Available online at www.sciencedirect.com

ScienceDirect

Computer Networks 51 (2007) 258–284

Computer Networks

www.elsevier.com/locate/comnet

Deriving protocol specifications from service specifications written as Predicate/Transition-nets

Hirozumi Yamaguchi [a,*], Khaled El-Fakih [b,1], Gregor v. Bochmann [c], Teruo Higashino [a]

[a] *Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyamacho, Toyonaka, Osaka 560-8531, Japan*

[b] *School of Engineering, American University of Sharjah, Sharjah, P.O. Box 26666, United Arab Emirates*

[c] *School of Information Technology and Engineering, University of Ottawa, 800 King Edward Avenue, Ottawa, Ont., Canada K1N 6N5*

Received 16 June 2005; received in revised form 19 December 2005; accepted 16 March 2006
Available online 5 June 2006

Responsible Editor: Juan Quemada

Abstract

We consider the derivation of a protocol specification from a service specification written in Predicate/Transition-nets (Pr/T-nets). The service specification describes the global behavior of a system and includes the allocation of the Pr/T-net places to N distributed sites. The paper presents a new algorithm for deriving a protocol specification that defines the behavior of N communicating entities that execute on the N sites and coordinate their actions in order to conform to the global behavior defined by the service specification. Our algorithm decomposes each transition of the service specification into a set of communicating Pr/T-subnets running on the N entities. Moreover, for efficiently controlling the conflict for shared resources, we present a timestamp-based contention control algorithm and incorporate it into the derivation algorithm. A tool has been developed that implements our algorithm and works together with other existing tools for the graphical representation of the service and derived protocol specifications. Two application examples are discussed.

Keywords: Petri net; Distributed system; Specification; Automated design

1. Introduction

Synthesis methods have been used to derive the specification of a set of application components running in a distributed system of networked computers (hereafter called *protocol specification*) automatically from a given specification of services to be provided by the distributed application to its users (called *service specification*). The service

[*] Corresponding author. Tel.: +81 6 6850 6607; fax: +81 6 6850 6609.

E-mail addresses: h-yamagu@ist.osaka-u.ac.jp (H. Yamaguchi), kelfakih@aus.edu (K. El-Fakih), bochmann@site.uottawa.ca (G.v. Bochmann), higashino@ist.osaka-u.ac.jp (T. Higashino).

[1] Tel.: +971 06 5152556; fax: +971 06 5152979.

specification is written in the form of a centralized model, and does not contain any message exchanges between different physical locations. However, the definition of the behavior of the application components, called *protocol entities* (*PEs*), includes the message exchanges between these entities. Protocol synthesis methods have been used to specify and derive such complex message exchanges automatically in order to reduce the design costs and errors that may occur when manual methods are used.

Many synthesis methods have been proposed in the literature. The methods use different computational models as service definition languages. For example, the methods presented in [1–3] use CCS/LOTOS models, the methods in [4–8] use FSM/EFSM models and the methods in [9–16] use Petri net models. Similar methods may also be used for deriving distributed testers for distributed applications [17] and for deriving specifications of real-time systems [18–20]. In this paper we consider service and protocol specifications written in high-level Petri nets. These are extended Petri nets where tokens have values and the firability of transitions may depend on those values. Popular versions of high-level Petri nets are predicate/transition nets (Pr/T-nets) [21,22] and coloured Petri nets (CPN) [23]. These models have enough modeling power, analytical power and tool support (such as CPN Tools [24]) to specify, verify and analyze large and practical software systems [25], communication protocols [26,27], control systems and so on [23,28].

In this paper, we propose a new algorithm for the derivation of a protocol specification in Pr/T-nets, which is the specification of N communicating entities (N is given), from a given service specification in Pr/T-nets and an allocation of the places of the service specification to the N entities. Our algorithm decomposes each transition of the service specification into a set of communicating Pr/T-subnets running on the N entities. Moreover, in order to improve the efficiency of controlling the conflict between different transitions over shared resources, we present a timestamp-based contention control algorithm and incorporate it into the derivation algorithm. A tool has been developed that includes our derivation algorithm and works together with other existing tools for the representation of the service and the derived protocol specifications. As application examples we discuss the application of our synthesis method to a distributed media transcoding service on overlay networks and to a distributed software development process [29].

Our approach is very powerful in the sense that general Pr/T-nets are allowed to be used for specifying services. Such Pr/T-nets may include complex conflict structures between transitions that require read and/or write access to shared resources in the form of tokens with values stored at shared places. Since these resources may reside on different sites and the transitions should be initiated when all required resources are available, we have to deal with this complex problem of distributed synchronization for the different transitions involving the protocol entities on the different sites. We first present a basic transition execution protocol where a transition is initiated by its "primary site" without having full knowledge about the available resources; the transition is then canceled whenever there appears to be some conflict or deadlock possibility.

Some existing synthesis methods also allow to treat variables (parameters) in their modeling languages as for instance a CCS-based model with I/O parameters [1] and Petri nets with external variables [9,15]. However, since these existing methods mainly focus on value exchanges between entities, only simple control flows are allowed; the combination of choices and synchronization involving parameters, which often represents resource conflict, is not treated by those methods. Therefore, the class of acceptable service specifications has been considerably extended by the approach described in this paper. As far as we know, no previous paper has presented synthesis approaches for general Pr/T-nets.

We note that the basic idea of this paper was presented in [30]. Here we extend that work in several ways. First, we enhance the derivation algorithm by including a new derivation policy. Using a few additional messages, this policy prevents large size resources from being exchanged between entities. Second, we include a detailed timestamp-based contention control algorithm for efficiently controlling conflict for shared resources. Third, we developed a tool that includes our derivation algorithm and can interwork with other Petri net tools. Fourth, we provide arguments for the validity of our method and discuss the application of the method to two realistic examples.

This paper is organized as follows. Section 2 includes the definition of Pr/T-nets and provides examples of service and protocol specifications written in this notation. In Section 3 we present our derivation algorithm and in Section 4 we enhance this algorithm by incorporating a timestamp-based contention control algorithm. In Section 5 we describe two application examples and in Section 6 we conclude the paper.

2. Service and protocol specifications in Pr/T-nets

2.1. Predicate/transition-nets

We use Predicate/Transition-nets (Pr/T-nets) [21] for representing service and protocol specifications of target systems. In Petri nets, a place (denoted as a circle) represents a state or data of a system, and a transition (denoted as a rectangle) represents a task (or job) of the system. A place and a transition may be connected by a directed edge called an arc (denoted by an arrow). Tokens (denoted as black dots) in places represent the current state of the system, and execution ("firing" in the Petri net terminology) of a transition may consume/produce tokens from/to the places connected to the transition.

Pr/T-nets are an extended form of Petri nets. Intuitively, in Pr/T-nets, each incoming arc to a transition t from a place p has a label of the form of $k_1X_1k_2X_2\ldots$ called an *arc label* where k_i is a positive integer, X_i is a n-tuple of variables like $\langle x_1, x_2, \ldots, x_n\rangle$ and n is an arbitrary non-negative integer assigned to place p. Place p may have tokens, each of which is a n-tuple of values $C_i = \langle c_1, c_2, \ldots, c_n\rangle$. A set of tokens which can be assigned to an incoming arc to transition t is called an *assignable set* of the arc. Moreover, a transition t may be associated with a logical formula of variables from the labels of incoming arcs of t, called a *condition*. Conditions are depicted inside transitions rectangles. A transition t may fire *iff* there exists an assignable set in each input place of t and the assignment of values to variables by the assignable set satisfies the condition of t. Also, each outgoing arc from transition t to a place p' has a label of the form of $k'_1Y_1k'_2Y_2\ldots$ where k'_i is a positive integer and Y_i is a n'-tuple of values, variables on the incoming arc labels of t or functions over the variables. Therefore, if t fires, the values of the labels on the outgoing arcs from t are determined by the assigned input tokens according to the output arc labels. New sets of tokens are generated and put into the output places of t.

Fig. 1 includes an example of Pr/T-net. In Fig. 1(a), the incoming arc to t from p_1, (p_1, t), has the label $2\langle x, y\rangle$ where x and y are variables. This means that two tokens each consisting of a pair of values are necessary in place p_1 for the firing of transition t. Here, since the following assignable sets $2\langle$"a", "c"$\rangle$ in p_1 ("*a*" and "*c*" are strings here), $\langle$"a"$\rangle$ and $\langle$"c"$\rangle$ in p_2 and two tokens without values in p_3 satisfy the condition of t, $(x = z \wedge y = w)$, t can fire using these sets. Note that tokens without values are represented as black dots in the following figures. After the firing of t, new tokens are generated to the output places p_4 and p_5 using those token values. The marking after the firing of t is shown in Fig. 1(b). Note that "@" is a concatenation function of two strings. Thus a tuple of strings "*aa*", "*c*" and "*a*" is generated to p_4. The arc label "1", which means the delivery of one token without values, is omitted in the following figures.

In the following we formally define the Pr/T-nets model. For related detailed definition, the reader may refer to [21,22].

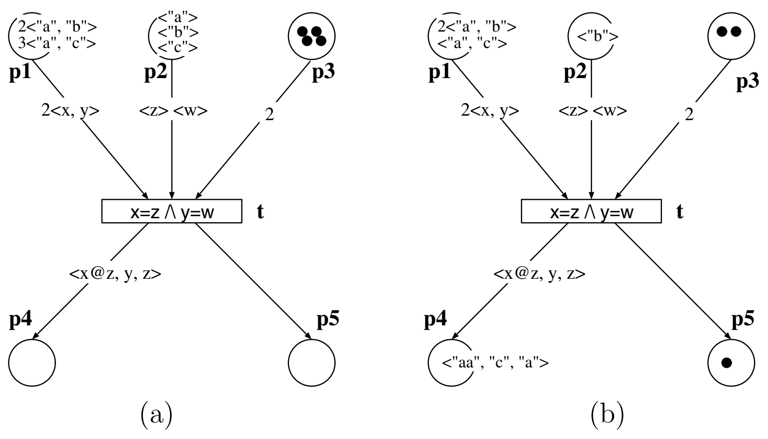


Fig. 1. An example of Pr/T-nets: (a) before firing of t, (b) after firing of t.

Definition 1. $N = (P, T, F, U, L, C, m_0)$ is called a Predicate/Transition-net (Pr/T-net) iff

1. (P, T, F) is a Petri net where P, T and $F \subseteq (P \times T) \cup (T \times P)$ are sets of places, transitions and arcs, respectively.
2. U is a finite set of values, variables and operators over the values and variables, for example, $U = \{$"a", "b",...; $x, y, \ldots; >, =, \vee, \ldots\}$.
3. L is a set of arc labeling functions. Each function assigns an arc label to either an incoming arc (p, t) to a transition t or an outgoing arc (t', p') from a transition t'. The arc label assigned to (p, t) has a form like $k_1X_1 \ldots k_mX_m$ $(m \geqslant 0)$ where k_i is a positive integer and X_i is a n-tuple of variables. The arc label assigned to (t', p') has a form like $k'_1Y_1 \ldots k'_{m'}Y_{m'}$ $(m' \geqslant 0)$ where k'_i is a positive integer and Y_i is a n'-tuple of values, the variables on the labels of incoming arcs to t' and functions in U over the variables.
4. C is a set of transition labeling functions. Each function assigns to a transition t a logical formula over the values and variables in U. This is called a *condition*. Variables in a condition of transition t are from the variables on the labels of incoming arcs to t.
5. m_0 is the initial marking of N which assigns to each place p n-tuples of values. Each *tuple* of values is called a *token*.

Hereafter $\bullet t$ and $t\bullet$ denote the sets of input and output places of t, respectively. A transition t may fire at a marking m *iff* for each place $p \in \bullet t$ there exists an assignment β_p that assigns to $L(p, t)$ (the label of arc (p, t)) a subset of $m(p)$ (tokens in place p) and those assignments $\cup_{p\in\bullet t}\beta_p$ make the value of $C(t)$ be true. If t fires, for each output place $p' \in t\bullet$, the set of tokens (the value of $L(t, p')$ determined by the assignments $\cup_{p\in\bullet t}\beta_p$) is generated to p'. If the labels of more than one incoming arc of t contain variables with the identical names, the values assigned by β_p to these variables must be equal.

2.2. Service specification

A service specification is a description of services to be provided to the service users of a distributed system. Fig. 2(a) shows an example service specification. For readability, we use a very simple example. The system works as follows. At the initial marking, transition t_u can fire, since there exists an assignable set in each input place of t_u and these assignable sets satisfy the condition of t_u. For example, ⟨"a"⟩ in p_1, ⟨"b"⟩ in p_2, ⟨"c"⟩ in p_3 and ⟨"d"⟩ in p_4 are such assignable sets that satisfy the condition, since character values "a", "b", "c" and "d" are assigned to variables x, y, z and w, respectively, and the condition "$x < z$" under those assignments becomes "a" < "c" (this is true in the alphabetical order). Let us assume that these tokens are used for the firing of t_u. If t_u fires, these tokens are removed and three tokens ⟨"a"⟩, ⟨"b"⟩ and ⟨"c"⟩ are generated to the output place p_4, and a new token ⟨"a", "b","d"⟩ is generated to p_5.

2.3. Protocol specification

A protocol specification is a lower level specification of the distributed system that consists of N entities (distributed components) communicating with each other. These entities are called *sites* in this paper. In distributed systems, computer resources (such as databases), which are usually represented as places with tokens in Pr/T-nets, are usually distributed over multiple sites. In describing a service specification, developers are not required to be aware of the location of the places. However, in a protocol specification, these sites need to cooperatively collect/distribute tokens from/to these places to execute the transition. Thus, a protocol specification is a set of specifications of N sites and contains communicating behavior between the sites.

Fig. 2(b) shows a protocol specification, which is a distributed specification of the service in Fig. 2(a) over three sites. In protocol specifications, we introduce places called *communication places* for modeling asynchronous and reliable communication channels, represented by dotted circles. They are like "fusion places" in coloured Petri nets [23]. We assume that two communication places with a common name "$X_{u.ij}$" where $X = \alpha$, β or γ (explained in the next section) in the Pr/T-nets of two different sites i and j represent the end points (send and receive buffers) of a reliable communication channel from site i to site j. If a token is put on "$X_{u.ij}$" at site i, the token is eventually removed and put onto "$X_{u.ij}$" at site j. Note that u means that these communication places are used with respect to the execution of transition t_u of the service specification. In the following figures, communication places are represented as dotted circles with their names inside. Arcs without labels carry tokens without values. Also, we use "!" and "!=" to represent logical negation "$\neg$" and the operator "$\neq$", respectively.

