ELSEVIER

Journal of Systems Architecture 44 (1998) 955–970

JOURNAL OF SYSTEMS ARCHITECTURE

Petri net-based analysis on object assignment in distributed object-oriented systems

Wen-Tsung Chang [a,1], Chien-Chao Tseng [b,*], Wen-Kuang Chou [c,2]

[a] *Multimedia Laboratory, Special Systems Division, Institute for Information Industry, Taipei, Taiwan, ROC*

[b] *Department of Computer Science and Information Engineering, Institute of Computer Science and Information Engineering, National Chiao-Tung University, 1001 Ta Hsueh Road Hsinchu 30050 Taiwan, ROC*

[c] *Department of Computer Science and Information Management, Providence University, Shalu, Taiwan, ROC*

Received 19 January 1996; revised 13 December 1996; accepted 19 May 1997

Abstract

Object-oriented programming [9], which treats objects as processes in execution, has shown significant effectiveness in distributed systems. This effectiveness is greatly influenced by how objects are assigned to nodes. In this paper, we present a colored generalized stochastic Petri net (CGSPN) model to analyze the behavior of object invocations when an assignment strategy is applied. The effectiveness of an object assignment is also analyzed by our CGSPN model. Moreover, this analysis provides guidelines to develop an efficient object assignment strategy. [4–8]

Keywords: Petri net; Distributed Systems; Assignment strategy; Object-oriented programming

1. Introduction

Distributed object-oriented systems are composed of number of heterogeneous or homogeneous processing nodes that are linked to an interconnection network (see Fig. 1). Objects in nodes cooperate to accomplish a given task, and objects in different nodes interact with each other via invocations [1,2]. However,

* Corresponding author. E-mail: cctseng@csie.nctu.edu.tw

[1] E-mail: wtchang@iii.org.tw

[2] E-mail: wkchou@simon.pu.edu.tw

PII: S 1 3 8 3 - 7 6 2 1 (9 7) 0 0 0 4 9 - 0

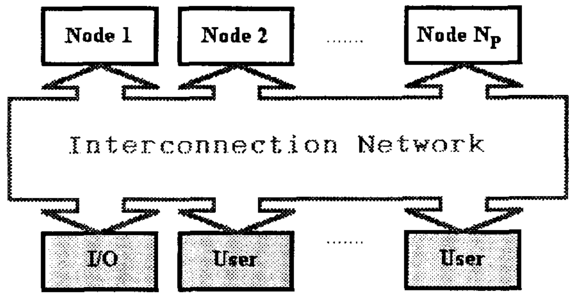


Fig. 1. Abstract model of distributed systems.

the invocation overhead between nodes is a major bottleneck that affects overall performance. To minimize such overhead, we should first analyze the behavior of objects handled in a distributed manner.

There are two approaches to modeling the behavior of distributed object-oriented systems: the queueing networks (QNs) theory and the generalized stochastic Petri net (GSPN) model. The QNs [10–13] approach is to model objects as servers with queues processing incoming requests, whereas the GSPN [14–20] approach is to model objects with states, transitions and the notation of stochastic process. The GSPN model gives a better description of how transitions, concurrency, and synchronization behave in distributed systems. However, most of them were designed to model the internal behavior of objects in specific languages [17–20]. In our study, we intend to analyze the communication overhead of a distributed object-oriented system. Therefore, we focus on the behavior of interaction among objects rather than the internal behavior of objects. We develop a generalized modeling technique based on the colored GSPN (CGSPN) model since the CGSPN model can clearly describe the behavior of distributed object-oriented systems [21,22]. Moreover, we further use our model to analyze the factors to the effectiveness of an assignment strategy in a distributed object-oriented system.

The rest of this paper is organized as follows: Section 2 introduces our CGSPN model. Section 3 further describes our CGSPN model in a semantic structure. Section 4 verifies our CGSPN model and discusses the effectiveness evaluation of an assignment strategy based on our CGSPN model, and Section 5 presents our conclusions.

2. The CGSPN modeling of object invocations

2.1. An abstract object model

Before describing our CGSPN model, we should first define an abstract object model. From the viewpoint of programming languages, Snyder defined an abstract model based on the following concepts [23]:

- An object explicitly embodies an abstraction (class) characterized by services or operations (methods).
- Operations can be generic; an operation can be uniformly performed with visibly different behaviors on a range of objects (polymorphism).
- Objects can be classified by their services, forming a class hierarchy.

- Objects can share the same implementation, either in full (*class instances*) or in part (*class inheritance*).

To analyze the execution behavior of objects, we further define an object as follows:

- Objects are units of execution, with independent storage containing local variables and associated operations (methods) that maintain these variables.
- Each object belongs to a class, i.e., an object is an instance of a certain class.
- An object is activated by incoming invocations. If the required method is not found in its local storage, the object will by-pass this invocation up to its superclass, until the required method is found, or a failure message is returned.

To simplify the analysis of our abstract object model, we make some assumptions about the behavior of object invocations:

1. An object can only execute one invocation at a time, that is, an object has a queue collecting all types of method invocations. Invocations are executed in FCFS (First-Come-First-Served) order, without preemption or priority.
2. To ensure consistency in execution, data access in an object is a critical section managed by an operating system, and programs in this operating system are assumed to be deadlock-free.
3. The arrivals of invocations are Poisson processes.

Using the above assumptions, we have proposed a five-phase invocation protocol to describe the interaction behavior of objects handled in a distributed manner [3]:

Phase 1: Start invocation (Issue).
Phase 2: Route invocation to target object (Transmit).
Phase 3: Carry out the appropriate computations (Execute).
Phase 4: Branch to nested invocations and continue execution (Branch).
Phase 5: Return (Return).

