0)$. Also add $\gamma_{r_i}^m (\forall P E_m \in P E_{\text{start}}(T_0 \bullet \bullet))$ to let $P E_m$ know that the execution of $S_i(T_0)$ had been finished.

(AC$_{1-}$)  $P E_{\text{subst}}(T_0) = P E_{\text{subst}}(T_0) \setminus \{ P E_i \}$

(RS$_{1-}$)  Delete $\gamma_{r_i}^m (\forall P E_i)$ since $P E_i$ no longer executes $S_i(T_0)$. Also delete $\gamma_{r_i}^m (\forall P E_m \in P E_{\text{subst}}(T_0))$ if it has no value. Finally, if at least one $\gamma_{r_i}^m$ still exists, add $\alpha_{r_i}^j$.

$R_{\text{subst}}(T_0)$ has been changed:

(AC$_{2+}$)  $R_{\text{subst}}(T_0) = R_{\text{subst}}(T_0) \cup \{ R_h \}$

(RS$_{2+}$)  Include the value of $R_h$ in an existing message $\beta_{r_j}^m$ where $P E_j$ has $R_h$, since $P E_j$ needs the value of $R_h$ for the execution of $S_i(T_s)$. If such a message does not exist, add a new $\beta_{r_j}^m$ including the value of $R_h$. Also add $\alpha_{r_j}^j (P E_i = P E_{\text{start}}(T_0), i \neq j)$ if $P E_j$ does not receive $\alpha_{r_j}^j$.

(AC$_{2-}$)  $R_{\text{subst}}(T_0) = R_{\text{subst}}(T_0) \setminus \{ R_h \}$

(RS$_{2-}$)  Exclude the value of $R_h$ from $\beta_{r_j}^m (\forall P E_j)$ since $P E_j$ no longer needs the value of $R_h$ for the execution of $S_i(T_0)$. Then delete $\beta_{r_j}^m$ only if (a) $\beta_{r_j}^m$ exists ($j' \neq j$) and (b) $\beta_{r_j}^m$ has no register values. Finally, delete $\alpha_{r_j}^j (P E_i = P E_{\text{start}}(T_0))$ only if there is no $\beta_{r_j}^m$ and no $\gamma_{r_j}^m$.

$P E_{\text{start}}(T_m \bullet \bullet)$ has been changed:

(AC$_{3+}$)  $P E_{\text{start}}(T_m \bullet \bullet) = P E_{\text{start}}(T_m \bullet \bullet) \cup \{ P E_m \}$

(RS$_{3+}$)  Add $\gamma_{r_i}^m (\forall P E_i \in P E_{\text{subst}}(T_m))$ if it does not exist, since $P E_m$ needs to know that the execution of $S_i(T_0)$ had been finished. If $P E_{\text{subst}}(T_m)$ is an empty set, add $\gamma_{r_i}^m$ where $P E_i = P E_{\text{start}}(T_0)$.

(AC$_{3-}$)  $P E_{\text{start}}(T_m \bullet \bullet) = P E_{\text{start}}(T_m \bullet \bullet) \setminus \{ P E_m \}$

(RS$_{3-}$)  Delete $\gamma_{r_i}^m (\forall P E_i)$ if it exists, since $P E_m$ no longer needs to know that the execution of $S_i(T_0)$ had been finished.

$R_{\text{start}}(T_m \bullet \bullet)$ has been changed:

(AC$_{4+}$)  $R_{\text{start}}(T_m \bullet \bullet) = R_{\text{start}}(T_m \bullet \bullet) \cup \{ R_h \}$

(RS$_{4+}$)  Include the value of $R_h$ in an existing message $\gamma_{r_i}^m$ where $P E_i$ has $R_h$, since $P E_m$ needs the value of $R_h$ for the execution of next transitions of $T_m$. If such a message does not exist, add a new $\gamma_{r_i}^m$ including the value of $R_h$. Also add $\alpha_{r_i}^j (P E_i = P E_{\text{start}}(T_m), i \neq j)$ if $P E_i$ does not receive $\alpha_{r_i}^j$ and $P E_i \notin P E_{\text{subst}}(T_m)$.

