Protocol Re-synthesis Based on Extended Petri Nets*

Khaled El-Fakih[1], Hirozumi Yamaguchi[2],
Gregor v. Bochmann[1], and Teruo Higashino[2]

[1] School of Information Technology and Engineering, University of Ottawa,
150 Louis Pasteur, Ottawa, Ontario K1N 6N5, Canada
{kelfakih,bochmann}@site.uottawa.ca
[2] Graduate School of Engineering Science, Osaka University,
1-3 Machikaneyamacho, Toyonaka, Osaka 560-8531, Japan
{h-yamagu,higashino}@ics.es.osaka-u.ac.jp

Abstract. Protocol synthesis is used to derive a specification of a distributed system from the specification of the services to be provided by the system to its users. Maintaining such a system involves applying frequent minor modifications to the service specification due to changes in the user requirements. In order to reduce the maintenance costs of such a system, we present an original method that consists of a set of rules that avoid complete protocol synthesis after these modifications. These rules are given for a system modeled as an extended Petri net. An application example is given along with some experimental results.

1 Introduction

Synthesis methods have been used (for surveys see [5, 6]) to derive a specification of a distributed system (hereafter called *protocol specification*) automatically from a given specification of the service to be provided by the distributed system to its users (called *service specification*). The service specification is written like a program of a centralized system, and does not contain any specification of the message exchange between different physical locations. However, the protocol specification contains the specification of communications between protocol entities (PE's) at the different locations.

A number of existing protocol synthesis strategies have been described in the literature. The first strategy, [9, 3, 4, 8, 10, 12, 14, 17, 18], aims at implementing complex control-flows using several computational models such as LOTOS, Petri nets, FSM/EFSM and temporal logic. The second strategy, [20, 23, 19, 24, 22], 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, [21, 25, 11, 15, 7, 16], deals with the management of distributed resources such as files and databases. The objective here,

* This work was partially funded by Communications and Information Technology Ontario (CITO).

is to determine how the values of these distributed resources are updated or exchanged between PE's for a given fixed resource allocation on different physical locations.

Some methods in the last strategy, especially these presented in our previous research work[26], have tried to synthesize a service specification by deriving its corresponding protocol specification with minimum communication costs and optimal allocation of resources.

As an example, we consider a Computer Supported Cooperative Work (CSCW) software development process. This process is distributed among engineers (developers, designers, managers and others). Each engineer has his own machine (PE) and participates in the development process using distributed resources (drafts, source codes, object codes, multimedia video and audio files, and others) placed on different machines. Considering the need for using these resources between different computers, we derive, using our protocol synthesis method, the engineer's sub-processes (protocol specification) knowing the whole software development cycle (service specification) and we decide on an allocation of resources that would minimize the communication costs. Both the service and protocol specifications are described using extended Petri nets.

In realistic applications, maintaining a system modeled by a given extended Petri net specification, involves modifying its specification as a result of changes in the user requirements. Synthesizing the whole system after each modification is considered expensive and time consuming. Therefore, it is important to re-synthesize the modified parts of service specification in order to reduce the maintenance cost, which was reported to account for as much as two-thirds of the cost of software production [30].

In this paper, we present a new method for re-synthesizing the protocol specification from a modified service specification. The method consists of a set of rules that would be applied to different PE's after a modification to the service specification, in order to produce new synthesized (henceforth called re-synthesized) PE's. The parts of the protocol specification that correspond to the unmodified parts of the service specification are preserved intact. As shown later, this method reduces the cost of synthesizing the whole system after each modification.

This paper is organized as follows. Section 2 gives examples of service and protocol specifications, and Section 3 describes the protocol synthesis method. Based on this method, we present in Section 4 protocol re-synthesis method along with some application examples in Section 5. Section 6 concludes this paper and includes our insights for future research.

2 Service Specification and Protocol Specification

2.1 Petri Net Model with Registers

We use an extended Petri net model called a *Petri Net with Registers* (*PNR* in short) [15] to describe both service and protocol specifications of a distributed

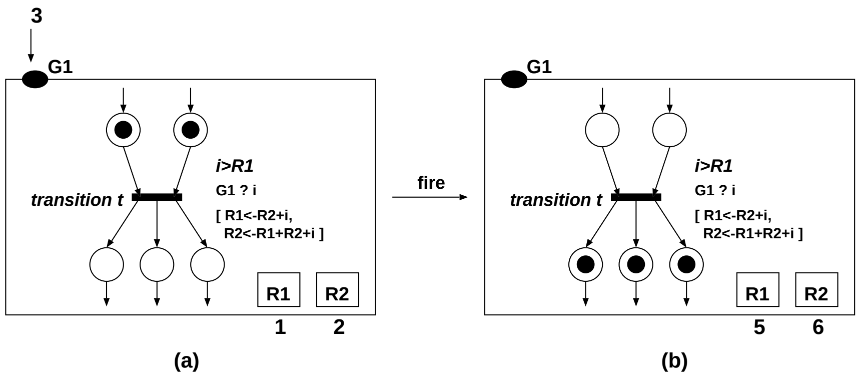

Fig. 1. Register Values and Token Locations before and after Firing of Transition in PNR

system. In this model, an I/O event between users and the system followed by the calculation of new values of variables inside the system is associated with the firing of a transition. Since distributed systems contain some variables (*e.g.* databases and files) and their values are updated according to inputs from users, they can be modeled by PNR naturally.