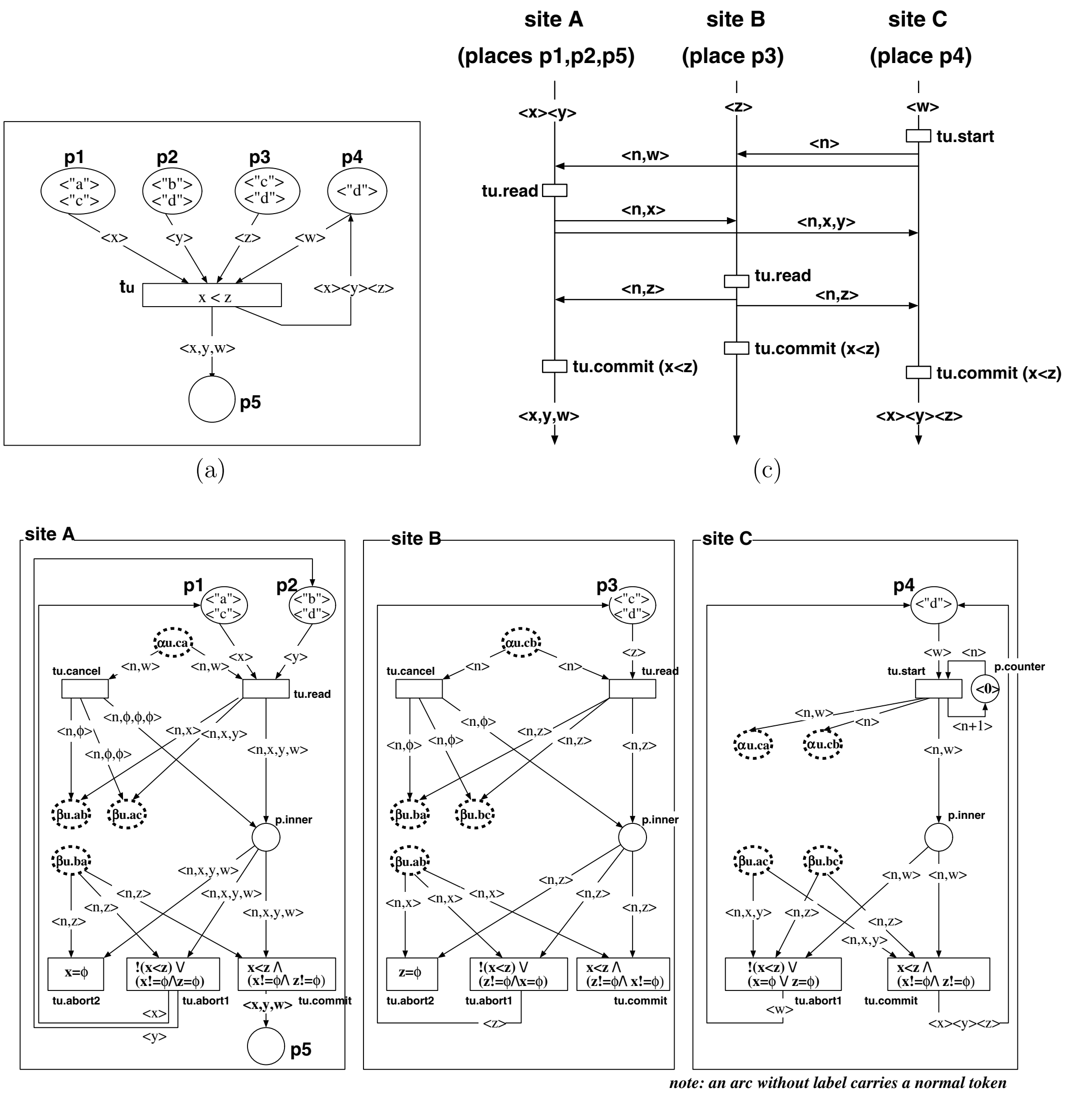


Fig. 2. (a) Service specification, (b) protocol specification, (c) timing chart.

Moreover, in protocol specifications we introduce a reserved symbol denoted by ϕ, used in tokens for notification purpose only.

In our derivation algorithm, we assume that an allocation of the places of the service specification to sites is given. In the protocol specification of Fig. 2(b), places p_1, p_2 and p_5 are located to site A, p_3 to site B, and p_4 to site C. Under this allocation, our derivation algorithm determines how the values in these places and other notification messages are exchanged to simulate the behavior of the service specification.

In the derivation algorithm, one of the sites that have input places of t_u starts the execution of t_u. In Fig. 2(b), this is site C. It starts the execution by firing the "start" transition "$t_u.start$", which sends an assignable set to site A carrying the values $\langle n, w\rangle$ via communication place "$\alpha_{u.ca}$" and the value $\langle n\rangle$ via communication place "$\alpha_{u.cb}$" to site B. We note that the variable n carries a non-negative integer and is used to identify each execution of transition t since several executions of t may fire simultaneously with different assignable sets. For this purpose, we introduce a place called *p.counter*

Table 1
Semantics of transitions in protocol specifications

Name	Semantics
t.start	Initiates the execution of transition *t* by taking an assignable set of tokens from each input place of *t* at the site, and sends them to the other sites
t.read	(Following *t.start*,) takes an assignable set of tokens from each input place of *t* at the site and sends them to the other sites
t.cancel	(Following *t.start*,) is executed if assignable sets of tokens are not available for all input places at the site. It sends tokens with symbol ϕ (cancellation tokens) to the other sites to let them know that an input place has no assignable set
t.commit	Commits the execution of *t*
*t.abort*1	Aborts the execution of *t* due to the lack of assignable sets of tokens on the other sites or the condition of *t*. The assignable sets of tokens taken from the input places are returned
*t.abort*2	Aborts the execution of *t* due to the lack of tokens on the site itself

attached to t_u.*start* that generates a unique integer for each firing of t_u.*start*. Thus the variable *n* is attached to all tuples of values used in the execution of *t*. If site *A* has assignable sets in the input places p_1 and p_2 and if it receives the value $\langle n, w\rangle$ from site *C*, it fires the "read" transition "t_u.*read*", which sends the values $\langle n, x\rangle$ to site *B* and the values $\langle n, x, y\rangle$ to site *C* via communication places "$\beta_{u.ab}$" and "$\beta_{u.ac}$", respectively. If either p_1 or p_2 does not have an assignable set, the "cancel" transition "t_u.*cancel*" will eventually fire on site *A* and will send cancellation tokens (tokens with ϕ) to sites *B* and *C* via communication places "$\beta_{u.ab}$" and "$\beta_{u.ac}$", respectively. The firing of the cancel transition at site *A* means that the execution of t_u will be aborted due to the lack of assignable sets in input places p_1 and p_2 of t_u.[2] Similarly, site *B* takes an assignable set from p_3 and sends the values $\langle n, z\rangle$ to both sites *A* and *C*. Consequently, every site can examine (i) whether all the input places of t_u have assignable sets or not, and (ii) whether the assignable sets satisfy the condition of t_u or not. If (i) and (ii) are true, the "commit" transition "t_u.*commit*" on each site will eventually fire and new tokens are generated on sites *A* and *C* for the output places p_5 and p_4, respectively (i.e. the execution of t_u has been committed). If either (i) or (ii) does not hold, the "abort" transition t_u.*abort*1 or t_u.*abort*2 on each site fires. t_u.*abort*1 fires in case that an input place at an other site does not have an assignable set or (ii) does not hold, and t_u.*abort*2 fires in case that an input place at the site itself does have an assignable set. If t_u.*abort*1 fires, the tokens read from the input places are returned to the input places, *i.e.*, the execution of t_u is aborted. These transitions are listed in Table 1. A possible time sequence diagram is shown in Fig. 2(c).

3. Derivation algorithm

3.1. Overview

Given a service specification *Sspec* written in the form of a Pr/T-net, the number *N* of sites, and an allocation of each place of the service specification to one of the *N* sites, our derivation algorithm derives a protocol specification *Pspec*, which consists of a set of specifications for the *N* sites. The derivation algorithm is presented in Section 3.3, and in Appendix B we comment on its validity.

The derivation of the protocol specification proceeds for each transition of the service specification, independently of the other service transitions. For each given transition *t* of the service specification, the derivation algorithm creates a Transition Execution (TE) protocol which is explained in the next subsection. In fact, an example of the TE protocol was already discussed in Section 2.3 for the example of Fig. 2. The TE protocol for a given service transition consists of a partial Petri-net behavior for each site *i*. For the protocol specification, the behavior specification for site *i* is obtained by putting together the partial Petri-net behaviors (for site *i*) obtained from all the transitions of the service specification. The resulting structure of the behavior specification for site *i* is similar to the structure that links the different transitions in the service specification.

3.2. Principle of the transition execution protocol

Depending on a given allocation of places, for each transition *t* of *Sspec*, we identify the set of sites called *reading sites* which have at least one input place of *t*, and also the set of sites called *writing sites* which have at least one output place of *t*. Then we select one of the reading sites as the *primary site*. This is summarized in Table 2.

[2] In the Petri net formalism, t_u.*cancel* may fire even when t_u.*read* can fire. In a practical aspect, it can be easily avoided by prioritizing the firing of t_u.*read* over that of t_u.*cancel*.

Table 2
The role of sites in transition execution protocol

Reading site	is a site which has at least one input place of t
Primary site	is a site selected from the reading sites of t. It starts the execution of this transition
Writing site	is a site which has at least one output place of t

In the following we describe the TE protocol:

1. The primary site (say site i) starts the execution of t by taking assignable sets from the input places of t allocated to site i. Then to the other reading and writing sites, it sends tokens (carried via α communication places) with the values included in the assignable sets. These values are used to examine the condition of t or for generating new tokens. Thus some tokens may include no value if a reading or writing site does not need these values.
2. When a reading site (say site j) receives token(s) from the primary site, site j selects an assignable set from each input place of t allocated to site j if such a set exists. Then to the other reading and writing sites, site j sends tokens (carried via β communication places) with the values included in the assignable sets. These values are used to examine the condition of t or for generating new tokens. Thus some tokens may include no value if a reading or writing site does not need these values. If an assignable set does not exist, site j sends tokens with null values ϕ, called "cancellation tokens". The reading and writing sites examine (i) whether all the input places of t have assignable sets or not, and (ii) whether the assignable sets satisfy the condition of t. The first condition can be checked by checking if the received tokens include ϕ or not, and the second condition can be checked by using the included values in the received tokens. If the conditions (i) and (ii) are true, the reading sites discard the assignable sets and the writing sites generate new tokens to the output places of t allocated to them. Otherwise, the execution of t is aborted. In this case, the assignable sets which have been acquired by the reading sites are returned to the original input places.

In order to prevent deadlocks due to waiting for tokens where an assignable set does not exist in an input place of a transition, we have introduced a mechanism to cancel and abort the execution of the transition when one of the input places has no assignable set. For performance reasons, it is clear that one would like to avoid the cancellation of a transition as much as possible. In the case of free-choice Petri nets, the choice between alternatives can be performed by a single place which is an input place to all the alternative transitions. In this case, we could choose as primary site of all those transitions the site to which that place is allocated. In this case the cancel transition in the protocol specification does not need to be implemented, thus simplifying the protocol specification and avoiding transaction cancellation. For the case of general Petri nets, a distributed contention control algorithm for partly avoiding transition cancellations is described in Section 4.

3.3. Derivation algorithm in detail

Here we present the protocol derivation algorithm which is based on the TE protocol described in Section 3.2. Hereafter, for a set P of places, let $ALC_i(P)$ denote the set of places in P allocated to site i. We note that $\cup_k ALC_k(P) = P$ and $\forall i, j$ $ALC_i(P) \cap ALC_j(P) = \emptyset$. For the set P of places of *Sspec* and for each site i, the set $ALC_i(P)$ of places allocated to site i is given.

For a given transition t of the service specification, let $RS(t)$ and $WS(t)$ denote the sets of reading sites and writing sites of transition t, respectively. They are uniquely determined by the given allocation. Let $ps(t)$ denote the primary site of t. We may select any reading site as the primary site. Also, let $Vin_i(t)$ denote the set of variables each of which is used in the label of an arc (p, t) $(p \in ALC_i(\bullet t))$, $Vout_i(t)$ denote the set of variables each of which is used in the label of an arc (t, p') $(p' \in ALC_i(t\bullet))$, and $Vcond(t)$ denote the set of variables used in the condition of t. We note that, for simplicity, in our derivation algorithm, for each transition t and any pair of two reading sites i and j of t we assume $Vin_i(t) \cap Vin_j(t) = \emptyset$. If this is not true, we may rename variables with identical names differently and add a condition to t that indicates that the values of these renamed variables must be equal to satisfy the assumption. We also assume that the service specification does not have tokens without any values. If this is not true, we may give a dummy value to such tokens to satisfy this assumption. Therefore, these assumptions are not essential. A summary of all notations used in the algorithm is given in Table 3.

Table 3
Notations used in derivation algorithm

$ALC_i(P)$	the set of places contained in a given set P of places and allocated to site i
$ps(t)$	the primary site of t
$RS(t)$	the set of reading sites of t, including the primary site
$WS(t)$	the set of writing sites of t
$Vin_i(t)$	the set of variables in the label of an arc (p, t) where $p \in ALC_i(\bullet t)$
$Vout_i(t)$	the set of variables in the label of an arc (t, p') where $p' \in ALC_i(t\bullet)$
$Vcond(t)$	the set of variables in the condition of t

For demonstrating the different steps of the algorithm, we use *Sspec* in Fig. 2(a) (it is shown in Fig. 3(a) again). Let us assume that there are three sites and places p_1, p_2 and p_5 are allocated to site A, place p_3 is allocated to site B, and place p_4 to site C. This allocation of places is shown at the name of each place in Fig. 3(a). Under this allocation, $RS(t) = \{A, B, C\}$ and $WS(t) = \{A, C\}$. We have chosen $ps(t) = C$.

[**The Derivation Algorithm**]

For simplicity of notations, we let i denote $ps(t)$ in this algorithm description.

3.3.1. Step A: decompose transitions

Based on the given allocation of places to sites, this step (Step A) transforms every transition t of *Sspec* into a set of distributed transitions. As a result, this step builds a basic structure of *Pspec*, whose token values and transition conditions are determined later in Steps C and D. *t.read* (or *t.start*) transitions are introduced to take assignable sets from the input places of t at the reading sites as well as to receive assignable sets at the reading and writing sites sent from the primary site. *t.commit* transitions are introduced to check the condition of t and generate tokens to the output places of t at the reading and writing sites. Also, by adding certain other places, it is guaranteed that (1) *t.start* transition is executed first, (2) each *t.read* transition is executed after the *t.start* transition, and (3) each *t.commit* transition is executed after all *t.read* transitions at the reading sites. The result of applying Step A to our example is shown in Fig. 3(b).

For a formal description of the algorithm, we introduce the net transformation rules shown in Fig. 4(a), (b) and (c). Rule 1 splits a synchronization transition into m ($m > 1$) independent transitions where each input or output place is attached to one of these transitions. As a result, this operation removes synchronization. Rule 2 inserts a new transition and a new place before or after a transition. Rule 3 inserts a place to create an execution order between two independent transitions.

(A-1) Decompose *t* into *t.commit* transitions: For each reading or writing site k, this step generates $t.commit_k$ which has the input and output places of t allocated to site k. Formally,
- apply Rule 1 to divide each transition t of *Sspec* into a set of $t.commit_k$ transitions such that for each $k \in RS(t) \cup WS(t)$ a $t.commit_k$ transition is created for site k. Then attach the places in $ALC_k(\bullet t) \cup ALC_k(t\bullet)$ to $t.commit_k$.