(AC$_{4-}$)  $R_{\text{start}}(T_m \bullet \bullet) = R_{\text{start}}(T_m \bullet \bullet) \setminus \{ R_h \}$

(RS$_{4-}$)  Exclude the value of $R_h$ from $\gamma_{r_i}^m (\forall P E_i)$ since $P E_m$ no longer needs the value of $R_h$ for the execution of the next transitions of $T_m$. Then delete $\gamma_{r_i}^m$ only if (a) $P E_i \notin P E_{\text{subst}}(T_m)$, (b) $\gamma_{r_i}^m$ exists ($i \neq j$), and (c) $\gamma_{r_i}^m$ has no register value. Finally, delete $\alpha_{r_i}^j (P E_i = P E_{\text{start}}(T_m))$ only if there is no $\beta_{r_j}^m$ and no $\gamma_{r_j}^m (\forall P E_j)$.

---

Figure 9: Atomic Changes and Their Corresponding Re-synthesis Rules

---

23
does not exist, a new \(\gamma\)-message including the value of \(R_h\) sent from a PE (say \(PE_k\)) that has register \(R_h\) should be added. Note that this new \(\gamma\)-message may need an \(\alpha\)-message sent from \(P_{\text{start}}(T_x)\) to \(PE_k\) in order to know the timing for sending the \(\gamma\)-message.

\((AC_i^\bot)\) and \((RS_i^\bot)\) \((i = 1..4)\) are the opposite cases of \((AC_i^+)\) and \((RS_i^+)\).

For example, the modification to the service specification in Fig. 8(a) causes atomic change \((AC_2^+)\), where \(R_{\text{substit}}(T_1) = R_{\text{substit}}(T_1) \cup \{R_{xy}\}\). According to re-synthesis rule \((RS_2^+)\), a new \(\beta\)-message “\(\beta_2^1\)” is added to send the value of \(R_{xy}\) since no \(\beta\)-message has been sent from \(PE_2\) to \(PE_1\). Moreover, in order to let \(PE_2\) know the timing to send this \(\beta\)-message, an \(\alpha\)-message “\(\alpha_2^1\)” is sent from \(PE_1\) to \(PE_2\).

### 4.2 Re-synthesis of Protocol Specifications

According to the re-synthesis rules, some messages (or only some values in messages) and actions may be added or deleted. If a message \(M_{ij}^p\) should be added or deleted, we modify \(P_{\text{spec}i}\) and \(P_{\text{spec}j}\) as follows.

- For adding a new message \(M_{ij}^p\), two transitions that send and receive the message are added to \(SPnet_i^p\) and \(SPnet_j^p\), respectively. They are inserted satisfying the temporal relations between the existing transitions of \(SPnet_i^p\) and \(SPnet_j^p\) as specified in the synthesis rules in Fig. 3. Note that if \(SPnet_i^p\) (or \(SPnet_j^p\)) is an empty net, it has been deleted as an \(\varepsilon\)-transition in the step3 in Section 3.3. In this case, the \(\varepsilon\)-transition can be restored by connecting it with its original input/output places.

- For deleting an existing message \(M_{ij}^p\), two transitions that send and receive the message are deleted from \(SPnet_i^p\) and \(SPnet_j^p\), respectively. In this case, these transitions are replaced with \(\varepsilon\)-transitions and removed using the rules in Section 3.3.

In Fig. 8(b), in order to add two new messages \(\alpha_{12}\) and \(\beta_{21}^1\), two transitions that send \(\alpha_{12}\) and receive \(\beta_{21}^1\) are added to \(SPnet_1^1\), and two transitions that receive \(\alpha_{12}\) and send \(\beta_{21}^1\) are added to \(SPnet_2^1\). Note that \(SPnet_2^1\) has been deleted as an \(\varepsilon\)-transition. In such a case, we use the intermediate specification of \(P_{\text{spec}2}\) (i.e. \(P_{\text{spec}2}'\)) where \(SPnet_2^1\) remains as an \(\varepsilon\)-transition.
4.3 Generic Modification and Re-synthesis Example

In general, modifications to service specifications and resource allocations may cause complex changes in protocol specifications. In order to treat such modifications in a simple manner, we present these complex changes as sequential occurrences of atomic changes in Fig. 9 and apply the corresponding re-synthesis rules sequentially.

We assume three types of modifications by which (a) the label of a transition in a service specification is changed (event, pre-condition and set of substitution statements), (b) a new transition is added or an existing transition is deleted in a service specification, and (c) a resource allocation is changed. Hereafter, \((AC^t_i)\) denotes atomic change \((AC_i)\) concerning transition \(T_x\).