Each transition t in PNR has a label $\langle \mathcal{C}(t), \mathcal{E}(t), \mathcal{S}(t) \rangle$, where $\mathcal{C}(t)$ is a pre-condition statement (one of the firing conditions of t), $\mathcal{E}(t)$ is an event expression (which represents I/O) and $\mathcal{S}(t)$ is a set of substitution statements (which represents parallel updates of data values). Consider, for example, transition t of Fig. 1 where $\mathcal{C}(t) =$"$i > R_1$", $\mathcal{E}(t) =$"$G_1?i$" and $\mathcal{S}(t) =$"$R_1 \leftarrow R_2+i, R_2 \leftarrow R_1+R_2+i$". i is an input variable, which keeps an input value and its value is referred by only the transition t. R_1 and R_2 are registers, which keep assigned values until new values are assigned, and their values may be referred and updated by all the transitions in PNR (that is, global variables). G_1 is a gate, a service access point (interaction point) between users and the system. Note that "?" in $\mathcal{E}(t)$ means that $\mathcal{E}(t)$ is an input event.

A transition may fire if (a) each of its input place 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). 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 the value of "$i > R_1$" is true and the transition may fire. If it fires, the event "$G_1?i$" 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. Therefore 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: "$G_s \ !exp$", "$G_s \ ?iv$", or "τ". "$G_s \ !exp$" is an output event and it means that the value of expression "exp", whose arguments are registers, is output through gate G_s. "$G_s \ ?iv$" is

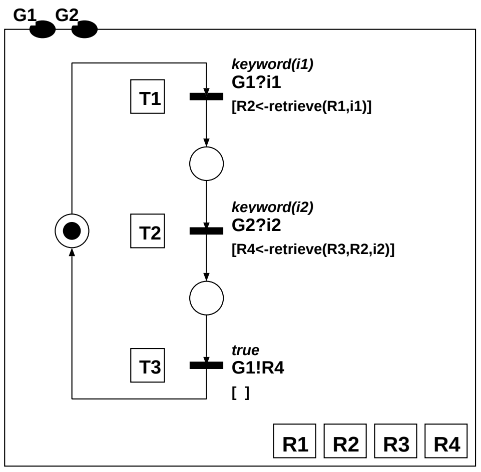

Fig. 2. Service Specification

an input event and it means that the value given through G_s is assigned to the input variable "iv". "τ" is an internal event, which is unobservable from the users. $\mathcal{S}(t)$ is a set of substitution statements, each of the form "$R_w \leftarrow exp_w$", where R_w is a register and exp_w is an expression whose arguments are from the input variable in $\mathcal{E}(t)$ and registers. If t fires, $\mathcal{E}(t)$ is executed followed by the parallel execution of statements in $\mathcal{S}(t)$.

2.2 Service Specification

At a highly abstracted 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 among them are hidden. The specification of the distributed system at this level is called a *service specification* and denoted by $Sspec$.

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. Fig. 2 shows $Sspec$ of a simple database system which has only three transitions. The system receives a keyword (input variable i_1) through gate G_1, retrieves an entry corresponding to the keyword from a database (register R_1), and stores the result to register R_2. This is done on transition T_1. Then the system receives another keyword (input variable i_2) through gate G_2, retrieves an entry corresponding to the keyword and the retrieved entry (register R_2) from another database (register R_3), and stores the result to register R_4. This is done on transition T_2. Finally the system outputs the second result (the value of register R_4) through G_1 on transition T_3 and returns to the initial state.