(A-2) Add *t.start* and *t.read* transitions and *p.inner* places: For each $t.commit_k$, this step inserts a transition $t.read_k$ (or *t.start*) and a place $p.inner_k$. Formally,
- for each $t.commit_k$ ($k \neq i$), apply Rule 2 to insert a transition $t.read_k$ and a place $p.inner_k$ before $t.commit_k$, and
- for $t.commit_i$, apply Rule 2 to insert a transition *t.start* and a place $p.inner_i$ before $t.commit_i$.

We note that *t.read* transitions at writing sites are needed to receive tokens via α-communication places introduced in step (A-3).

(A-3) Add communication places: This step introduces communication places between the above generated transitions. Formally,
- for each pair of *t.start* and $t.read_j$, apply Rule 3 to insert a communication place α_{ij}[3]
- for each pair of $t.read_j$ and $t.commit_k$ ($j \in RS(t)$, $j \neq i$ and $j \neq k$), apply Rule 3 to insert a communication place β_{jk}.

3.3.2. Step B: introduce cancellation and identification mechanisms

The second step (Step B) adds cancellation and identification mechanism for the consistent execution of distributed transitions. *t.cancel* transitions are introduced to check the availability of assignable set in the input places of t and *t.abort*1 and *t.abort*2 transitions are introduced to actually abort the execution of t. A *p.counter* place is introduced to

[3] As explained in Section 2.3, formally we denote a communication place by $X_{u.ij}$ ($X = \alpha$, β or γ) where u is the index of the target transition t_u. However, the target transition is denoted as t in this algorithm description. Thus we use the simplified notation X_{ij}.

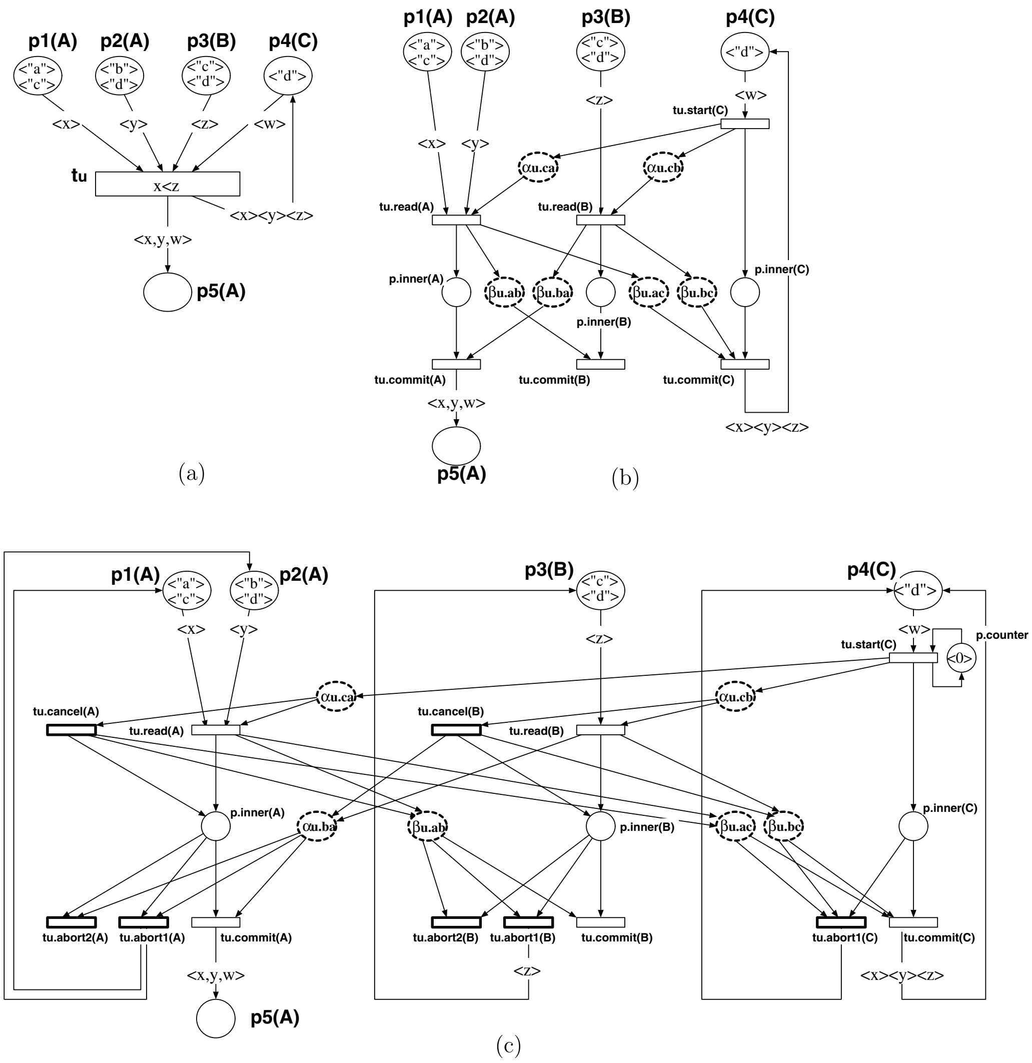


Fig. 3. Derivation algorithm snapshots.

distinguish simultaneous executions of t. The result of applying Step B to our example is shown in Fig. 3(c).

For the formal description of Step B, we use the net transformation rules shown in Fig. 4(d), (e) and (f). Rule 4 adds a transition that removes tokens from places and generates tokens to different places. Rule 5 adds a sink transition to discard tokens in places. Rule 6 adds a place to a transition to form a self-loop.

(B-1) **Add *t.cancel* transitions**: For each $t.read_j$ ($j \in RS(t)$ and $j \neq i$), a transition $t.cancel_j$ is introduced to generate cancellation tokens when some input places at site j have no assignable set. Formally,

– for each $t.read_j$ ($j \in RS(t)$ and $j \neq i$), apply Rule 4 to add a transition $t.cancel_j$ where $\bullet t.cancel_j = \{\alpha_{ij}\}$ and $t.cancel_j\bullet = t.read_j\bullet$.

(B-2) **Add *t.abort1* transitions**: For each $t.commit_k$, a transition $t.abort1_k$ is introduced to abort the execution of t, and to return the tokens to the original input places at site k if site k is a reading site. Formally,

– for each $t.commit_k$, apply Rule 4 to add a transition $t.abort1_k$ where $\bullet t.abort1_k = \bullet t.commit_k$ and $t.abort1_k\bullet = ALC_k(\bullet t)$.

(B-3) **Add *t.abort2* transitions**: For each $t.commit_k$ ($k \in RS(t)$ and $k \neq i$), a transition $t.abort2_k$ is introduced to abort the execution of t. Unlike $t.abort1_k$, $t.abort2_k$ fires if no assign-

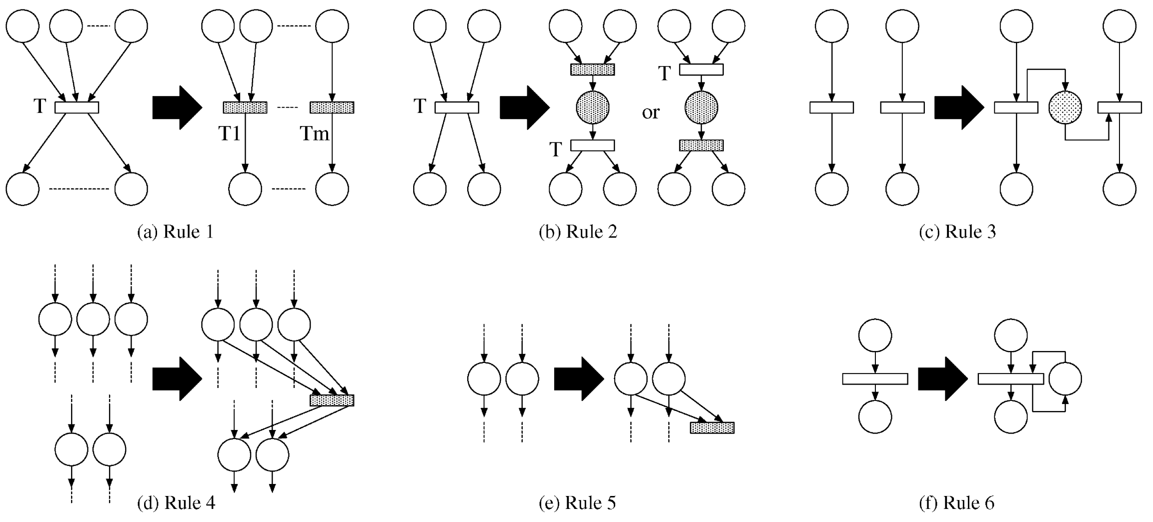


Fig. 4. Transformation rules used in steps A and B of derivation algorithm.

able set is taken from the places at site k (thus site k does not need to return the assignable set). Formally,

– for each $t.commit_k$ ($k \in RS(t)$ and $k \neq i$), apply Rule 5 to add a transition $t.abort2_k$ where $\bullet t.abort2_k = \bullet t.commit_k$.

(B-4) **Add *p.counter* place**: This place is introduced to distinguish concurrent executions of transition t with different assignable sets. Formally,

– for $t.start$, apply Rule 6 to add a place $p.counter$ with a token $\langle 0\rangle$ inside.

3.3.3. Step C: set arc labels

This step sets the arc labels attached to α and β communication places, *p.counter* place and *p.inner* places. This step determines which types of values are exchanged between sites.

(C-1) **Set arc labels of *p.counter* place**: *p.counter* is used to set a unique identifier to every execution of t. For this purpose, in Step (B-4), *p.counter* is assigned a token with an integer, which is initially zero. This value increases for each execution of $t.start$. Therefore, $\langle n\rangle$ is set as the label of ($p.counter$, $t.start$) and $\langle n+1\rangle$ is set as the label of ($t.start$, $p.counter$).

(C-2) **Determine values carried by α communication places**: This step lets each α_{ij} carry the values in the assignable set obtained at site i to site j that needs these values to examine the condition of the transition or to generate tokens for the output places of t. Formally,

– for each α_{ij}, set a tuple of the variable n and all the variables in $Vin_i(t) \cap (Vout_j(t) \cup Vcond(t)) \setminus Vin_j(t)$ as the labels of the arcs ($t.start$, α_{ij}), (α_{ij}, $t.read_j$) and (α_{ij}, $t.cancel_j$).

(C-3) **Determine values carried by β communication places**: This step lets each β_{jk} carry the values in the assignable set obtained at site j, to site k which needs the values to examine the condition or to generate tokens for the output places of t. We note that β_{jk} carries a cancellation token to site k in case that $t.cancel_j$ fires at site j. Formally,

– for each β_{jk}, set a tuple of the variable n and the variables in $Vin_j(t) \cap (Vout_k(t) \cup Vcond(t)) \setminus Vin_k(t)$ as the labels of the arcs ($t.read_j$, β_{jk}), (β_{jk}, $t.commit_k$), (β_{jk}, $t.abort1_k$) and (β_{jk}, $t.abort2_k$), and

– set a tuple of the variable n and m ϕ's as the label of ($t.cancel_j$, β_{jk}) where $m = |Vin_j(t) \cap (Vout_k(t) \cup Vcond(t)) \setminus Vin_k(t)|$.

(C-4) **Determine values kept by *p.inner* places**: This step lets each $p.inner_j$ keep the values in the assignable set obtained at site j itself and the values received through α_{ij}. We note that $p.inner_j$ keeps a cancellation token in case that $t.cancel_j$ fires at site j. Formally,

- for each $p.inner_j$ $(j \neq i)$, set the tuple of the variable n and all the variables in $Vin_i(t) \cap (Vout_j(t) \cup Vcond(t)) \cup Vin_j(t)$ as the labels of the arcs $(t.read_j, p.inner_j)$, $(p.inner_j, t.commit_j)$, $(p.inner_j, t.abort1_j)$ and $(p.inner_j, t.abort2_j)$,
- for $p.inner_i$, set a tuple of the variable n and all the variables in $Vin_i(t)$ as the labels of the arcs $(t.start, p.inner_i)$, $(p.inner_i, t.commit_i)$ and $(p.inner_i, t.abort1_i)$, and
- set a tuple of the variable n and m ϕ's as the label of the arc $(t.cancel_j, p.inner_j)$ where $m = |Vin_i(t) \cap (Vout_j(t) \cup Vcond(t)) \cup Vin_j(t)|$.

(C-5) **Determine values returned by *t.abort1* transitions**: This step determines when $t.abort1_k$ is executed the values to be returned by $t.abort1_k$ to the input places of t allocated to site k. Formally,
- for each $(t.abort1_k, p)$ $(p \in ALC_k(\bullet t))$, set its label as that of $(p, t.read_k)$ (or $(p, t.start)$).

3.3.4. Step D: set conditions

This step determines the conditions of *t.commit*, *t.abort*1 and *t.abort*2 transitions so that they can check the executability of transition t.

(D-1) **Set the conditions of *t.commit* transitions**: This step determines the condition of each $t.commit_k$ so that it fires only if all the input places of t have assignable sets and they satisfy the condition of t. Formally,
- for each $t.commit_k$ $(k \neq i)$, set the predicate $C(t) \wedge C' \wedge C''$ to be the condition of $t.commit_k$. $C(t)$ is the condition of t in the service specification, C' is a logical formula "$w \neq \phi$" where w is a variable in the label of arc $(p.inner_k, t.commit_k)$, and C'' is a logical formula "$\bigwedge_j x_j \neq \phi$" where x_j is a variable in the label of arc $(\beta_{jk}, t.commit_k)$, and
- for $t.commit_i$, set the predicate $C(t) \wedge C''$ to be the condition of $t.commit_i$.

(D-2) **Set the conditions of *t.abort1* transitions**: This step determines the condition of each $t.abort1_k$ so that $t.abort1_k$ fires only if the condition of t is not satisfied or some input places which are not at site k do not have assignable sets. Formally,
- for each $t.abort1_k$ $(k \neq i)$, set the predicate $\neg C(t) \vee C' \wedge C''$ to be the condition of $t.abort1_k$. $C(t)$ is the condition of t in the service specification, C' is a logical formula "$w \neq \phi$" where w is a variable in the label of arc $(p.inner_k, t.abort1_k)$, and C'' is a logical formula "$\bigvee_j x_j = \phi$" where x_j is a variable in the label of arc $(\beta_{jk}, t.abort1_k)$, and
- for $t.abort1_i$, set the predicate $\neg C(t) \vee C''$ to be the condition of $t.abort1_i$.

(D-3) **Set the conditions of *t.abort2* transitions**: This step determines the condition of each $t.abort2_k$ so that it fires only if some input places at site k do not have assignable sets. Formally,
- for each $t.abort2_k$, set the condition $w = \phi$ where w is a variable in the label of $(p.inner_k, t.abort2_k)$.

3.3.5. Step E: decompose net

After applying Step D, we obtain an integrated form of the protocol specification *Pspec* which includes the behavior specifications for all sites interconnected by the communication places. This step decomposes the obtained net into N independent specifications, one for each site. This is done by splitting each communication place into two places so that the specification of each site can be an independent Pr/T-net.