**(a) label of a transition is modified** Assume that the event, pre-condition and set of substitution statements of \(T_w\) (\(E(T_w), C(T_w)\) and \(S(T_w)\) are changed to \(E'(T_w), C'(T_w)\) and \(S'(T_w)\), respectively. This modification causes a sequence of atomic changes as follows.

1. If \(E(T_w)\) has used \(G_s\) and \(E'(T_w)\) uses \(G_s'\), and if \(G_s\) and \(G_s'\) are allocated to \(PE_m\) and \(PE_{m'}\) respectively, \(PE_m = PE_{start}(T_x)\) may be removed from \(PE_{start}(T_w \bullet \bullet)\) and \(PE_m = PE_{start'}(T_x)\) may be newly added to \(PE_{start}(T_w \bullet \bullet)\) for each previous transition \(T_w\) of \(T_x\). Moreover, some registers may be deleted from \(Rest_{start}(T_w \bullet \bullet)\) and added to \(Rest_{start'}(T_w \bullet \bullet)\) because registers in \(E(T_w)\) or \(C(T_w)\) may be different from those in \(E'(T_w)\) or \(C'(T_w)\). Therefore, it is regarded that the following sequence of atomic changes occurs for each \(T_w \in \bullet \bullet T_x\).

(a) For \(PE_m = PE_{start}(T_x)\) and each \(R_h \in Rest_{start}(T_w \bullet \bullet)\),

\[
(AC^w_{1-}) : Rest^l_{start}(T_w \bullet \bullet) = Rest^l_{start}(T_w \bullet \bullet) \setminus \{R_h\}
\]

(b) If \(PE_m \in PE_{start}(T_w \bullet \bullet) \setminus PE_{start'}(T_w \bullet \bullet)\),

\[
(AC^w_{2-}) : PE_{start'}(T_w \bullet \bullet) = PE_{start}(T_w \bullet \bullet) \setminus \{PE_m\}
\]

(c) If \(PE_{m'} \in PE_{start}(T_w \bullet \bullet) \setminus PE_{start'}(T_w \bullet \bullet)\),

\[
(AC^w_{3-}) : PE_{start'}(T_w \bullet \bullet) = PE_{start}(T_w \bullet \bullet) \cup \{PE_{m'}\}
\]
(d) For $PE_{m'} = PE_{start^l(T_x)}$ and each $R_{ly} \in Restart^l_m(T_w \bullet \bullet \bullet \bullet) \setminus Restart^l_m(T_w \bullet \bullet \bullet)$,

\[
    (AC_{4+}^u) : \ restart^l_m(T_w \bullet \bullet \bullet) = \ restart^l_m(T_w \bullet \bullet \bullet) \cup \{ R_{ly} \}
\]

2. $PE_{\text{subst}^l(T_x)}$ may be different from $PE_{\text{subst}(T_x)}$ because registers that are updated in $S'(T_x)$ may be different from those in $S(T_x)$. Moreover, for each $PE_i$ in $PE_{\text{subst}(T_x)}$, some registers may be deleted from $R_{\text{subst}_i}(T_x)$ and for each $PE_i$ in $PE_{\text{subst}^l(T_x)}$, some registers may be added to $R_{\text{subst}_i}(T_x)$. Therefore, it is regarded that the following sequence of atomic changes occurs for $T_x$.

(a) For each $PE_i \in PE_{\text{subst}(T_x)}$ and each $R_i \in R_{\text{subst}_i}(T_x) \setminus R_{\text{subst}_i}^l(T_x)$,

\[
    (AC_{2+}^u) : \ R_{\text{subst}_i}^l(T_x) = R_{\text{subst}_i}(T_x) \setminus \{ R_i \}
\]

(b) For each $PE_k \in PE_{\text{subst}(T_x) \setminus PE_{\text{subst}^l(T_x)}$,

\[
    (AC_{1-}^u) : \ PE_{\text{subst}^l(T_x)} = PE_{\text{subst}(T_x)} \setminus \{ PE_k \}
\]

(c) For each $PE_h \in PE_{\text{subst}^l(T_x) \setminus PE_{\text{subst}(T_x)}$,

\[
    (AC_{1+}^u) : \ PE_{\text{subst}^l(T_x)} = PE_{\text{subst}(T_x)} \cup \{ PE_h \}
\]

(d) For each $PE_r \in PE_{\text{subst}^l(T_x)}$ and each $R_{ly} \in R_{\text{subst}_r}(T_x) \setminus R_{\text{subst}_r}(T_x)$,

\[
    (AC_{2+}^u) : \ R_{\text{subst}_r}(T_x) = R_{\text{subst}_r}(T_x) \setminus \{ R_{ly} \}
\]

We apply their corresponding re-synthesis rules sequentially.