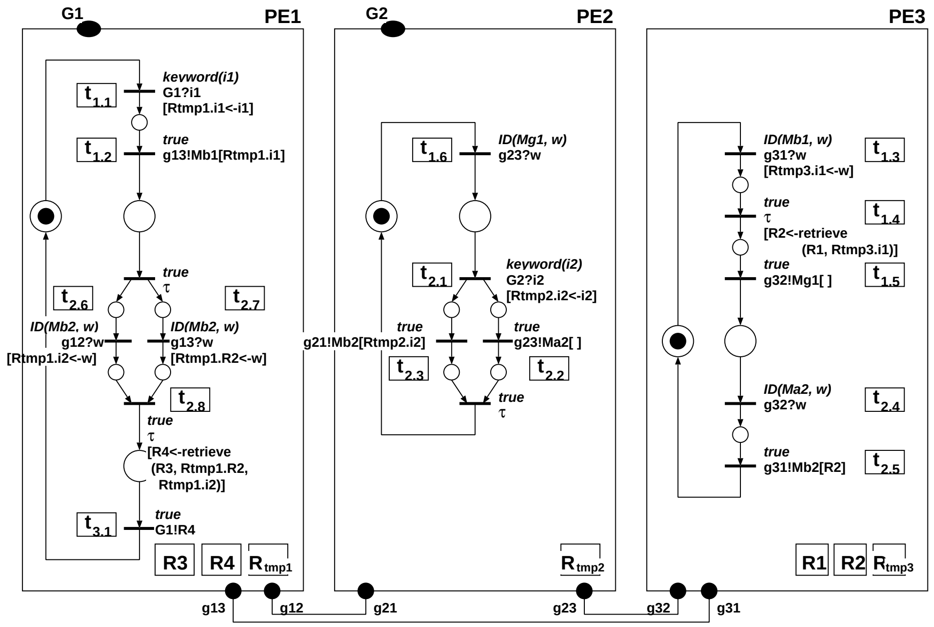

Fig. 3. Protocol Specification

2.3 Protocol Specification

A distributed system is a communication system which consists of p protocol entities PE_1, PE_2, ... and PE_p. We assume a duplex and reliable communication channel with infinite capacity buffers at both ends, between any pair of PE_i and PE_j. The PE_i (PE_j) side of the communication channel is represented as gate g_{ij} (g_{ji}). Moreover, we assume that some resources (registers and gates) are allocated to certain PE's of the distributed system.

Two PE's communicate with each other by exchanging messages. If PE_i executes an output event "$g_{ij}!M[R_w]$", the value of register R_w located on PE_i is sent to PE_j through the communication channel between them and put into the buffer at PE_j's end. M is an identifier to distinguish several values which may exist at the same time on the same channel. PE_j can take the value identified by M from the buffer, by executing an input event "$g_{ji}?w$" with a pre-condition $ID(M, w)$. $ID(M, w)$ is a predicate whose value is true *iff* the identifier in input variable w is M. Note that more than one register's or input variable's value can be sent at a time. If a received data contains multiple values, they are distinguished by suffix such as $w.R_1$ and $w.i$. A set of an identifier and register/input values is called a message. A message may contain no value and sending such a message is represented as an output event "$g_{ij}!M[\ ]$".

In order to implement a distributed system which consists of p PE's, we must specify the behavior of these PE's. A specification of PE_k is called a *protocol entity specification* and denoted by $Pspec_k$. A set of p protocol entity specifications $\langle$ $Pspec_1$, ..., $Pspec_p$ $\rangle$ is called a *protocol specification* and denoted by $Pspec^{\langle 1,p \rangle}$. We need a protocol specification to implement the distributed system.

As an example, let us assume that there are three PE's PE_1, PE_2 and PE_3 in order to implement the service specification of Fig. 2. We also assume that an allocation of resources to these PE's has been fixed as follows. PE_1 has the gate G_1 and the registers R_3 and R_4, PE_2 has the gate G_2, and PE_3 has the registers R_1 and R_2. Note that in addition to these registers, we assume that each PE_i has another register $Rtmp_i$ to keep received values given through gates (inputs and message contents). $Rtmp_i$ can contain several values. The values can be distinguished by adding the name of the value as suffix, such as $Rtmp_1.R_3$[1]. Fig. 3 shows an example of $Pspec^{\langle 1,3 \rangle}$, which provides the service of Fig. 2, based on this allocation of resources.

According to the specification of Fig. 3, PE_1 first receives an input (input variable i_1) through G_1 and stores it to $Rtmp_1.i_1$ (on transition $t_{1.1}$). Then it sends the value of $Rtmp_1.i_1$ to PE_3 as a message (on transition $t_{1.2}$), since PE_3 needs the value of i_1 to change the value of R_2. PE_3 receives and stores the value to $Rtmp_3.i_1$ on transition $t_{1.3}$. Then it changes the value of R_2 using its own value and the value of $Rtmp_3.i_1$ on transition $t_{1.4}$, and sends a message to PE_2 on transition $t_{1.5}$. When PE_2 receives the message on transition $t_{1.6}$, PE_2 knows that it can now check the value of $\mathcal{C}(T_2)$ and execute $\mathcal{E}(T_2)$. PE_2 receives an input (input variable i_2) and stores it to $Rtmp_2.i_2$ on transition $t_{2.1}$, and sends two messages. One is to send the value of i_2 to PE_1 (on transition $t_{2.3}$) and another is to incite PE_3 to send the value of R_2 to PE_1 (on transition $t_{2.2}$). PE_1 receives these values and stores them to $Rtmp_1.i_2$ and $Rtmp_1.R_2$ on transitions $t_{2.6}$ and $t_{2.7}$, respectively. Then it changes the value of R_4 on transition $t_{2.8}$. Finally, PE_1 outputs the value of R_4 on transition $t_{3.1}$ and PE_1, PE_2 and PE_3 return to their initial states.

As exemplified 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 protocols that behave correctly. Therefore we would like to derive a protocol specification automatically, such that it provides the same service as a given service specification.

3 Synthesis Overview

A method for deriving protocol specification with an optimal allocation of resources from a given service specification is presented in this section. This method is based on a set of rules (called henceforth synthesis rules) that specify how to execute each transition $T_x = \langle \mathcal{C}(T_x), \mathcal{E}(T_x), \mathcal{S}(T_x) \rangle$ of the service specification by the corresponding PE's in the protocol specification. Furthermore, based on

[1] We can realize such a register that contains several values by using several registers. However, for simplicity of discussion, we use these registers.

these rules, it decides on an optimal allocation of resources (registers and gates) amongst different derived PE's.