3.4. Another transition execution protocol

As stated, the transition execution (TE) protocol may cancel an execution of transition t to avoid deadlock. In this case, the values exchanged for using generating tokens are discarded without being used. If the size of the values of these unused tokens is large and if such a cancellation is repeated many times, the protocol becomes inefficient. In this case, we may use the following transition execution protocol that exchanges these values only if it is known that t will be executed (thus no cancellation of t happens after the decision).

We change the original TE protocol as described in Section 3.2 as follows:

1. At Step (1) of the original TE protocol, site i sends only the values used to check the condition of t and only to the reading sites.

2. At Step (2) of the original TE protocol, site j sends only the values used to check the condition of t and only to the other reading sites.
3. At Step (3) of the original TE protocol, only the reading sites examine the conditions (i) and (ii). If (i) and (ii) are true, the reading sites send at this moment the values used to generate new tokens to the writing sites which need them. The writing sites receive these values and generate new tokens.

As an application example, let us consider again the service specification given in Fig. 2(a). Here, we assume that the size of values of the tokens ⟨"*b*"⟩ and ⟨"*d*"⟩ is large. Fig. 5 shows a protocol specification derived based on the above given TE protocol. Similar to the previous example, we assume that the input places p_1, p_2 and p_5 are located on site A, p_3 on site B, and p_4 on site C.

Here unlike the previous TE protocol, the token value assigned to the variable w in place p_4 and needed for generating tokens at site A is not sent to A by $\alpha_{u.ca}$ since it is not needed for examining the condition $x < z$ of t. The value of w is only used for generating tokens at site A. Similarly, the value of y is not sent from site A to site C. If condition of t_u is true, sites A and C send via "$\gamma_{u.ac}$" and "$\gamma_{u.ca}$" the values of y and w, respectively. Using these values, sites A and C generate new tokens. This means that, the large-size token value ⟨"*b*"⟩ or ⟨"*d*"⟩ in p_2 is only sent to site C and this is done only if the condition of t_u is true. Similarly, the large-size value token ⟨"*d*"⟩ in p_4 is only sent to site A and this is done only if the condition of t_u is true. The derivation algorithm corresponding to this TE protocol is given in Appendix A.

4. Timestamp-based contention control

4.1. Motivation and outline

For a given service specification, our derivation algorithm derives a protocol specification that is deadlock free. This is due to the fact that each transition of a service specification can be executed only when it acquires tokens from its input places. In our transition execution algorithm, the primary site of a transition sends requests, for executing

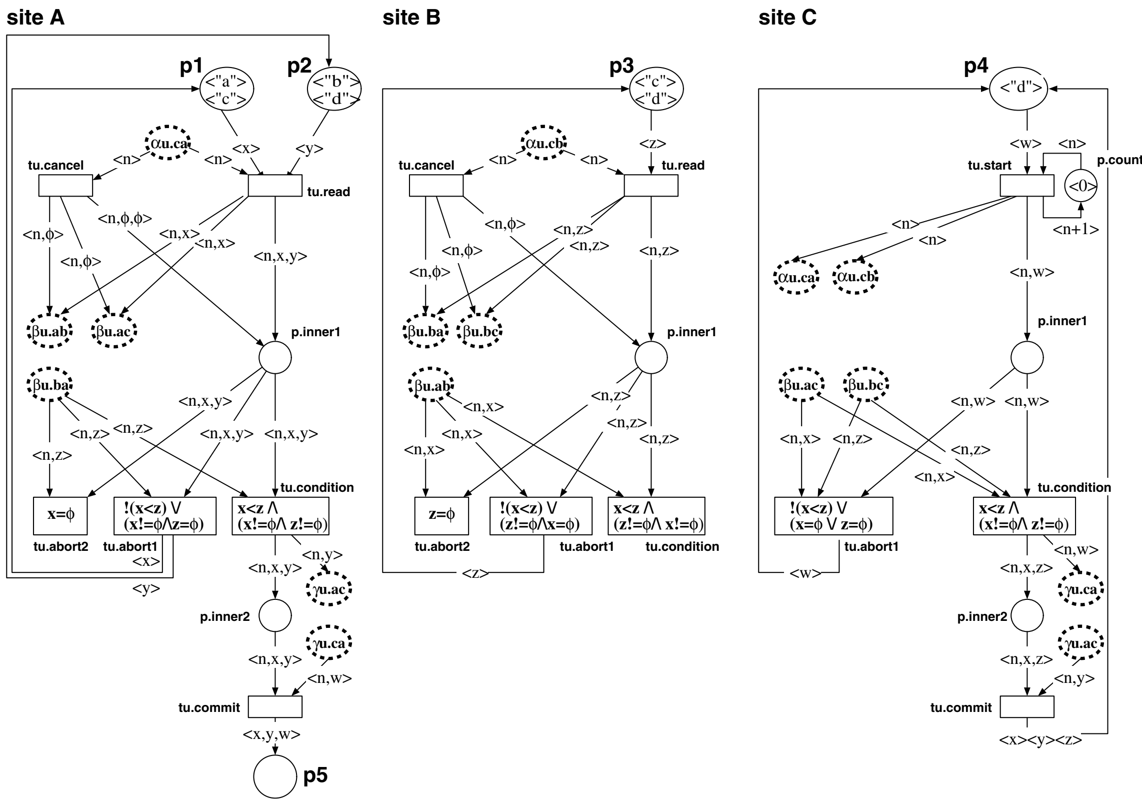


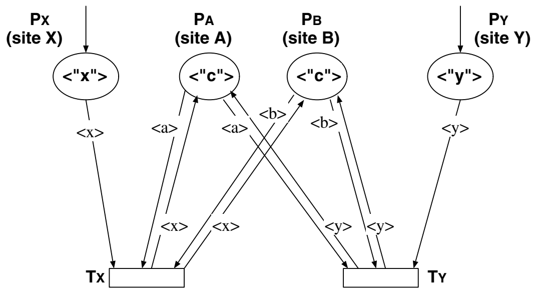
Fig. 5. A protocol specification corresponding to the service specification in Fig. 2(a) derived using the TE protocol given in Section 3.4.

the transition, to all other reading sites that have input places of the transition. If all the required tokens are acquired by these sites and the condition of the transition is true, the execution of the transition will be committed. Otherwise, the execution of the transition will be canceled and thus the protocol specification is always deadlock-free.

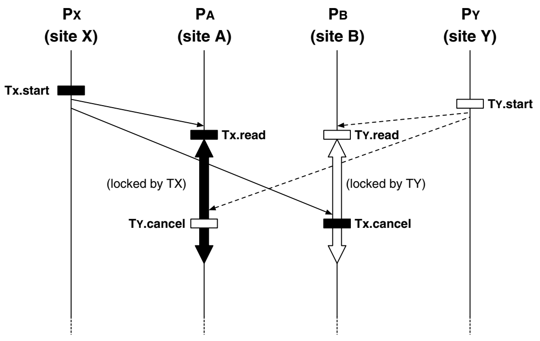
However, if the service specification includes competitive transitions that share multiple choice places (i.e. the Pr/T-net of the service specification is not a free-choice net) as shown in Fig. 6(a), and if the shared choice places are allocated to different sites, the tokens in the shared places are obtained at those different sites by these transitions using a first-come-first-serve policy. In this case, if a transition fails to obtain all the tokens in the shared choice places, it immediately releases already obtained tokens as explained above. As a result, the protocol specification may have live-lock problems, where (some or all) competitive transitions may starve. This may happen particularly if the propagation delays from the primary sites to the sites to which the shared places are allocated are different.

As an example, assume that transitions T_X and T_Y of a given service specification share the two input places P_A and P_B, as shown in Fig. 6(a). Assume that each of these input places has a token, places P_A and P_B are allocated to sites A and B, respectively, and sites X and Y are the primary sites of T_X and T_Y, respectively, as illustrated in Fig. 6(a). It might happen that if the request from the primary site of transition T_X (i.e. site X) has a delay to site A shorter than that of the primary site of T_Y (i.e. site Y), and if the request from the primary site of T_X has a longer delay to site B than that of the primary site of T_Y, then T_X can acquire the token in P_A at site A and T_Y can acquire the token in P_B at site B. Consequently, since neither T_X nor T_Y can acquire both tokens in places P_A and P_B, the requests from the primary sites are not granted and the execution of T_X and T_Y is aborted immediately as shown in the timing chart of Fig. 6(b). This scenario may be repeated several times until either T_X or T_Y successfully acquires both tokens.

In order to reduce the possibility of having such cancellations, we introduce a Timestamp-based Contention Control (TCC) algorithm. The TCC algorithm controls the order of requests to shared places so that all these shared places have the same total order. In order to reduce cancellations, the TCC algorithm does not cancel a request unless it violates the total order at a site which has some shared resources. For this purpose, the TCC uses global time to set an order. First, the primary sites of T_X and T_Y timestamp their requests using the global time, and send these requests to sites A and B. Let $t(T_X)$ and $t(T_Y)$ denote these timestamps and let us assume that $t(T_X) < t(T_Y)$. The timestamp of a request is recorded on the place when the place's token is acquired by the request. If the token of place P_A or P_B has been acquired by the request of T_X when the request of T_Y arrives, the request of T_Y can wait for the request of T_X to release the token, since the ordering $T_X;T_Y$ satisfies the timestamp ordering rule. On the other hand, if the token of place P_A or P_B has been acquired by the request of T_Y, then T_X is canceled immediately since the ordering $T_Y;T_X$ violates the timestamp ordering rule. Consequently, the request of T_Y is not canceled in this case as shown in Fig. 7(a). Moreover, in this case, a subsequent request of T_X with a newer timestamp than T_Y is permitted.



(a)



(b)

Fig. 6. (a) Service specification and place allocation (we assume that the primary sites of T_X and T_Y are sites X and Y, respectively); (b) timing chart (the execution of both T_X and T_Y is aborted).

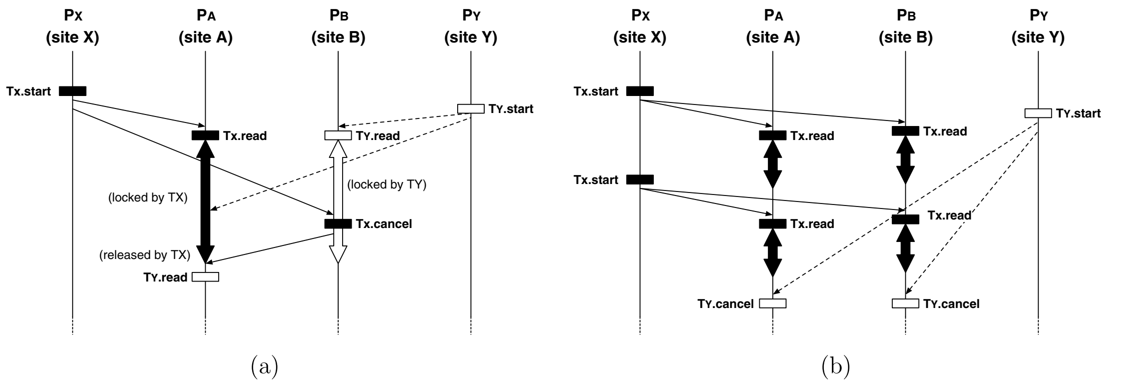


Fig. 7. Timestamp-based contention control algorithm.

The concept of our TCC algorithm is similar to the concurrent transaction control in database systems. But usually such transaction control does not assume multiple shared resources distributed over multiple sites, so we design a new distributed algorithm for such a purpose. The goal of our design is to reduce cancellations without introducing special messages or special structures that may render the protocol specifications more complex. Our TCC algorithm requires a timestamp mechanism that uses global time without introducing additional messages into protocol specifications. Some existing work such as [9,31] has also considered implementing contention control at the protocol level. In this paper, we deal with more complex contention where multiple transactions compete for multiple resources distributed over sites.

In the following subsections, we define a set of transitions and places of a service specification, called *conflict set*, that are subject to contention control. Then we present the TCC algorithm that adds a timestamp mechanism to the parts of the protocol specification corresponding to the conflict set.

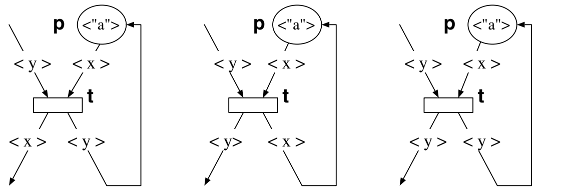
4.2. Preliminaries

We recall that for a transition t, $\bullet t$ (or $t\bullet$) denotes the set of input (or output) places of t. Also, a place which never loses nor gains tokens by firing of transitions (i.e. the place is in self-loops) is called a *persistent place* hereafter.

A set $\mathscr{C}$ of transitions and places is said to be a *conflict set* iff (i) for each place p in $\mathscr{C}$, p has at least two transitions in $\mathscr{C}$ as its output transitions, (ii) for each transition $t \in \mathscr{C}$, t has at least two places in $t \in \mathscr{C}$ as its input places, and (iii) each place in $\mathscr{C}$ is a persistent place. For example, in Fig. 6(a), $\{T_X, T_Y, P_A, P_B\}$ is a conflict set. We apply the TCC algorithm for each conflict set. The reason why we focus only on persistent places is that in such a place, tokens will be returned and transitions can wait for tokens to be back even if some other transitions currently use them.

We say that a pair of a place and a transition $[p, t]$ in a conflict set has a *read attribute* if there exist a variable x and a place p' such that x is in the arc label of (p, t) and x is in the arc label of (t, p'). We also say that $[p, t]$ has a *write attribute* if $L(t, p)$ is not the same as $L(p, t)$ (that is, if (t, p) and (p, t) have different arc labels). $[p, t]$ is said to be (a) *RW-persistent* (i.e. read-write) if $[p, t]$ has both read and write attributes, (b) *RO-persistent* (i.e. read-only) if $[p, t]$ has only a read attribute, and (c) *WO-persistent* (*i.e.* write-only) if $[p, t]$ has only a write attribute.

For example, in Fig. 8(a), $[p, t]$ is RW-persistent since the variable x is used on an output arc of t and $L(t, p) = \langle y\rangle$ is not the same as $L(p, t) = \langle x\rangle$. In Fig. 8(b), $[p, t]$ is RO-persistent since x is used on $L(t, p)$ and $L(t, p) = L(p, t) = \langle x\rangle$. In Fig. 8(c), $[p, t]$ is WO-persistent since x is not used in any output arc of t and $L(t, p)$ is not the same as $L(p, t)$. This classification of place-transition pairs will be used



(a) RW-persistent (b) RO-persistent (c) WO-persistent

Fig. 8. Classification of pair $[p, t]$ in conflict set: (a) RW-persistent, (b) RO-persistent, (c) WO-persistent.

in the TCC algorithm for updating the timestamps attached to tokens in the places of a conflict set.