An example, assume that the label of transition $T_3$ in $S_{spec}$ of Fig. 2(a) is modified to

\[
    \langle S'(T_3), S'(T_3), S'(T_3) \rangle = \langle \text{"true", } \text{"false", } \{ \} \rangle
\]

so that $R_{cut}$ is no longer used. The modified $S_{spec}$ is shown in Fig. 10(a).

1. $PE_{start^l(T_3)} = PE_{start(T_3)} = PE_2$. Therefore, $PE_{start^l(T_2 \bullet \bullet \bullet)} = PE_{start(T_2 \bullet \bullet \bullet)} = \{ PE_2 \}$.

   Also, $Restart_2(T_2 \bullet \bullet \bullet) = \{ R_{max}, R_{cut} \}$ and $Restart_2^l(T_2 \bullet \bullet \bullet) = \{ R_{max} \}$.

   (a) $(RS_{1-}^u)$ is applied for register $R_{cut}$, and the value of $R_{cut}$ is excluded from the message $\gamma_{T_2}^2$ sent on transition $t_{2d}$ of Fig. 2(b).

   (b) No $(RS_{2-}^u)$ is applied.

26
Figure 10: (a) Label of $T_3$ is Modified in $S_{\text{spec}}$ and (b) $P_{\text{spec}}^{(1,2)}$ is Re-synthesized.

(c) No $(RS_{2,+}^3)$ is applied.

(d) No $(RS_{2,-}^3)$ is applied.

2. $P_{\text{sub}}(T_3) = \{PE_1\}$ and $P_{\text{sub}}'(T_3) = \emptyset$. Also, $R_{\text{sub}}(T_3) = \{R_{\text{cyc}}\}$ and $R_{\text{sub}}'(T_3) = \emptyset$.

(a) $(RS_{2,-}^3)$ is applied for register $R_{\text{cyc}}$, and the value of $R_{\text{cyc}}$ is excluded from the message “$\beta_{21}^i$” sent on transition $t_{3,f}$ of Fig. 2(b).

(b) $(RS_{2,+}^3)$ is applied for $PE_1$, and the message “$\beta_{21}^i$” is deleted. This makes $t_{3,f}$ and $t_{3,b}$ be $\varepsilon$-transitions.

(c) No $(RS_{1,+}^3)$ is applied.

(d) No $(RS_{2,+}^3)$ is applied.
3. Finally, $S(T_x)$ is deleted from $t_{3e}$. (This makes $t_{3e}$ be an $\varepsilon$-transition.

The re-synthesized protocol specification is shown in Fig. 10(b). In this protocol specification, $\varepsilon$-transitions $t_{3a}$, $t_{3b}$, $t_{3c}$, $t_{3e}$, $t_{3f}$, and $t_{3h}$ are deleted, and the value of register $R_{env}$ is excluded from the message $^{a}g^{2}_{12}$ on transitions $t_{2a}$ and $t_{2i}$.

[(b) a transition is inserted/removed] If a new transition $T_x$ is inserted to $Spec$, $SPnet_k^2$ for each $PE_k$ is synthesized according to the synthesis rules. Then each $SPnet_k^2$ is inserted to the intermediate specification of $PE_k$ ($\overline{Pspec_k}$), and $Pspec_k$ is obtained by removing $\varepsilon$-transitions from $\overline{Pspec_k}$. On the other hand, if an existing transition $T_x$ is removed from $Spec$, each transition in $SPnet_k^2$ is replaced with an $\varepsilon$-transition and then removed according to the procedure explained in Section 3.3. However, in both cases, for each previous transition $T_w$ of $T_x$, $PEstart(T_w \bullet \bullet)$ and $Restart_m(T_w \bullet \bullet)$ may be changed. Therefore, it is regarded that the following sequence of atomic changes occurs.