3.1 Synthesis Rules

For executing a transition $T_x = \langle \mathcal{C}(T_x), \mathcal{E}(T_x), \mathcal{S}(T_x) \rangle$ of the service specification by the corresponding set of transitions $t_{x.1}, t_{x.2}, ...$ of the PE's in the protocol specification, we proceed as follows.

- The PE that has gate G_s used in $\mathcal{E}(T_x)$ (say PEstart(T_x)) checks the value of $\mathcal{C}(T_x)$ (pre-condition statement) and executes $\mathcal{E}(T_x)$ (event expression).
- After that, the PE sends messages called α-messages to the PE's which have the registers used in the arguments of $\mathcal{S}(T_x)$ (substitution statements).
- In response, these PE's send the register values to the PE's which have the registers to be updated in $\mathcal{S}(T_x)$ (PEsubst(T_x) denotes the set of those PE's) as messages called β-messages.
- The substitution statements are executed and notification messages called γ-messages are sent to those PE's which will start the execution of the next transitions.

For example, for transition T_2 of the service specification in Fig. 2, PEstart(T_2) is PE$_2$ and PEsubst(T_2) is {PE$_1$}. PE$_2$ checks the value of pre-condition statement "$keyword(i_2)$" and executes "$G_2?i_2$" on transition $t_{2.1}$. Then PE$_2$ sends an α-message "Ma_2" to PE$_3$ on transition $t_{2.2}$ since PE$_3$ has register R_2 which is used to substitute the value of R_4. PE$_2$ also sends the input value to PE$_1$ as a β-message "Mb_2" on transition $t_{2.3}$. PE$_3$ receives the α-message "Ma_2" on transition $t_{2.4}$ and sends the value of R_2 to PE$_1$ as a β-message "Mb_2" on transition $t_{2.5}$. PE$_1$ receives these two β-messages on transitions $t_{2.6}$ and $t_{2.7}$, and then executes "$R_4 \leftarrow \text{retrieve}(R_3, R_2, i_2)$" on transition $t_{2.8}$ using its own register R_3 and the received values of R_2 and i_2. The PE's which will start the execution of next transition T_3 is PE_1 itself. Therefore, PE$_1$ does not send any γ-message. Then PE$_1$ starts the execution of T_3 on transition $t_{3.1}$.

In Fig. 4, we present the details of the above rules [26], that are classified into action and message rules. Action rules specify which PE checks the pre-condition and executes the event and substitution statements of T_x. Message rules specify how the PE's exchange messages, and the contents and types of these messages.

Three types of messages are exchanged for the execution of T_x. (1) α-messages are sent by the PE that starts the execution of T_x (*i.e.* PEstart(T_x)) to inform those PE's who need to send their registers' values to other PE's, that they can go ahead and send these values. Thus, an α-message does not contain values of registers. (2) β-messages are sent in order to let each PE which executes some substitution statements of T_x (*i.e.* PE$_k$∈PEsubst(T_x)) know the timing and some values of registers' it needs for executing these statements. (3) γ-messages are sent to each PE$_m$∈PEstart($T_x \bullet\bullet$), note that $T_x \bullet\bullet$ is the set of each next transition of T_x, to let it know the timing and some values of registers it needs to start executing the next transitions (*i.e.* transitions in $T_x \bullet\bullet$).

We let $T_x = \langle \mathcal{C}(T_x), \mathcal{E}(T_x), \mathcal{S}(T_x) \rangle$ be a transition of *Sspec*.

[Action Rules]

(A_1) The PE which has the gate appearing in $\mathcal{E}(T_x)$ (denoted by G_s) checks that
(a) the value of $\mathcal{C}(T_x)$ is true,
(b) the execution of the previous transitions of T_x has been finished and
(c) an input has been given through G_s if $\mathcal{E}(T_x)$ is an input event.
Then the PE executes $\mathcal{E}(T_x)$. This PE is denoted by PEstart(T_x).

(A_2) After (A_1), the PE's which have at least one register whose value is changed in the substitution statements $\mathcal{S}(T_x)$ execute the corresponding statements in $\mathcal{S}(T_x)$. The set of these PE's is denoted by PEsubst(T_x).

[Message Rules]

($M_{\beta 1}$) Each $\text{PE}_k \in$PEsubst(T_x) must receive at least one β-message from some PE's (each called PE_j) in order to know the timing and values of registers it needs for executing its substitution statements (see ($M_{\beta 2}$)), except where PE_k=PEstart(T_x), in this case PE_k already knows the timing to start executing its substitution statements of T_x.

($M_{\beta 2}$) If $\text{PE}_k \in$PEsubst(T_x) needs the value of some register (say R_z) in order to execute its substitution statements, then PE_k must receive R_z through a β-message if R_z is not in PE_k.