Here, we introduce two types of variables that will be used by our TCC algorithm. For every token (say c) in a place of a conflict set in the protocol specification, we attach two (timestamp) variables denoted by $R(c)$ and $W(c)$. In the example of Fig. 6(a), we attach $R(c)$ and $W(c)$ variables to the tokens $\langle$"c"$\rangle$ in P_A and P_B in the protocol specification. At the initial marking, the values of these variables are set to zero. We assume that each site has a clock that is synchronized with those of the other sites.[4]

4.3. Timestamp-based contention control mechanism

As stated earlier, our timestamp-based contention control (TCC) algorithm allows a transition t in a conflict set to wait for tokens to be released. For this purpose, when a transition t is selected by a primary site for examining its executability at a global time, the primary site sets a timestamp of the global time (this global time is referred to as *occurrence time of* t) to the variable $o(t)$. This value is sent to the reading sites of t which have places of the conflict set in order to inform these reading sites that transition t has been chosen for examining its executability. If the execution of t at a reading site does not preserve the order of timestamps, then the reading site aborts this execution by sending cancellation tokens to all the other reading and writing sites. In response to these cancellation tokens, these sites release and return to their original places the tokens they already acquired for the execution of t.

In order to incorporate the above mechanism in our protocol specifications, we use the values of the timestamp variables $R(c)$ and $W(c)$ and the occurrence times of transitions set by the primary sites. The decision made by a reading site of a transition t to wait for a token (say c) in a place p of the conflict set or to cancel the execution of t, is based on the set of precondition rules. These rules are chosen according to the type of the pair $[p, t]$ as shown in Table 4.

These rules are inspired from those presented in Ref. [32]. For efficiency, we distinguish the types of place-transition pairs by their operations (read and write operations) so that a read-only operation can ignore other read-only operations (they do not affect each other). For example, rule (b) in Table 4 indicates that the RO-persistent pairs should only consider the order of other pairs that have write operations. Rule (a) and rule (c-i) in Table 4 indicate that RW- and WO-persistent pairs must consider the order of every other pair. Rule (c-ii), which is known as *Thomas's write rule*, indicates that a token value, written by the write-only operation and overwritten by a newer write operation, can be ignored if this value is not used by any read operation.

When transition t tries to acquire a token c from place p at a reading site, it first checks the precondition that corresponds to the type of $[p, t]$. If the precondition is true, the reading site acquires the token. If the token c is not in p, the reading site waits for c to be released. If all reading sites acquire assignable sets, the execution of t will actually take place. In this case, we update the values of the timestamp variables $R(c)$ and $W(c)$ as specified in the corresponding post-condition action shown in Table 4. On the other hand, if the precondition is not true, the reading site of t aborts the execution of t. The details of this algorithm are given in Appendix C.

4.4. Discussion

Our TCC algorithm may cause a fairness problem. For example, in Fig. 6(a), let us assume that the primary site of T_Y has much longer delays to sites A and B than the primary site of T_X. In this case, $t(T_Y)$ may be too old when the request of T_Y arrives at site A or site B, and thus T_Y's request may be cancelled, because during the propagation of T_Y's request, another newer request of T_X may use these tokens and update their timestamps. This results in unnecessary cancellation even if these tokens are available, as shown in Fig. 7(b). This cancellation does not happen if the TCC algorithm is not applied to the protocol specification. That is, without the contention control mechanism by TCC algorithm, T_Y's request is accepted in Fig. 7(b).

Since the TCC algorithm is *independent* of the derivation algorithm, we may choose another design option. For example, to pursue complete fairness, an alternative algorithm can be introduced that prevents cancellations, suffering extra delay. We assume that all the sites know the maximum delay D of all the channels between the sites as well as the global time. The primary site of a transition t sets a timestamp $o(t)$ to a request, and sends it to the reading sites which have shared places. Each reading site which has shared places has a list to store received requests.

[4] We use these clocks only to determine the total order of the execution of transitions in a conflict transition set. Thus, these clocks do not need to be precisely synchronized.

Table 4
Pre- and post-conditions to get token c in p for the execution of t

	Type of $[p, t]$	Pre-condition	Post-condition
(a)	`RW-persistent`	$R(c) < o(t) \wedge W(c) < o(t)$	$R(c) := W(c) := o(t)$
(b)	`RO-persistent`	$W(c) < o(t)$	$R(c) := o(t)$
(c-i)	`WO-persistent`	$R(c) < o(t) \wedge W(c) < o(t)$	$W(c) := o(t)$
(c-ii)		$R(c) < o(t) < W(c)$	–

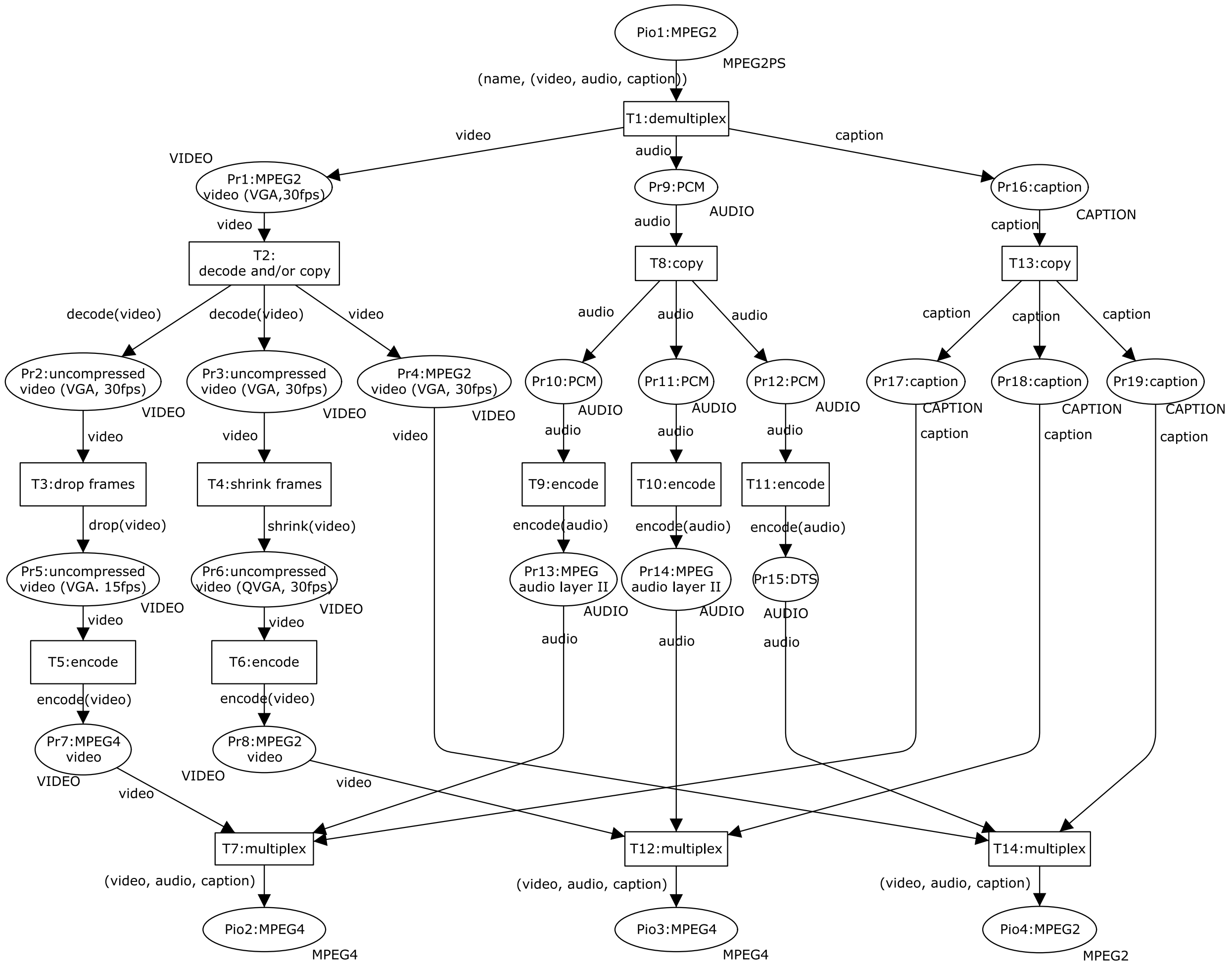


Fig. 9. Application example 1: service specification of an MPEG2 transcoding service.

Any request with timestamp $o(t)$ is removed from the list and processed exactly at time $o(t) + D$. Considering the fact that any request issued no later than global time $o(t)$ arrives no later than time $o(t) + D$, all the requests are completely ordered at each reading site. Due to this feature, no cancellation occurs.[5] Instead, any request must be delayed for D. Depending on the application domain, we may choose another variation of the algorithm. Space limitations do not allow us to discuss this further.

[5] It may happen that two requests have the same timestamp. In such a case we may use the unique identifiers of transitions to determine their orderings.

5. Tool support and application examples

Synthesis methods have been applied to many applications such as communication protocols [26,27], factory manufacturing systems [33], distributed cooperative work management [15] and so on [25,28].

In the following subsections we apply our synthesis method to a distributed Media Transcoding (MT) service on service overlay networks and to a distributed software development process called ISPW-6 [29].

5.1. Toolset

Deriving protocol specification by hand is very complex and time consuming for large systems. Accordingly, we have developed a toolset in Perl that implements our algorithm and co-works with a graphical tool "CPN Tools" [24] for the representation of the service and protocol specifications. In the toolset, we first describe the service specification using CPN Tools that is used to design, simulate and verify coloured Petri nets (CPNs) including Pr/T-nets. CPN Tools can be used for modeling Pr/T-nets since Pr/T-nets can be regarded as a sub-class of CPNs. Second, our tool parses the given service description written in the CPN Tools format (described in XML and DTD) using the XML parser [34]. Third, our Perl program uses the parsed specification and generates a corresponding protocol specification according to our derivation algorithm.

5.2. Application examples

Recently, the use of collaborative computation that connects distributed application components with each other to provide services became very popular. One typical example is a service overlay network where several servers build a virtual backbone by unicast tunnels (an overlay network) to provide transparent services to users [35]. As a realistic example, we consider the design of a media transcoding (MT) service on overlay networks. The MT service decomposes an MPEG2 file into its constituent media (video, audio, text), converts these media representations into versions with different quality and finally combines these different media

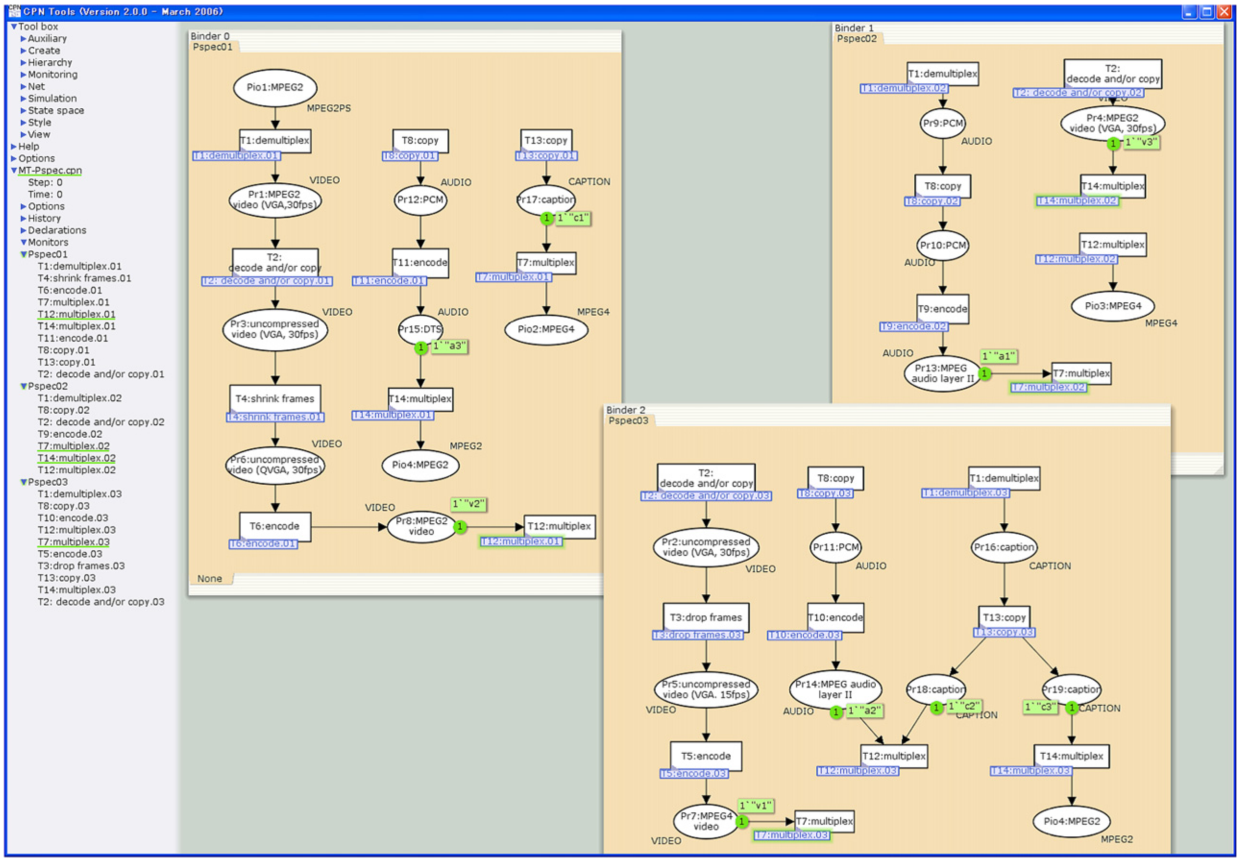


Fig. 10. Protocol specification of MPEG2 transcoding service shown by screen shot from CPN Tools [24]: top level descriptions.

versions to obtain several MPEG files that are suitable for playback by users with various quality requirements. The description of this MT service is shown in Fig. 9 using the notation of CPN Tools. The notation used by CPN Tools is different from what we have seen so far. The names of places and transitions are written inside circles and transitions, respectively. The strings associated with places and written outside the circles are place types, which were not explicitly written in the preceding figures.

In Fig. 9, the inputs and outputs of this service are represented as places named $Pio_1,\ldots, Pio_4$. The MPEG2 file is entered from Pio_1 and decomposed into three components by the transition T_1. Then each component is copied and transcoded, and finally components are merged by T_7, T_{12} and T_{14} in parallel for heterogeneous users which require specific qualities appropriate to the capability of their playback devices. We have distributed this service onto three sites 1, 2 and 3. We have used the following allocation:

Site 1: $\{Pio_1, Pio_2, Pr_1, Pr_3, Pr_6, Pr_8, Pr_{12}, Pr_{15}, Pr_{17}\}$,
Site 2: $\{Pio_3, Pr_4, Pr_9, Pr_{10}, Pr_{13}\}$ and
Site 3: $\{Pio_4, Pr_2, Pr_5, Pr_7, Pr_{11}, Pr_{14}, Pr_{16}, Pr_{18}, Pr_{19}\}$.