1. For each $PE_m \in PEstart(T_w \bullet \bullet)$ and each $R_h \in Restart_m(T_w \bullet \bullet)\setminus Restart'_m(T_w \bullet \bullet)$,
   $$(AC_{3a}^w) : Restart'_m(T_w \bullet \bullet) = Restart_m(T_w \bullet \bullet) \setminus \{R_h\}$$

2. For each $PE_m \in PEstart(T_w \bullet \bullet) \setminus PEstart(T_w \bullet \bullet)$,
   $$(AC_{3b}^w) : PEstart(T_w \bullet \bullet) = PEstart(T_w \bullet \bullet) \setminus \{PE_m\}$$

3. For each $PE_m' \in PEstart(T_w \bullet \bullet) \setminus PEstart(T_w \bullet \bullet)$,
   $$(AC_{3c}^w) : PEstart(T_w \bullet \bullet) = PEstart(T_w \bullet \bullet) \cup \{PE_m'\}$$

4. For each $PE_m' \in PEstart(T_w \bullet \bullet)$ and each $R_h' \in Restart_m'(T_w \bullet \bullet) \setminus Restart_m'(T_w \bullet \bullet)$,
   $$(AC_{3d}^w) : Restart_m'(T_w \bullet \bullet) = Restart_m(T_w \bullet \bullet) \cup \{R_h'\}$$

We apply the corresponding re-synthesis rules sequentially.

As an example, a place $P_3$ and a transition $T_4$ with the following label

$$\langle C(T_4), E(T_4), S(T_4) \rangle = \langle \text{"true"}, \text{"Genv'Renv"}, \text{"\{\}"} \rangle$$

28
Figure 11: (a) Transition $T_4$ is inserted in $S_{spec}$ and (b) $P_{spec}^{1,2}$ is re-synthesized.

are inserted between $T_3$ and $P_1$ of $S_{spec}$ in Fig. 10(a). The modified $S_{spec}$ is shown in Fig. 11(a).
In this case, sub-PNR’s $SP_{net_1}$ and $SP_{net_2}$ are synthesized for $PE_1$ and $PE_2$, and inserted to their intermediate specifications, respectively. Then, because $PE_{start}(T_3 \bullet \bullet) = \{PE_1\}$ has been changed to $PE_{start}'(T_3 \bullet \bullet) = \{PE_2\}$, the following sequence of re-synthesis rules is applied.

1. $(RS_{\|_\perp}^i)$ is applied for $PE_1 \in PE_{start}(T_w \bullet \bullet)$ and $R_{max} \in R_{start_1}(T_3 \bullet \bullet) \setminus R_{start_1}'(T_3 \bullet \bullet)$.

2. $(AC_{\|_\perp}^i)$ is applied for $PE_1 \in PE_{start}(T_3 \bullet \bullet) \setminus PE_{start}'(T_3 \bullet \bullet)$.

3. $(AC_{\|_\perp}^i)$ is applied for $PE_2 \in PE_{start}'(T_3 \bullet \bullet) \setminus PE_{start}(T_3 \bullet \bullet)$.

4. $(AC_{\|_\perp}^i)$ is applied for $PE_2 \in PE_{start}'(T_3 \bullet \bullet)$ and $R_{cnt} \in R_{start_2}(T_3 \bullet \bullet) \setminus R_{start_2}(T_3 \bullet \bullet)$.
As a result, transitions \( t_{3d} \) and \( t_{3g} \) are deleted and new transitions \( t_{3i}, t_{3j}, t_{3k} \) and \( t_{3l} \) are inserted (see Fig. 11(b)).

Note that the assumptions made on \( S_{spec} \) as presented in Section 3.1 must also hold on the modified \( S_{spec} \), i.e., (a) the Petri net of modified \( S_{spec} \) must be a live and safe Free-choice net, (b) only one PE makes a decision to select two transitions in any choice structure, and (c) two parallel transitions of the modified \( S_{spec} \) must not refer/update the value of the same register. For (a), a set of rules is presented in [1] to transform a free-choice net keeping its liveness and safeness. We assume that users follow those 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 [1].

[(c) resource allocation is changed] As explained in Section 3.2, our synthesis method allows a single register to be allocated to multiple PE’s. This means that the values of all the registers that have the same name must be updated to keep an identical value between them. This idea is very useful in some application areas. For example, in distributed databases, adding a copy of an existing register to some PE’s increases the robustness to fault, balances the load and reduces communication costs among these PE’s.