($M_{\beta 3}$) Each PE_j that sends some values of registers to $\text{PE}_k \in$PEsubst(T_x) through a β-message, knows the timing to send these values by receiving an α-message from PEstart(T_x). Note, if PE_j=PEstart(T_x) then PE_j knows the timing to send these values without receiving an α-message.

(M_α) After (A_1), the only PE that can send α-messages to the PE's which need them is PEstart(T_x).

($M_{\gamma 1}$) Each $\text{PE}_m \in$PEstart($T_x \bullet\bullet$), where $T_x \bullet\bullet$ is the set of next transitions of T_x, must receive a γ-message from each $\text{PE}_k \in$PEsubst(T_x) after (A_2), except where $m = k$. This allows PE_m to know that the execution of the substitution statements of T_x had been finished.

($M_{\gamma 2}$) Each $\text{PE}_m \in$PEstart($T_x \bullet\bullet$) must receive at least one γ-message from some PE_l (where $m \neq l$) in order to know that the execution of T_x had been finished and/or to know some values of registers it needs to evaluate and execute its condition and event expression, respectively.

($M_{\gamma 3}$) Each PE_l that sends a γ-message to $\text{PE}_m \in$PEstart($T_x \bullet\bullet$) :
(a) must be in PEsubst(T_x) (see ($M_{\gamma 1}$)), or
(b) must receive an α-message from PEstart(T_x) to know the timing to send the γ-message to PE_m, or
(c) it is itself PEstart(T_x). In this case, PE_l sends the γ-message to let PE_m know the timing and/or some values of registers to start evaluating and executing its condition and event expressions.

($M_{\gamma 4}$) If $\text{PE}_m \in$PEstart($T_x \bullet\bullet$) needs the value of some register (say R_v) in order to evaluate and/or execute its substitution statements, then PE_m must receive R_v through a γ-message if R_z is not in PE_m.

Fig. 4. Derivation Method in Detail

3.2 Integer Linear Programming Model for Protocol Derivation

Based on the above synthesis rules, we determine a behavior of the derived PE's that would minimize their communication cost while optimally allocating their resources, using an Integer Linear Programming (ILP) model. This cost could be based on the number of messages to be exchanged between different PE's [25]. Moreover, other cost criteria can also be considered such as the costs of resource allocation, size of messages exchanged between different PE's, and frequencies of transition execution.

The ILP Model (for details see [26, 25]) consists of an objective function that minimizes the communication cost and decides on an optimal allocation of resources, based on a set of constraints. These constraints are based on the above synthesis rules, and they consist of 0-1 integer variables indicating (a) whether a PE should send a message or not, (b) whether a message contains a register value or not, or (c) whether a register/gate is allocated to a PE or not.

4 Protocol Re-synthesis

In this section, we present our new method for re-synthesizing the protocol specification from a modified service specification. The method consists of a set of rules that would be applied to different PE's after a modification to the service specification, in order to produce new synthesized (re-synthesized) PE's.

For each simple modification (henceforth called *atomic modification*) made on the service specification *Sspec*, we define its corresponding *atomic re-synthesis rules*. As shown later, these atomic re-synthesis rules can also be sequentially applied to deal with more than one modification. Note that the atomic re-synthesis rules are based on the synthesis rules described in Section 3. Consequently, we show next to the description of each re-synthesis rule its corresponding synthesis rule.

4.1 Atomic Modifications and Their Corresponding Re-synthesis Rules

For each of the following possible atomic modifications to *Sspec*, we present its corresponding atomic re-synthesis rules. Note that each modification to *Sspec* changes the label of a transition T_x in *Sspec* from $\langle \mathcal{E}(T_x), \mathcal{C}(T_x), \mathcal{S}(T_x)\rangle$ to $\langle \mathcal{E}'(T_x), \mathcal{C}'(T_x), \mathcal{S}'(T_x)\rangle$. For convenience, we denote the following sets of registers:

- Rev^x: the set of registers that PEstart(T_x) needs to evaluate $\mathcal{C}(T_x)$ or execute $\mathcal{E}(T_x)$
- $Rrsub_i^x$: the set of registers that are used in PE$_i$ $\in$PEsubst(T_x) to execute the statements in $\mathcal{S}(T_x)$
- $Rcsub_i^x$: the set of registers that are defined (*i.e.* referenced) by the left-hand-sides of the substitution statements in $\mathcal{S}(T_x)$ in PE$_i$ $\in$PEsubst(T_x).