Then the derived protocol specification is shown as a screen shot from the CPN Tools in Figs. 10 and 11 (the net layout was adjusted manually). For better readability, our tool provides a hierarchical description of the protocol specification. It includes an overview of the protocol specification as well as the detailed behavior that simulates each transition of the service specification. Specifically, the protocol specification of each site i, say $Pspec_i$, preserves the structure of the service specification where only the related transitions and the places allocated to that site remain. This is referred to as a *top level description*. Each transition t of $Pspec_i$ in the top level description is a *substitution transition*, which is replaced with a net called *subpage* that simulates

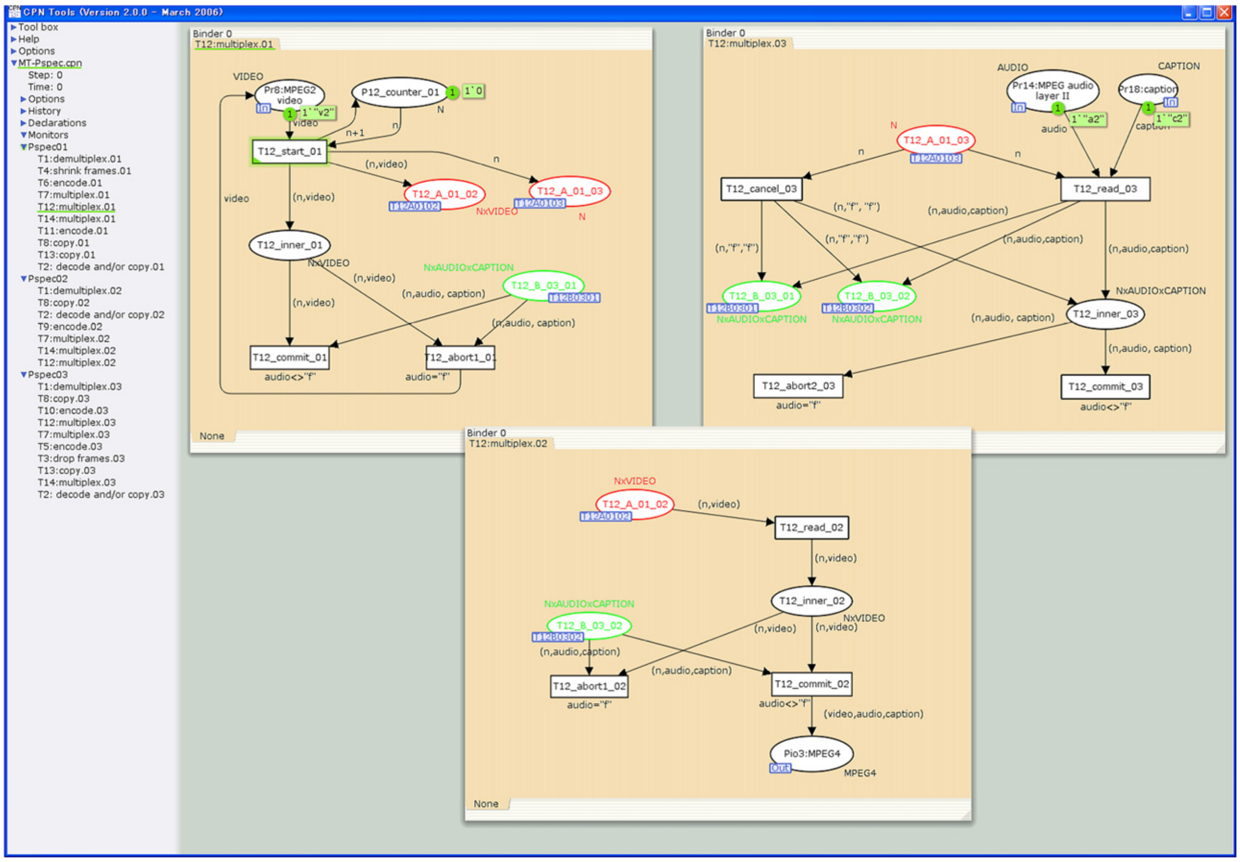


Fig. 11. Protocol specification of MPEG2 transcoding service shown by screen shot from CPN Tools [24]: sub-pages corresponding to substitution transition T_{12}.

transition t. A substitution transition and its corresponding subpage are interconnected by fusion places. Fig. 10 shows the top level descriptions of $Pspec_1$, $Pspec_2$ and $Pspec_3$, and Fig. 11 shows the three sub-pages substituted for transition T_{12} in these top level descriptions. The sub-pages of the other transitions are omitted due to space limitations.

As another example, we consider distributed development of software that involves five engineers (project manager, quality assurance, design, and two software engineers). Each engineer has his/her own machine connected through a network, and participates in the software development process using the machine. The resources used in the process are allocated to those machines. Then the distributed process specification of each engineer clearly indicates how he/she proceeds with the development process. This development process includes tasks for scheduling and assigning tasks, design modification, design review, code modification, test plan modification, modification of unit test packages, unit testing, and progress monitoring. The engineers cooperate with each other to finish these sub-sequential tasks in a suitable order. The reader may refer to *ISPW-6 Core Problem* [29] for a complete description of this process, which was provided as an example to help the understanding and comparison of various approaches to process modeling. Fig. 12

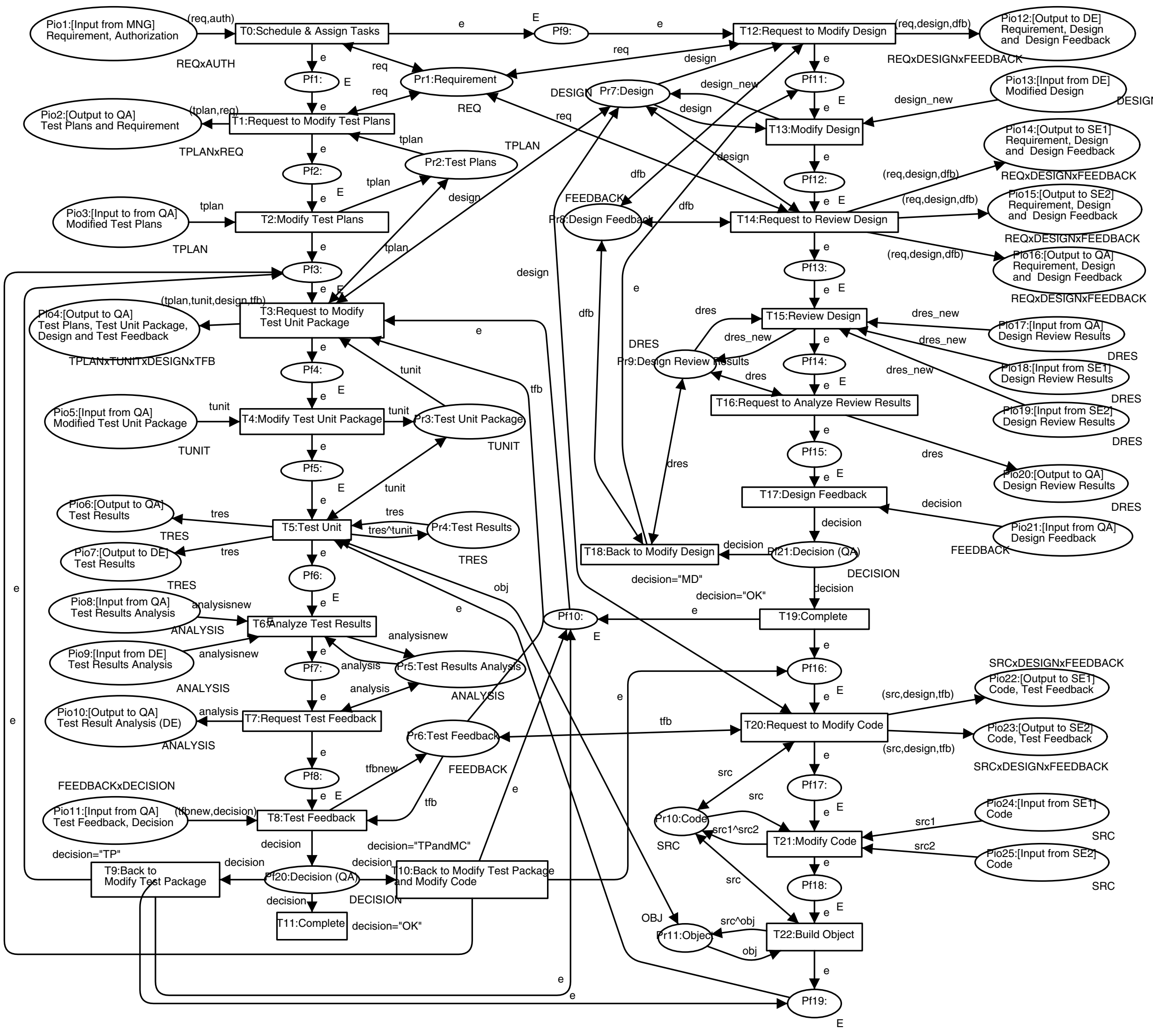

Pio1:[Input from MNG] Requirement, Authorization
T0:Schedule & Assign Tasks
Pf9:
T12:Request to Modify Design
Pio12:[Output to DE] Requirement, Design and Design Feedback
Pr1:Requirement
Pr7:Design
T1:Request to Modify Test Plans
T13:Modify Design
Pr2:Test Plans
T2:Modify Test Plans
Pr8:Design Feedback
T14:Request to Review Design
T3:Request to Modify Test Unit Package
T15:Review Design
T4:Modify Test Unit Package
Pr3:Test Unit Package
T16:Request to Analyze Review Results
T5:Test Unit
Pr4:Test Results
T17:Design Feedback
T18:Back to Modify Design
Pr21:Decision (QA)
T6:Analyze Test Results
T19:Complete
Pr5:Test Results Analysis
T7:Request Test Feedback
Pr6:Test Feedback
T20:Request to Modify Code
T8:Test Feedback
Pr10:Code
T21:Modify Code
T9:Back to Modify Test Package
Pf20:Decision (QA)
T10:Back to Modify Test Package and Modify Code
T11:Complete
Pr11:Object
T22:Build Object
Pf19:

Fig. 12. Application example 2: service specification of ISPW-6 given in [29].

shows the process description in the notation of the CPN Tools. We note that for convenience, we do not show the progress monitoring tasks in Fig. 12. We omit to show the protocol specification due to space limitations.

It took 15 and 25 seconds on a Windows XP PC with Intel Xeon 3.0 GHz CPU and 1.5 GB memory to derive these protocol specifications.

6. Conclusion and current research work

We have proposed a protocol synthesis technique for systems modeled as Pr/T-nets (predicate/transition-nets), a first-order extension of Petri nets. Our technique is based on a top-down approach where a service requirement is defined in the form of a Pr/T-net with a centralized view, and then it is decomposed into communicating components located on different sites which together provide the required service. The originality of our approach is the fact that non-restricted Petri nets with conflicting transitions can be used for the description of the services which are the starting point of the protocol development. This is a very important feature for modeling recent distributed collaborative systems since they often include multiple (and to be distinguished) processes such as mobile users. Moreover, we have presented a contention control algorithm for a distributed environment based on timestamps. A tool has been developed that implements our algorithm and works together with other existing tools. Moreover, two application examples are provided. The existing Petri net based approaches [9–15] do not support all of the above features.

We would like to mention the advantage of our algorithm in terms of readability and reusability of derived specifications, which are very important factors in software development and management. Our algorithm splits (decomposes) one by one every transition of the service into corresponding transitions that can be executed in a distributed way. Accordingly, the structure of the derived protocol specification is similar to the structure of the service specification which increases the readability. Moreover, due to the nature of our derivation algorithm, we have a relation between each transition of the service specification with corresponding transitions of the protocol specification. Similar to our previous work in [15], this relation makes it easy to re-use unaffected parts of an already derived protocol specification after modifying the corresponding service specification.

The synthesis protocol presented in this paper assumes that the allocation of service places to different communicating components is given. However, in the context of distributed applications, there may be a large number of possible place allocations, and their choice may have an important impact on the performance of the resulting system. Therefore it is desirable to find an optimized place allocation and system. This is part of our current research work. In a preliminary version of this work [36] we incorporated into the protocol presented in this paper a model that determines an optimal allocation of places based on some cost criteria such as channel utilization and total response time costs. Currently, we are enhancing this model by incorporating more cost criteria that could be used in various application areas for deriving protocol specifications.

Acknowledgement

The authors would like to thank Dr. Juan Quemada and the anonymous referees for their useful comments.

Appendix A. Derivation algorithm for another te protocol

We present the derivation algorithm corresponding to another TE protocol presented in Section 3.4. We use the notations in Table 3 and the rules in Fig. 4. For simplicity of notations, we let i denote $ps(t)$ in this algorithm description.

A.1. Step A: decompose transitions

(A-1) Decompose *t* into *t.condition* transitions and *t.commit* transitions:
- Apply Rule 1 to divide t into a set of $t.condition_k$ transitions and a set of $t.commit_h$ transitions such that for each $k \in RS(t)$ a $t.condition_k$ transition is created at site k and for each $h \in WS(t) \backslash RS(t)$ a $t.commit_h$ transition is created at site h. Then attach the places in $ALC_k(\bullet t) \cup ALC_k(t\bullet)$ to $t.condition_k$ and the places in $ALC_h(t\bullet)$ to $t.commit_k$.

(A-2) Add *t.commit* transitions and *p.inner2* places:
- For each $t.condition_k$ ($k \in RS(t) \cap WS(t)$), apply Rule 2 to insert a place $p.inner2_k$ and a transition $t.commit_k$ after $t.condition_k$.

(A-3) Add *t.start* and *t.read* transitions and *p.inner1* places:

- For each $t.condition_k$ ($k \neq i$), apply Rule 2 to insert a transition $t.read_k$ and a place $p.inner1_k$ before $t.condition_k$, and
- for $t.condition_i$, apply Rule 2 to insert a transition $t.start$ and a place $p.inner1_i$ before $t.condition_i$.

(A-4) Add communication places:

- For each pair of $t.start$ and $t.read_j$, apply Rule 3 to insert a communication place α_{ij},
- for each pair of $t.read_j$ and $t.condition_k$ ($j \neq k$), apply Rule 3 to insert a communication place β_{jk},
- for each pair of $t.condition_k$ and $t.commit_h$ ($h \in RS(t)$ and $k \neq h$) where $Vin_k(t) \cap Vout_h(t) \setminus (Vin_h(t) \cup Vcond(t)) \neq \emptyset$, apply Rule 3 to insert a communication place γ_{kh},
- for each pair of $t.condition_k$ and $t.commit_h$ ($h \notin RS(t)$ and $k \neq h$) where $(Vin_k(t) \cup Vcond(t)) \cap Vout_h(t) \neq \emptyset$, apply Rule 3 to insert a communication place γ_{kh}, and
- for each $t.commit_h$ ($h \notin RS(t)$) where $Vout_h(t) = \emptyset$, apply Rule 3 to insert a communication place γ_{ih} from $t.condition_i$ to $t.commit_h$.

A.2. Step B: introduce cancellation and identification mechanisms

(B-1) Add *t.cancel* transitions:

- For each $t.read_j$, apply Rule 4 to add a transition $t.cancel_j$ where $\bullet t.cancel_j = \{\alpha_{ij}\}$ and $t.cancel_j \bullet = t.read_j \bullet$.