Here, assume that an existing register \( R_h \) allocated to some PE(s) is now also allocated to \( PE_k \). This means that for each \( T_x \), if \( R_h \) is updated in \( S(T_x) \), \( \{ PE_k \} \) may be added to \( PE_{subst}(T_x) \). Moreover, some registers may be added to \( R_{subst_k}(T_x) \) because \( PE_k \) needs those registers to update \( R_h \). On the other hand, \( R_h \) may be deleted from \( R_{subst_k}(T_x) \) and \( Restart_k(T_x \bullet \bullet) \) since now \( PE_k \) has \( R_h \).

1. For each \( T_x \) where \( R_h \in R_{subst_k}(T_x)\setminus R_{subst_k}'(T_x) \),

\[
(AC_{2^-}) : R_{subst_k}'(T_x) = R_{subst_k}(T_x)\setminus \{ R_h \}
\]

2. For each \( T_x \) where \( R_h \in Restart_k(T_x \bullet \bullet)\setminus Restart_k'(T_x \bullet \bullet) \),

\[
(AC_{4^-}) : Restart_k'(T_x \bullet \bullet) = Restart_k(T_x \bullet \bullet)\setminus \{ R_h \}
\]

3. For each \( T_x \) where \( PE_k \in PE_{subst}(T_x)\setminus PE_{subst}(T_x) \),

\[
(AC_{6^-}) : PE_{subst}(T_x) = PE_{subst}(T_x)\setminus \{ PE_k \}
\]

30
\[(AC_{1+}^{R_{k}}) : PEs_{\text{Subst}}^{l}(T_{x}) = PEs_{\text{Subst}}^{l}(T_{x}) \cup \{PE_{k}\}\]

4. For each \(T_{x}\) and each register \(R_{z} \in R_{\text{Subst}}^{l}(T_{x}) \backslash R_{\text{Subst}}^{l}(T_{x})\),
\[(AC_{2+}^{R_{k}}) : R_{\text{Subst}}^{l}(T_{x}) = R_{\text{Subst}}^{l}(T_{x}) \cup \{R_{z}\}\]

On the other hand, if register \(R_{h}\) allocated to more than one PE is removed from \(PE_{k}\), the opposite sequence of the above case is applied.

1. For each \(T_{x}\) and each register \(R_{z} \in R_{\text{Subst}}^{l}(T_{x}) \backslash R_{\text{Subst}}^{l}(T_{x})\),
\[(AC_{2-}^{R_{k}}) : R_{\text{Subst}}^{l}(T_{x}) = R_{\text{Subst}}^{l}(T_{x}) \backslash \{R_{z}\}\]

2. For each \(T_{x}\) where \(PE_{h} \in PEs_{\text{Subst}}(T_{x}) \backslash PEs_{\text{Subst}}^{l}(T_{x})\),
\[(AC_{1-}^{R_{k}}) : PEs_{\text{Subst}}^{l}(T_{x}) = PEs_{\text{Subst}}(T_{x}) \cup \{PE_{h}\}\]

3. For each \(T_{x}\) where \(R_{h} \in \text{Restart}_{h}(T_{x} \cdot \bullet \bullet) \backslash \text{Restart}_{h}(T_{x} \cdot \bullet \bullet)\),
\[(AC_{3+}^{R_{k}}) : \text{Restart}_{h}(T_{x} \cdot \bullet \bullet) = \text{Restart}_{h}(T_{x} \cdot \bullet \bullet) \cup \{R_{h}\}\]

4. For each \(T_{x}\) where \(R_{h} \in \text{Subst}_{h}(T_{x}) \backslash \text{Subst}_{h}(T_{x})\),
\[(AC_{3+}^{R_{k}}) : \text{Subst}_{h}(T_{x}) = \text{Subst}_{h}(T_{x}) \cup \{R_{h}\}\]

We apply the corresponding re-synthesis rules sequentially.

As an example, assume that a copy of register \(R_{max}\) allocated to \(PE_{2}\) is also allocated to \(PE_{1}\) in \(P_{\text{spec}}^{1,2}\) of Fig.11(b). In this case,

1. No \((RS_{x}^{l})\) \((x = 1, 2, 3, 4)\) is applied.

2. \((RS_{1}^{l})\) and \((RS_{2}^{l})\) are applied for \(T_{1}\) and \(T_{2}\), respectively.