[**Atomic Modifications**]

1. $Rev^x \leftarrow Rev^x \setminus \{R_h\}$
2. $Rev^x \leftarrow Rev^x \cup \{R_h\}$
3. $Rrsub_k^x \leftarrow Rrsub_k^x \setminus \{R_h\}$
4. $Rrsub_k^x \leftarrow Rrsub_k^x \cup \{R_h\}$
5. $Rcsub_k^x \leftarrow Rcsub_k^x \setminus \{R_h\}$
6. $Rcsub_k^x \leftarrow Rcsub_k^x \cup \{R_h\}$

[**Atomic Re-synthesis Rules**]

1. $Rev^x \leftarrow Rev^x \setminus \{R_h\}$:
   The following rules take into account that the value of R_h which has been sent to PEstart(T_x) is no longer necessary after the modification. These rules are applied to the part of the protocol specification where each previous transition (say T_w) of T_x is executed, if applicable.
   (a) Each PE (say PE$_l$) which sends a γ-message including the value of R_h to PEstart(T_x), should exclude the value of R_h from the γ-message (*c.f.* **synthesis rule (M$_{\gamma 4}$)**).
   (b) If (a) is done, then the γ-message can be deleted only if
      – PE$_l$ $\notin$PEsubst(T_w) (*c.f.* **synthesis rule (M$_{\gamma 1}$)**),
      – there is still at least one γ-message sent to PEstart(T_x) after deleting it (*c.f.* **synthesis rule (M$_{\gamma 2}$)**) and
      – it no longer has values (*c.f.* **synthesis rule (M$_{\gamma 4}$)**).
   (c) If (b) is done, then an α-message sent to PE$_l$ can be deleted only if PE$_l$ no longer sends β- and γ-messages (*c.f.* **synthesis rule (M$_{\gamma 3}$)(b)**).
2. $Rev^x \leftarrow Rev^x \cup \{R_h\}$:
   The following rules take into account that the value of R_h must be sent to PEstart(T_x) after the modification. These rules are applied to the part of the protocol specification where each previous transition (say T_w) of T_x is executed, if applicable.
   (a) One of the PE's which have R_h and send γ-messages to PEstart(T_x) should include the value of R_h in its γ-message to PEstart(T_x), if such a PE exists (*c.f.* **synthesis rule (M$_{\gamma 4}$)**).
   (b) Otherwise, one of the PE's which have R_h should send a new γ-message which includes the value of R_h to PEstart(T_x). If the PE does not receive α-messages and is not PEstart(T_x), PEstart(T_w) should send an α-message to the PE. (*c.f.* **synthesis rule (M$_{\gamma 3}$)**).
3. $Rrsub_k^x \leftarrow Rrsub_k^x \setminus \{R_h\}$:
   The following rules take into account that the value of R_h sent to PE$_k$ is no longer necessary after the modification. These rules are applied to the part of the protocol specification where T_x is executed.
   (a) Each PE (say PE$_j$) which sends a β-message including the value of R_h to PE$_k$ should exclude the value from the β-message (*c.f.* **synthesis rule (M$_{\beta 2}$)**).
   (b) If (a) is done, then the β-message can be deleted only if

  - there is still at least one β-message sent to PE$_k$ after deleting it (*c.f.* **synthesis rule (M$_{\beta 1}$)**) and
  - it no longer has values (*c.f.* **synthesis rule (M$_{\beta 2}$)**).

   (c) If (b) is done, the α-message sent to PE$_j$ can be deleted only if PE$_j$ no longer sends β- and γ-messages. (*c.f.* **synthesis rule (M$_{\beta 3}$)**).

4. $Rrsub_k^x \leftarrow Rrsub_k^x \cup \{R_h\}$:
   The following rules take into account that the value of R_h must be sent to PE$_k$ after the modification. These rules are applied to the part of the protocol specification where T_x is executed.
   (a) One of the PE's which have R_h and send β-messages to PE$_k$ should include the value of R_h to its β-message to PE$_k$, if such a PE exists. (*c.f.* **synthesis rule (M$_{\beta 2}$)**).
   (b) Otherwise, one of PE's which have R_h should send a new β-message which includes R_h to PE$_k$. If the PE does not receive α-messages and is not PEstart(T_x), PEstart(T_x) should send an α-message to the PE.
5. $Rcsub_k^x \leftarrow Rcsub_k^x \setminus \{R_h\}$:
   Removing a substitution statement. Usually, this may cause an additional modification $Rrsub_k^x \leftarrow Rrsub_k^x \setminus \{R_{h_1}, R_{h_2}, ..., R_{h_k}\}$, since the deleted statement uses values of registers. In this case, we consider that the atomic modification (3) was made on $Sspec$ k times and apply its corresponding atomic re-synthesis rule (3) k times.
6. $Rcsub_k^x \leftarrow Rcsub_k^x \cup \{R_h\}$:
   Adding a substitution statement. Usually, this may cause an additional modification $Rrsub_k^x \leftarrow Rrsub_k^x \cup \{R_{h_1}, R_{h_2}, ..., R_{h_k}\}$, since the added statement uses values of registers. As the case of the re-synthesis rule (5), we apply the atomic re-synthesis rule (4) k times.

4.2 Modifications to the Service Specification

In this section, we describe how modifications to $Sspec$ can be represented as the set of atomic modifications presented in the previous subsection. We consider modifications to the label of a transition T_x of $Sspec$.

- If $\mathcal{E}(T_x)$ (or $\mathcal{C}(T_x)$) is modified to $\mathcal{E}'(T_x)$ (or $\mathcal{C}'(T_x)$), then this modification can be represented as a set of the atomic modifications of type (1) and/or (2) which involve adding and/or removing registers from the set of registers Rev^x that PEstart(T_x) needs to execute $\mathcal{E}(T_x)$ (or evaluate $\mathcal{C}(T_x)$).
- If $\mathcal{S}(T_x)$ is modified to $\mathcal{S}'(T_x)$, then this modification can be represented by a sequence of atomic modifications of type (3), (4), (5) or (6), respectively.