(B-2) Add *t.abort1* transitions:

- For each $t.condition_k$, apply Rule 4 to add a transition $t.abort1_k$ where $\bullet t.abort1_k = \bullet t.condition_k$ and $t.abort1_k \bullet = ALC_k(\bullet t)$.

(B-3) Add *t.abort2* transitions:

- For each $t.condition_k$ ($k \neq i$), apply Rule 5 to add a transition $t.abort2_k$ where $\bullet t.abort2_k = \bullet\ t.condition_k$.

(B-4) Add *p.counter* place:

- For $t.start$, apply Rule 6 to add a place $p.counter$ with a token $\langle 0 \rangle$ inside.

A.3. Step C: set arc labels

(C-1) Set arc labels of *p.counter* place:

- Set $\langle n \rangle$ as the label of $(p.counter, t.start)$ where n is an integer variable, and $\langle n+1 \rangle$ as the label of $(t.start, p.counter)$.

(C-2) Determine values carried by α communication places:

- For each α_{ij}, set the tuple of the variable n and all the variables in $Vin_i(t) \cap Vcond(t) \setminus Vin_j(t)$ as the labels of the arcs $(t.start, \alpha_{ij})$, $(\alpha_{ij}, t.read_j)$ and $(\alpha_{ij}, t.cancel_j)$.

(C-3) Determine values carried by β communication places:

- For each β_{jk}, set the tuple of the variable n andthe variables in $Vin_j(t) \cap Vcond(t) \setminus Vin_k(t)$ as the labels of the arcs $(t.read_j, \beta_{jk})$, $(\beta_{jk}, t.condition_k)$, $(\beta_{jk}, t.abort1_k)$ and $(\beta_{jk}, t.abort2_k)$, and
- set a tuple of the variable n and m ϕ's as the label of $(t.cancel_j, \beta_{jk})$ where $m = |Vin_j(t) \cap Vcond(t) \setminus Vin_k(t)|$.

(C-4) Determine values carried by γ communication places:

- For each γ_{kh} ($h \in RS(t)$), set a tuple of the variable n and the variables in $(Vin_k(t) \cap Vout_h(t)) \setminus (Vin_h(t) \cup Vcond(t))$ as the labels of the arcs $(t.condition_k, \gamma_{kh})$ and $(\gamma_{kh}, t.commit_h)$, and
- for each γ_{kh} ($h \notin RS(t)$), set a tuple of the variable n and the variables in $(Vin_k(t) \cup Vcond(t)) \cap Vout_h(t)$ as the labels of the arcs $(t.condition_k, \gamma_{kh})$ and $(\gamma_{kh}, t.commit_h)$.

(C-5) Determine values kept by *p.inner1* places:

- For $p.inner1_i$, set a tuple of the variable n and all the variables in $Vin_i(t) \cap Vcond(t)$ as the labels of the arcs $(t.start, p.inner1_i)$, $(p.inner1_i, t.condition_i)$ and $(p.inner1_i, t.abort1_i)$,
- for each $p.inner1_j$ ($j \neq i$), set a tuple of the variable n and all the variables in $Vin_i(t) \cap Vcond(t) \cup Vin_j(t)$ as the labels of the arcs $(t.read_j, p.inner1_j)$, $(p.inner1_j, t.condition_j)$, $(p.\ inner1_j, t.abort1_j)$ and $(p.inner1_j, t.\ abort2_j)$, and
- set a tuple of the variable n and m ϕ's as the label of the arc $(t.cancel_j, p.inner1_j)$ where $m = |Vin_i(t) \cap Vcond(t) \cup Vin_j(t)|$.

(C-6) Determine values kept by *p.inner2* places:

- For each $p.inner2_h$, set a tuple of the variable n and all the variables in $Vout_h(t) \cap (Vin_h(t) \cup Vcond(t))$ as the labels of the arcs $(t.condition_h, p.inner2_h)$ and $(p.inner2_h, t.commit_h)$.

(C-7) Determine values returned by *t.abort1* transitions:

- For each $(t.abort1_k, p)$ ($p \in ALC_k(\bullet t)$), set its label as that of $(p, t.read_k)$ (or $(p, t.start)$).

A.4. Step D: set conditions

(D-1) Set the conditions of *t.condition* transitions:

- For each $t.condition_k$ $(k \neq i)$, set the predicate $C(t) \wedge C' \wedge C''$ to be the condition of $t.condition_k$. $C(t)$ is the condition of t in the service specification, C' is a logical formula "$w \neq \phi$" where w is a variable in the label of arc ($p.inner1_k$, $t.condition_k$), and C'' is a logical formula "$\bigwedge_j x_j \neq \phi$" where x_j is a variable in the label of arc (β_{jk}, $t.condition_k$), and
- for $t.condition_i$, set the predicate $C(t) \wedge C''$ as the condition of $t.condition_i$.

(D-2) Set the conditions of *t.abort1* transitions:

- For each $t.abort1_k$ $(k \neq i)$, set the predicate $\neg C(t) \vee (C' \wedge C'')$ to be the condition of $t.abort1_k$. $C(t)$ is the condition of t in the service specification, C' is a logical formula "$w \neq \phi$" where w is a variable in the label of arc ($p.inner1_k$, $t.abort1_k$), and C'' is a logical formula "$\bigvee_j x_j = \phi$" where x_j is a variable in the label of arc (β_{jk}, $t.abort1_k$).
- For $t.abort1_i$, set the predicate $\neg C(t) \vee C''$ to be the condition of $t.abort1_i$.

(D-3) Set the conditions of *t.abort2* transitions:

- For each $t.abort2_k$, set the condition $w = \phi$ where w is a variable in the label of ($p.inner1_k$, $t.abort2_k$).

A.5. Step E: decompose net

After applying Step D, we obtain an integrated form of the protocol specification *Pspec* which includes the behavior specifications for all sites interconnected by the communication places. This step decomposes the obtained net into N independent specifications, one for each site. This is done by splitting each communication place into two places so that the specification of each site can be an independent Pr/T-net.

Appendix B. On the validity of the derivation algorithm

Here we comment on the validity of the derivation algorithm presented in Section 3.3. Particularly, we show that the Pr/T-subnets obtained by applying the TE protocol to a transition t of *Sspec* realize the same behavior as t. Moreover, any executable transition sequence of *Sspec* is preserved in *Pspec* and vice versa.

Hereafter, the set of N sites is denoted by S. Also *Pspec* of site i is denoted by $Pspec_i$. Thus $\cup_{i \in S} Pspec_i = Pspec$. Moreover, the Pr/T-subnet of $Pspec_i$ that corresponds to t is denoted by $Pspec_i(t)$. In Fig. B.1, we show $Pspec_A$ of Fig. 2(b) as an example where $Pspec_A(t_u)$ is emphasized for readability. We first comment on the fact that $\{Pspec_i(t) | \forall i \in S\}$ simulates the behavior of t.

Validity of simulation of each transition. Due to Step *A* of the algorithm, any combination of *t.start*, $t.read_j$ and $t.commit_k$ are executed in this order. Due to Step B of the algorithm, $t.cancel_j$ may fire instead of $t.read_j$ at a reading site, and $t.abort1_k$ or $t.abort2_k$ may fire instead of $t.commit_k$ at a reading or writing site. Consequently, we can say that any combination of *t.start*, $t.read_j$ (or $t.cancel_j$) and $t.commit_k$ (or $t.abort1_k$ or $t.abort2_k$) are executed in this order.

Due to Step C of the algorithm, we can easily say that each $t.commit_k$ at a reading site has all the values needed to check the condition of t and to generate tokens to the output places of t allocated to site k, in its input places. We can also say that $t.abort1_k$ has all the values needed to be returned to the input places of t allocated to site k. Assuming these facts, we show the following facts for the validity of the algorithm. (1) If *t.start* and all $t.read_j$ transitions fire and if the condition of t is true, then all $t.commit_k$ transitions will fire, and tokens are generated to the output places of t. (2) If *t.start* and all $t.read_j$ transitions fire and if the condition of t is *not* true, then all $t.abort1_k$ transitions will fire that restore the tokens taken by $t.read_k$ transitions to the input places of t. (3) If *t.start*, some $t.read_j$ transitions and some $t.cancel_{j'}$ transitions fire, then $t.abort1_j$ transitions will fire that restore the tokens taken by $t.read_j$ transitions to the input places of t and $t.abort2_{j'}$ transitions will fire.

First, due to Step D-1 of the algorithm, each $t.commit_k$ fires only if *t.start* and all the $t.read_j$ fire and if the condition of t is true. Therefore, the above fact (1) is shown. Secondly, due to Step D-2 of the algorithm, each $t.abort1_k$ will fire only if *t.start* and all the $t.read_j$ fire but the condition of t is not true. Therefore, the above fact (2) is shown. Finally, let j' denote a reading site where some input places of t allocated to site j' do not have assignable sets. In this case, $t.cancel_{j'}$ fires and each reading or writing site (say k) receives a token filled with ϕ via communication place $\beta_{j'k}$. As a result, due to Step D-2 of the algorithm, $t.abort1_k$ will fire and the tokens taken by $t.read_k$ transition are restored to the input place of t at site k. At site j, after firing of $t.cancel_j$, a

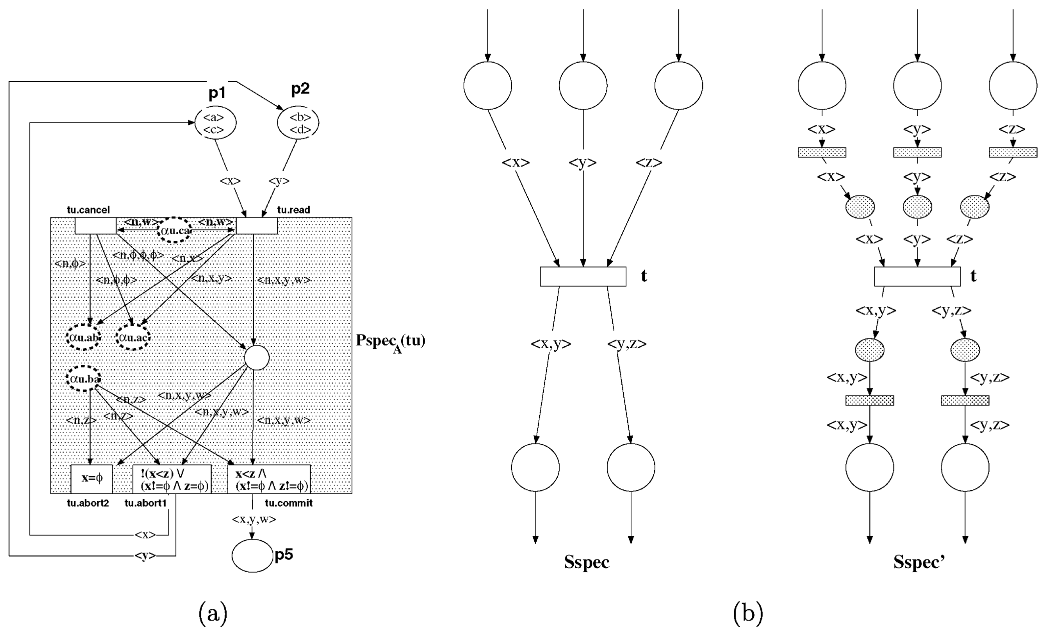


Fig. B.1. (a) $Pspec_A$ of Fig. 2(b) ($Pspec_A(t_u)$ is emphasized by a meshed square.); (b) *Sspec* and *Sspec'*.

token filled with ϕ is generated in $p.inner_k$. Due to Step D-3 of the algorithm, $t.abort2_j$ will eventually fire. In this case, no assignable sets are returned because no assignable sets are taken from the input places at site j.

Then assuming the correctness of $\{Pspec_i(t)|\forall i \in S\}$, we comment on the fact that any transition sequence executable in *Sspec* is preserved in *Pspec* and vice versa. For simplicity of discussion, we assume that any $\{Pspec_i(t)|\forall i \in S\}$ does not have the cancellation mechanism, which just returns the obtained assignable sets to the original input places.

Validity of simulation of whole net. Due to Step A of the algorithm, $\{Pspec_i(t)|\forall i \in S\}$ has the following properties. (1) *t.start* is executed before any *t.read* transition. (2) *t.start* transition and all *t.read* transitions at reading sites are executed before any *t.commit* transition. (3) Once a *t.commit* transition is executed, all the rest of *t.commit* transitions will eventually be executed. Here, since any reading site can be a primary site in our algorithm, the execution order of *t.start* and *t.read* does not affect the validity. Therefore, we ignore the property (1) in this proof. Each $\{Pspec_i(t)|\forall i \in S\}$ simulates the behavior of t correctly, but from the properties (2) and (3), timing to take tokens from input places of t at each reading site may not be synchronized with the other reading sites, and timing to produce tokens to output places of t at each writing site may not also be synchronized with the other writing sites. From this fact, we can regard that *Pspec* is equivalent to a modified *Sspec* where a pair of a dummy transition and a dummy place is inserted for each pair of a transition and a place of *Sspec*. We denote the modified *Sspec* by *Sspec'* and show an example in Fig. B.1(b), where dummy transitions and places are shown by meshed parts. Obviously, this transformation does not change the executable set of transition sequences. Then we can obviously say that any transition sequence executable in *Sspec* is also executable in *Sspec'* and vice versa. This means that any executable transition sequence of *Sspec* is preserved in *Pspec* and vice versa.

Appendix C. Detailed algorithm for adding timestamp-based contention control mechanism

We present the algorithm to add our Timestamp-based Contention Control (TCC) described in Section 4.

For each conflict set $\mathscr{C}$ in the service specification, this algorithm is applied to the corresponding part of the protocol specification. We let i denote the primary site of t.

1. **Set timestamps and identifiers to tokens:** For each token c in place $p \in \mathscr{C}$, add two zero values (the initial values of the read and write timestamps) and a unique identifier to c.
2. **Set occurrence time of transitions:** For each transition $t \in \mathscr{C}$, add occurrence time $o(t)$ to the label of $(t.start, \alpha_{ij})$ at the primary site i and to the labels of $(\alpha_{ij}, t.read_j)$ and $(\alpha_{ij}, t.cancel_j)$ at each site j.