3. \((RS_{3}^{l})\) is applied for \(T_{2}\).

4. No \((RS_{x}^{l})\) \((x = 1, 2, 3, 4)\) is applied.

Note that one may need such a modification that moves register \(R_{h}\) from one PE \(PE_{k}\) to another PE \(PE_{l}\). This case can be treated as a pair of the above modifications where \(R_{h}\) allocated to \(PE_{k}\) is also allocated to \(PE_{l}\) and then \(R_{h}\) on \(PE_{k}\) is removed.

31
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[15], distributed cooperative work management[14] and so on.

We apply our synthesis method and re-synthesis method to the distributed development of software that involves five engineers (project manager, quality assurance, design, and two software engineers). Each engineer has his own machine connected with the others, and participates in the development through a gate (interfaces) of this machine, using distributed resources placed on this machine. This distributed development process includes scheduling and assigning tasks, design modification, design review, code modification, test plans modification, modification of unit test packages, unit testing, and progress monitoring tasks. The engineers cooperate with each other to finish these sub-sequential tasks. The reader may refer to ISPW-6 [30] 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.

Figure 12 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 process tasks in Fig. 12.

5.2 Experimental Results

We regard this workflow as the service specification, and we derive its corresponding protocol specification using our synthesis method and program developed in [27]. The tool lp_solve[31] is used to solve the ILP problem to determine the optimal allocation of resources shown in Table 2. Due to the limitation of space, the derived protocol specification is omitted.

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

We consider the following modifications to the given service specification:
<table>
<thead>
<tr>
<th></th>
<th>$P_{Emng}$</th>
<th>$P_{Edc}$</th>
<th>$P_{Ese1}$</th>
<th>$P_{Ese2}$</th>
<th>$P_{Eqa}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gate</td>
<td>MNG</td>
<td>DE</td>
<td>SE1</td>
<td>SE2</td>
<td>QA</td>
</tr>
<tr>
<td>Register</td>
<td>$R_{req}$</td>
<td>$R_{design}$</td>
<td>$R_{design,ab}$</td>
<td>$R_{visit,est}$</td>
<td>$R_{visit,est}$</td>
</tr>
<tr>
<td></td>
<td>$R_{des1}$</td>
<td>$R_{des2}$</td>
<td>$R_{test}$</td>
<td>$R_{test}$</td>
<td>$R_{des}$</td>
</tr>
</tbody>
</table>

Table 2: Optimal Allocation of Resources for Engineers' Machines

<table>
<thead>
<tr>
<th>Synthesis Time (sec.)</th>
<th>Number of Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-synthesis</td>
<td>Complete Synthesis</td>
</tr>
<tr>
<td>case1</td>
<td>1</td>
</tr>
<tr>
<td>case2</td>
<td>1</td>
</tr>
<tr>
<td>case3</td>
<td>1</td>
</tr>
<tr>
<td>case4</td>
<td>1</td>
</tr>
</tbody>
</table>

MMX-Pentium 200 MHz, 128MB Memory

Table 3: Experimental Results

1. An additional source code (register $R_{code, new}$) 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 $T_{20}$).

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

3. DE analyzes the test feedback (register $R_{test,ab}$) and gives his comments to QA. For this purpose, a new register $R_{report}$ is introduced on DE’s machine and his comments are stored on it in transition $T_{20}$. Then it is shown to QA on $T_{25}$.

4. For fault tolerance, a new copy of the existing code $R_{code}$ (placed on $P_{Edc}$) is placed on $P_{Emng}$.

After each modification, we have used the program developed in [27] to measure the time (in seconds)
it takes to derive a completely new protocol specification. Moreover, we have also measured the time it took to re-derive the protocol specifications using the re-synthesis rules and a program that we have developed for this purpose. Table 3 shows these times. The reader can clearly see that the re-synthesis time is much less than the time for 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 specifications after each modification. Moreover, we do not have to re-optimize the number of messages sent between different PE’s because (as shown in Table 3) the re-derived protocol specifications still have optimal (or near-optimal in general cases) solutions.

6 Conclusion

We have proposed a new synthesis method to re-synthesize only the corresponding parts of the current protocol specification after modifications to a 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. This would make protocol synthesis and maintenance more practical for realistic applications.

Currently, we are investigating the extension of our re-synthesis method to specifications modeled as timed Petri nets.

References