4.3 Changing the Resource Allocation for the Protocol Specification

In some application areas, the allocation of resources between different PE's is necessary. For example, in distributed databases, adding a copy of an existing register to some PE's is necessary to increase the fault tolerance and balance the load amongst these PE's. Here we consider the case where a copy of an existing register R_h in PE$_j$ is added to another PE PE$_k$. For each transition T_x where the value of R_h is changed (defined) in the substitution statement $\mathcal{S}(T_x)$, PE$_k$ must execute this substitution statement to update the value of register R_h. Consequently, this modification can be represented by the atomic modification (6).

5 Example and 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[14], distributed cooperative work management[13] and so on.

In this section, we apply our synthesis method [26] 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 [28] 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 5 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. 5.

We regard this workflow as the service specification, and we derive its corresponding protocol specification using the method and programs used in our previous work[26], where we have developed two programs that generate for the given specification its corresponding ILP problem constraints, and derive the protocol specifications using the synthesis rules. The tool lp_solve[29] is used to solve the ILP problem and obtain the minimal number of messages to be exchanged between the derived protocol entities. It took 639 seconds on MMX-Pentium 200MHz PC to synthesize the given specification. The optimal allocation of the registers is shown in Table 1 and the minimum number of messages to be exchanged between the different PE's is 40.

5.2 Experimental Results

In this section, we show the effectiveness of our re-synthesis method by comparing the time it takes to synthesize the given service specification again after an assumed modification to the time it takes using our re-synthesis method.

We consider the following modifications to the given service specification:

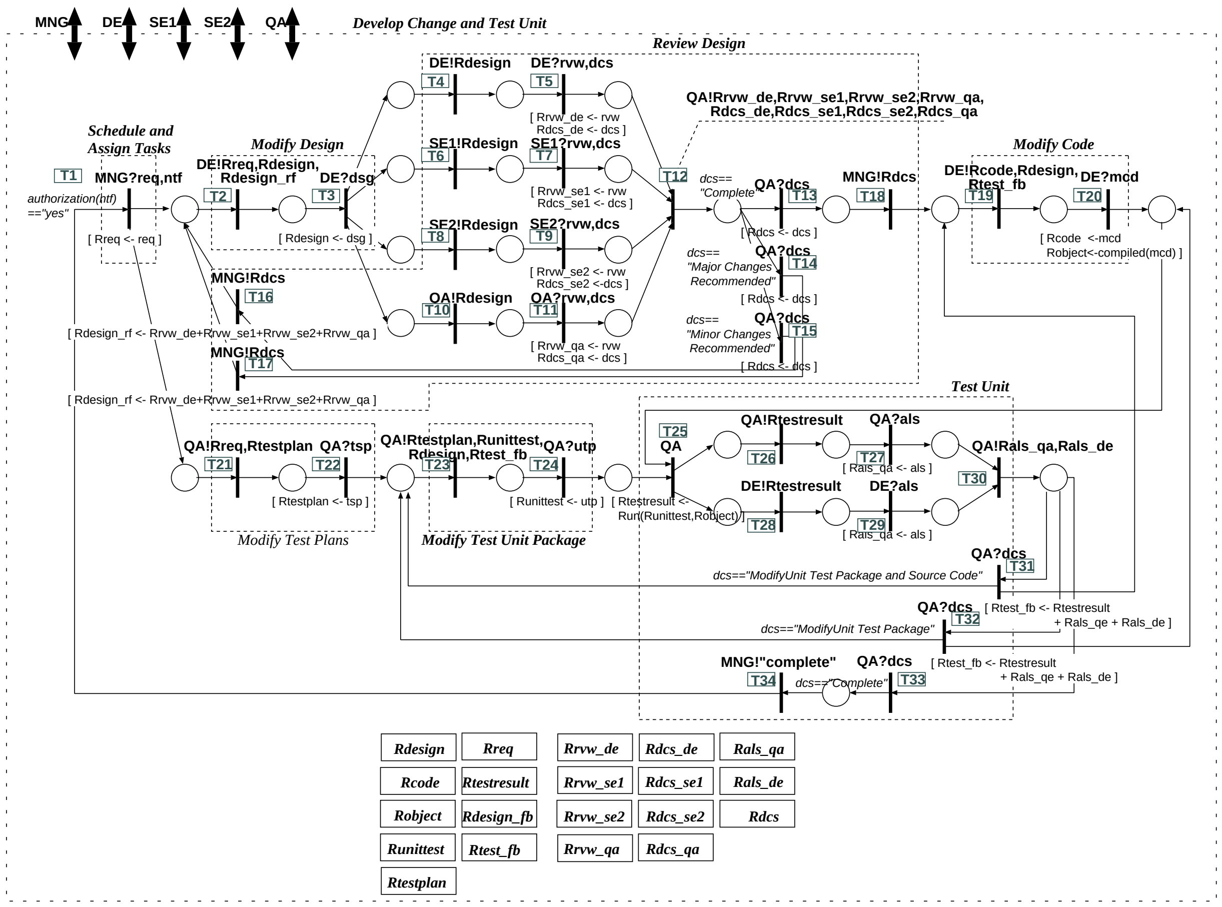

Fig. 5. Modeling the Core Problem in the ISPW-6 Example

	PE$_{mng}$	PE$_{de}$	PE$_{se1}$	PE$_{se2}$	PE$_{qa}$
Gate	MNG	DE	$SE1$	$SE2$	QA
Register		R_{req} R_{design} R_{design_fb} R_{rvw_de} R_{dcs_de} R_{code} R_{test_fb} $R_{testplan}$	R_{rvw_se1} R_{dcs_se1} $R_{unittest}$ $R_{testresult}$	R_{rvw_se2} R_{dcs_se2}	R_{rvw_qa} R_{dcs_qa} R_{dcs} R_{object} R_{alc_qa} R_{als_de}

Table 1. Optimal Allocation of Resources for Engineers' Machines

	Synthesis Time (sec.)		Number of Messages	
	Re-synthesis	Complete Synthesis	Re-synthesis	Complete Synthesis
case1	1	958	44	44
case2	1	1021	46	46
case3	1	940	40	40
case4	1	1640	42	42

`MMX-Pentium 200 MHz, 128MB Memory`

Table 2. 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 $R_{unittest\_new}$) 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}).
3. DE analyzes the test feedback (register R_{test_fb}) 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 PE$_{de}$) is placed on PE$_{mng}$.

After each modification, we have used the programs developed in [26] to measure the time (in seconds) it takes to synthesize the given 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 2 shows these times. The reader can clearly see that the re-synthesize 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 2) the re-derived protocol specifications still have optimal (or near-optimal in general cases) solutions.

6 Conclusion and Further Research

Based on our previous work on protocol synthesis of systems modeled as extended Petri nets, we have developed a set of rules that avoid complete synthesis after incremental modifications to such a system. These rules are applied to the affected parts of derived protocol specification. This would make protocol synthesis and maintenance more practical for realistic applications.

Currently, we are developing a re-synthesis method to specifications modeled as finite state machines. Moreover, we are investigating the extension of our re-synthesis method to specifications modeled as timed Petri nets.

References

1. T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. of the IEEE*, Vol. 77, No. 4, pp. 541–580, 1989.
2. R. Milner, "Communication and Concurrency," *Prentice-Hall*, 1989.
3. 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.
4. H. Erdogmus and R. Johnston, "On the Specification and Synthesis of Communicating Processes," *IEEE Trans. on Software Engineering*, Vol. SE-16, No. 12, 1990.
5. R. Probert and K. Saleh, "Synthesis of Communication Protocols: Survey and Assessment," *IEEE Trans. on Computers*, Vol. 40, No. 4, pp. 468–476, 1991.
6. K. Saleh, "Synthesis of Communication Protocols: an Annotated Bibliography," *ACM SIGCOMM Computer Communication Review*, Vol. 26, No. 5, pp. 40–59, 1996.
7. 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.
8. 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.
9. 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.
10. P. -Y. M. Chu and M. T. Liu, "Protocol Synthesis in a State-transition Model," *Proc. of COMPSAC '88*, pp. 505–512, 1988.
11. 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.
12. 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.

13. K. Yasumoto, T. Higashino and K. Taniguchi, “Software Process Description Using LOTOS and its Enaction,” Proc. of the 16th Int. Conf. on Software Engineering (ICSE-16), pp. 169-179, 1994.
14. D. Y. Chao and D. T. Wang, “A Synthesis Technique of General Petri Nets,” *Journal of System Integration*, Vol. 4, pp. 67–102, 1994.
15. H. Yamaguchi, K. Okano, T. Higashino and K. Taniguchi, “Synthesis of Protocol Entities' Specifications from Service Specifications in a Perti Net Model with Registers,” *Proc. of 15th Int. Conf. on Distributed Computing Systems (ICDCS-15)*, pp. 510–517, 1995.
16. 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.
17. 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.
18. 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.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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 the 11th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'99)*, 1999.
26. 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 the ASIC Int. Conf. on Software Engineering Applied to Networking and Parallel/ Distributed Computing (SNPD'00)*, pp. 19–26, 2000.
27. S.S. Skiena, “The ALGORITHM Design Manual,” *TELOS - The Electronic Library of Science (A Springer-Verlag Imprint)*, 1998.
28. Kellner, M. et al. : “ISPW-6 Software Process Example,” *Proc. of the 1st Int. Conf. on the Software Process*, pp. 176-186, 1991.
29. “lp_solve,” `ftp://ftp.ics.ele.tue.nl/pub/lp_solve/`
30. G. Rothermel and M. J. Harrold, “Analyzing Regression Test Selection Techniques,” *IEEE Trans. on Software Engineering*, Vol. 22, No. 8, pp. 529–551, 1996.