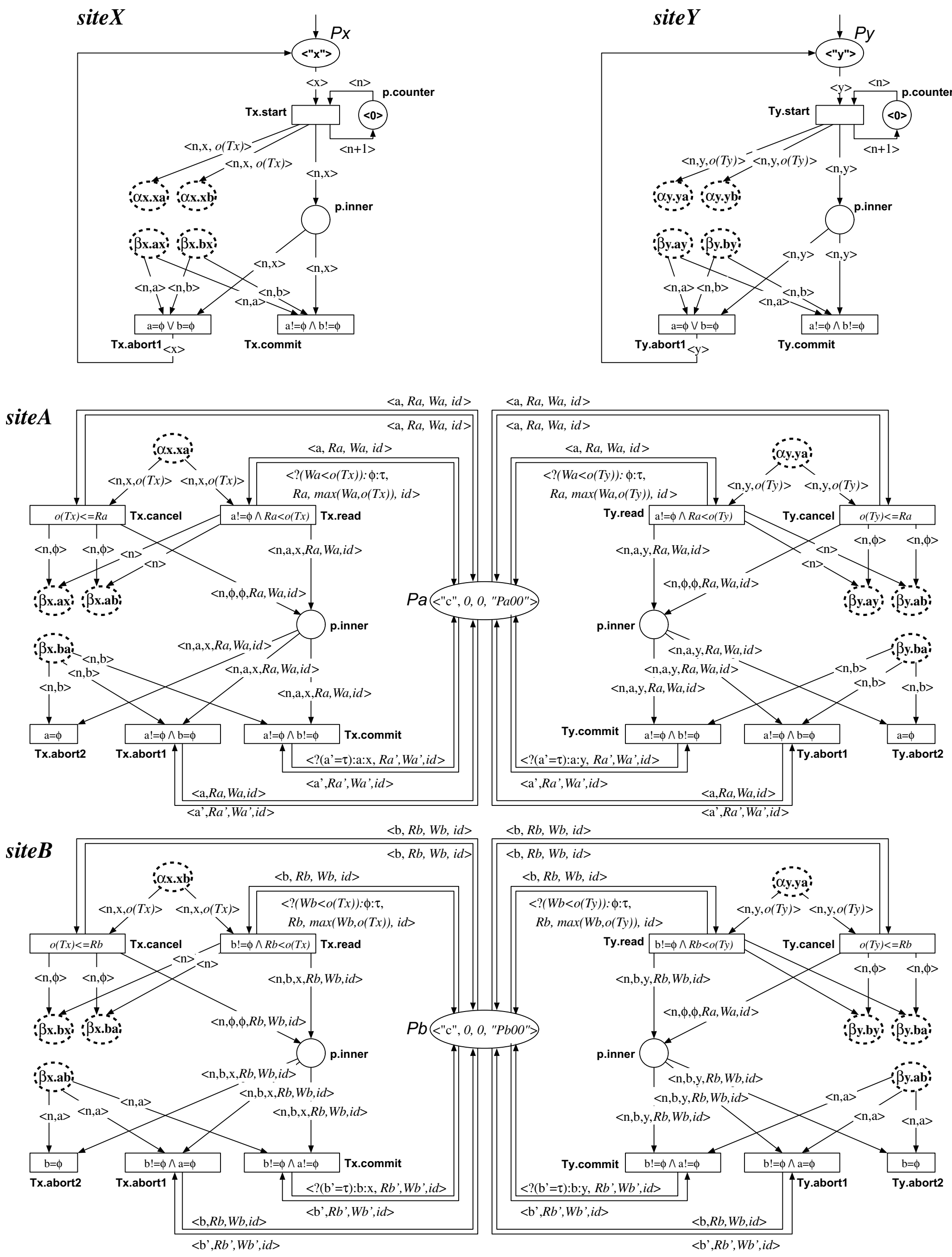


Fig. C.1. Timestamp-based contention control mechanism is applied to protocol specification derived from service specification and place allocation in Fig. 6(a).

3. **Add arcs:** For each connected pair $[p, t]$ where $p, t \in \mathscr{C}$ (we let j denote the site which has place p), add arcs $(t.read_j, p)$, $(t.cancel_j, p)$, $(p, t.cancel_j)$, $(p, t.commit_j)$ and $(p, t.abort1_j)$ to the protocol specification.
4. **Set transition conditions and arc labels:** For each connected pair $[p, t]$ where $p, t \in \mathscr{C}$ (we let j denote the site which has place p), we identify each tuple (say u) of variables in the label of arcs $(p, t.read_j)$ (or $(p, t.start)$). Then there must be the tuples in the labels of arcs $(t.commit_j, p)$ and $(t.abort1_j, p)$ that return the token assigned to u since p is a persistent place. We denote these tuples by v and w, respectively. The followings are applied for each tuple u (and its corresponding tuples v and w). We introduce two timestamp variables $R(u)$ and $W(u)$ to store the timestamps of a token assigned to u, and the variable $id(u)$ that carries the identifier of the token. We also introduce two timestamp variables $R(u)'$ and $W(u)'$ to store the timestamps updated by the post-conditions of Table 4.
   (i) Add the pre-condition of Table 4 corresponding to the type of $[p, t]$ to the condition of $t.read_j$ (or $t.start$). If $[p, t]$ is WO-persistent, use "$R(u) < o(t)$" instead of the precondition of Table 4. This is to implement Thomas's write rule. Also add the negation of the pre-condition to the condition of $t.cancel_j$.
   (ii) Add a logical formula $z \neq \phi$ where z is a variable in u to the condition of $t.read_j$.
   (iii) Add the variables $R(u)$, $W(u)$ and $id(u)$ to the labels of $(p, t.read_j)$ (or $(p, t.start)$). Then set the same label of $(p, t.read_j)$ to $(p, t.cancel_j)$ and $(t.cancel_j, p)$.
   (iv) Add the tuple of ϕ's, the updated timestamps by the corresponding post-condition of Table 4, and the variable $id(u)$ to the label of arc $(t.read_j, p)$. If $[p, t]$ is WO-persistent, use the function "$?(W(u) < o(t))$: ϕ: τ" instead of each ϕ. This is to implement Thomas's write rule. "$?A$:B:C" is a function that returns B if A is true, and returns C if A is not true.
   (v) Add the variables $R(u)$, $W(u)$ and $id(u)$ to the labels of the arcs to/from $p.inner$ place.
   (vi) Add the variables $R(u)'$, $W(u)'$ and $id(u)$ to tuple v in the label of arc $(t.commit_j, p)$. Correspondingly, add the tuple (say v') of arbitrary but unique variables and the variables $R(u)'$, $W(u)'$ and $id(u)$ to the label of arc $(p, t.commit_j)$. If $[p, t]$ is WO-persistent, instead of each function (say y) in tuple v, use "$?(z = \tau)$: x: y" where z is a variable in tuple v' and x is the variable in tuple u that corresponds to function y. This is to implement Thomas's write rule.
   (vii) Add the variables $R(u)$, $W(u)$ and $id(u)$ to tuple w in the label of arc $(t.abort1_j, p)$. Correspondingly, add the tuple of arbitrary but unique variables and the variables $R(u)'$, $W(u)'$ and $id(u)$ to the label of $(p, t.abort1_j)$.

The protocol specification corresponding to the service specification and place allocation in Fig. 6(a) where this algorithm is applied is shown in Fig. C.1.

References

[1] R. Gotzhein, G.v. Bochmann, Deriving protocol specifications from service specifications including parameters, ACM Trans. Comput. Syst. 8 (4) (1990) 255–283.
[2] H. Erdogmus, R. Johnston, On the specification and synthesis of communicating processes, IEEE Trans. Software Eng. 16(12).
[3] C. Kant, T. Higashino, G.v. Bochmann, Deriving protocol specifications from service specifications written in LOTOS, Distrib. Comput. 10 (1) (1996) 29–47.
[4] P.-Y.M. Chu, M.T. Liu, Protocol synthesis in a state-transition model, in: Proceedings of COMPSAC '88, 1988, pp. 505–512.
[5] T. Higashino, K. Okano, H. Imajo, K. Taniguchi, Deriving protocol specifications from service specifications in extended FSM models, in: Proceedings of 13th International Conference on Distributed Computing Systems (ICDCS-13), 1993, pp. 141–148.
[6] B. Caillaud, P. Caspi, A. Girault, C. Jard, Distributing automata for asynchronous networks of processors, Eur. J. Autom. Syst. (JESA) (1997) 503–524.
[7] J.C. Park, R.E. Miller, Synthesizing protocol specifications from service specifications in timed extended finite state machines, in: Proceedings of 17th International Conference on Distributed Computing Systems (ICDCS-17), 1997.
[8] A. Khoumsi, K. Saleh, Two formal methods for the synthesis of discrete event systems, Comput. Networks ISDN Syst. 29 (7) (1997) 759–780.
[9] H. Kahlouche, J. Girardot, A stepwise requirement based approach for synthesizing protocol specifications in an interpreted Petri net model, in: Proceedings of INFOCOM '96, 1996, pp. 1165–1173.
[10] H. Yamaguchi, K. Okano, T. Higashino, K. Taniguchi, Protocol synthesis from time Petri net based service specifications, in: Proceedings of 1997 International Conference on Parallel and Distributed Systems (ICPADS'97), 1997, pp. 236–243.

[11] A. Al-Dallal, K. Saleh, Protocol synthesis using the Petri net model, in: Proceedings of 9th International Conference on Parallel and Distributed Computing and Systems (PDCS'97), 1997.
[12] K. El-Fakih, H. Yamaguchi, G.v. Bochmann, A method and a genetic algorithm for deriving protocols for distributed applications with minimum communication cost, in: Proceedings of 11th International Conference on Parallel and Distributed Computing and Systems (PDCS'99), 1999, pp. 863–868.
[13] H. Yamaguchi, K. El-Fakih, G.v. Bochmann, T. Higashino, A Petri net based method for deriving distributed specification with optimal allocation of resources, in: Proceedings of ASIC International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing (SNPD'00), 2000, pp. 19–26.
[14] K. El-Fakih, H. Yamaguchi, G.v. Bochmann, T. Higashino, Automatic derivation of Petri net based distributed specification with optimal allocation of resources, in: Proceedings of 15th IEEE International Conference on Automated Software Engineering (ASE'2000), 2000, pp. 305–308.
[15] H. Yamaguchi, K. El-Fakih, G.v. Bochmann, T. Higashino, Protocol synthesis and re-synthesis with optimal allocation of resources based on extended Petri nets, Distrib. Comput. 16 (1) (2003) 21–35.
[16] K. El-Fakih, H. Yamaguchi, G.v. Bochmann, T. Higashino, Petri net protocol synthesis with minimum communication costs, J. Franklin Inst.: Eng. Appl. Math., in press.
[17] L. Tanguy, C. Viho, C. Jard, Synthesizing coordination procedures for distributed testing of distributed systems, in: Proceedings of Distributed System Validation and Verification (DSVV'2000), 2000, pp. E67–E74.
[18] M. Kapus-Kolar, Deriving protocol specifications from service specifications with heterogeneous timing requirements, in: Proceedings of 1991 International Conference on Software Engineering for Real Time Systems, 1991, pp. 266–270.
[19] A. Khoumsi, G.v. Bochmann, R. Dssouli, On specifying services and synthesizing protocols for real-time applications, in: Proceedings of 14th IFIP WG6.1 Symp. on Protocol Specification, Testing and Verification (PSTV-14), 1994, pp. 185–200.
[20] A. Nakata, T. Higashino, K. Taniguchi, Protocol synthesis from timed and structured specifications, in: Proceedings of 1995 International Conference on Network Protocols (ICNP'95), 1995, pp. 74–81.
[21] H.J. Genrich, K. Lautenbach, The analysis of distributed systems by means of Predicate/transition-nets, in: Proceedings of International Symp. on Semantics of Concurrent Computation (LNCS 70), 1979, pp. 123–147.
[22] H.J. Genrich, K. Lautenbach, System modeling with high-level Petri nets, Theoret. Comput. Sci. 13 (1) (1981) 109–136.
[23] K. Jensen, Coloured Petri NetsBasic Concepts, Analysis Methods and Practical Use Volume: 1 Basic Concepts, Monographs in Theoretical Computer Science An EATCS Series, Springer-Verlag, 1997.
[24] A. Ratzer, L. Wells, H. Lassen, M. Laursen, J. Qvortrup, M. Stissing, M. Westergaard, S. Christensen, K. Jensen, CPN Tools for editing, simulating, and analyzing coloured Petri nets, in: Proceedings of 24th International Conference on Application and Theory of Petri Nets, 2003.
[25] L.A. Cherkasova, V.E. Kotov, T. Rokicki, On net modeling of industrial size concurrent systems, in: Proceedings of 14th International Conference on Application and Theory of Petri Nets 1993 (LNCS 691), Springer-Verlag, 1993, pp. 552–561.
[26] P. Huber, V. Pinci, A formal executable specification of the ISDN basic rate interface, in: Proceedings of 12th International Conference on Application and Theory of Petri Nets, 1991, pp. 1–21.
[27] J. de Figueiredo, L. Kristensen, Using coloured Petri nets to investigate behavioural and performance issues of TCP protocols, in: Proceedings of 2nd Workshop on Practical Use of Coloured Petri Nets and Design/CPN, 1999, pp. 21–40.
[28] J.L. Rasmussen, M. Singh, Designing a security system by means of coloured Petri nets, in: Proceedings of 17th International Conference on Application and Theory of Petri Nets (LNCS 1091), 1996, pp. 400–419.
[29] M. Kellner, et al., ISPW-6 software process example, in: Proceedings of the 1st International Conference on the Software Process, 1991, pp. 176–186.
[30] H. Yamaguchi, G.v. Bochmann, T. Higashino, Decomposing service definition in predicate/transition-nets for designing distributed systems, in: Proceedings of IFIP 23rd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE2003), Lecture Notes in Computer Science 2767, 2003, pp. 399–414.
[31] M. Kapus-Kolar, Compositional service-based construction of multi-party time-sharing-based protocols, IEICE Trans. Fundam. Electron., Commun. Comput. Sci. E86-A (9) (2003) 2405–2412.
[32] H.F. Korth, A. Silberschatz, Database System Concepts, McGraw-Hill, 1991.
[33] D.Y. Chao, D.T. Wang, A synthesis technique of general Petri nets, J. Syst. Integrat. 4 (1) (1994) 67–102.
[34] Xerces (Perl XML Parser). Available from: <http://xml.apache.org/xerces-p/>.
[35] M. Wang, B. Li, Z. Li, sFlow: Towards resource-efficient and agile service federation in service overlay networks, in: Proceedings of 24th International Conference on Distributed Computing Systems (ICDCS2004), 2004.
[36] H. Yamaguchi, K. El-Fakih, A. Hiromori, T. Higashino, A formal approach to design optimized multimedia service overlay, in: Proceedings of 15th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2005), 2005, pp. 57–62.

Hirozumi Yamaguchi received his B.E., M.E. and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996 and 1998, respectively. He is currently an assistant professor at Osaka University. His research interests are design and implementation of distributed systems and wired and wireless communication protocols.

Khaled El-Fakih received his B.S. and M.S. degrees in Computer Science from the Lebanese American University and his Ph.D. in Computer Science from the University of Ottawa in 2002. He worked as a graduate fellow at the IBM Toronto Laboratory in 1997 and as a verification engineer at Cambrian Systems Corporation (a Nortel Company) in 1998. He is currently an assistant professor at the American University of Sharjah. His current research interests are in automating the design of distributed systems, test development from formal specifications, and fault diagnosis of distributed systems.

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

Teruo Higashino received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Osaka, Japan, in 1979, 1981 and 1984, respectively. He joined the faculty of Osaka University in 1984. Since 1999 he has been a Professor, and currently he belongs to Graduate School of Information Science and Technology, Osaka University. His current research interests include distributed systems, mobile computing, communication protocols and intelligent transportation systems. He is a senior member of IEEE, a fellow of Information Processing Society of Japan and a member of IFIP TC6/WG 6.1.