This five-phase protocol is developed based on the four-phase protocol which indicates the operation of object invocations by Tomlinson et al. [24]. When a source object activates an invocation to a target object in phase 1, this invocation travels through nodes in phase 2 if the source and target objects are not located in the same node. In phase 3, the target object performs the operations specified in the associated method. If such invocation activates further invocations, the protocol enters phase 4, which recursively repeats phases 1–5, until further invocations have been completed; the target object returns the results in phase 5 after the execution is finished.

Because of its generality, this five-phase protocol can be applied to both the statically typed programming languages, like Eiffel and C++, and the dynamically typed programming languages, like Smalltalk-80 and Common Lisp Object System (CLOS). Moreover, this description can also be applied to develop our analytical model.

2.2. The CGSPN invocation model

As mentioned earlier, CGSPN can be applied effectively to model distributed object-oriented systems since it clearly describes the dynamic behavior of invocations with different colors of tokens. In this section, we propose a model based on CGSPN to analyze the dynamic behavior of object invocations.

In a distributed system, objects are assigned to nodes to perform certain tasks in parallel by an assignment strategy. An assignment strategy can be represented with a mapping function. The mapping function

Map is depicted as *Map: Obj* → *Node*, mapping function of an assignment strategy, where *Obj* is the set of objects, and *Node* is the set of nodes.

This function maps an object to a certain node. Our CGSPN model is based on the description of nodes since we want to describe the behavior of objects among nodes. The CGSPN model for a particular node N_i in a distributed system, denoted as ND_i, is defined as follows:

Definition 1. CGSPN $\mathrm{ND}_i = (P^i, T^i, A^i, L^i, X\Lambda^i, M_0^i)$, where $P^i = \{P1_i, P2_i, P3_i\}$ is the set of places (states of invocation behavior), $T^i = \{t1_i, t2_{i,i,0}, \ldots, tr_{i,n-1}, tm_{i,0}, \ldots, tm_{i,n-1}\}$ the set of transitions (n = number of nodes in target system), $A^i \subseteq \{(P^i \times T^i)\}\{\cup\{(T^i \times P^i)\}$ the set of arcs connecting places and transitions, $L^i = \{\lambda, \mu_i^m\}$ the set of method m firing rates associated with timed transitions, $X = \{c_1, \ldots, c_k\}$ the set of token colors (k = number of methods in the class hierarchy), $\Lambda^i : P^i \in X^*$ the function indicating the numbers and colors of tokens in a given place, and M_0^i is the initial marking of a node N_i.

Tokens in different places stand for states of invocation behavior according to our protocol. A transition is enabled when a sufficient number of tokens are accumulated in all its input places. When a transition is enabled, it may fire immediately, or after a period of time. The duration of time period is determined by the set of firing rates L^i. Firing a transition may change the color of a token by firing rules. We assume that the firing rules are determined by a source program, and whenever an object method is invoked, its codes can be found in its local processing node.

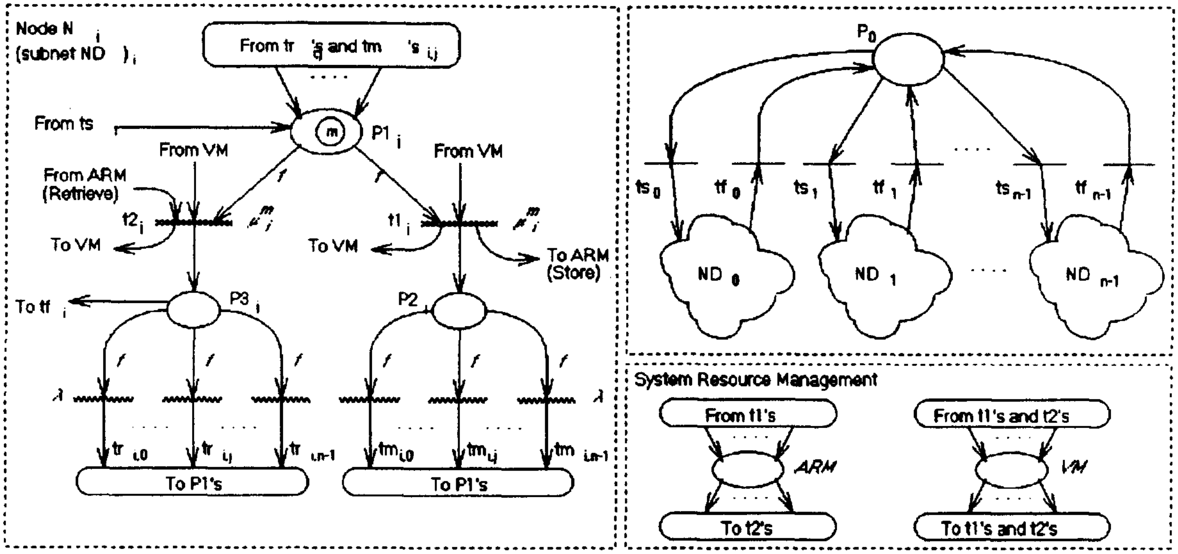
The CGSPN model ND_i only represents the description of a node. In general, an distributed object-oriented program consists of several nodes. Moreover, we need constructs to control invocation activation and variable access. Hence the CGSPN model of an object-oriented program, denoted as DS, can be depicted as follows:

Definition 2. CGSPN $\mathrm{DS} = (P, T, A, L, X, \Lambda, M_0)$, where $P = \{\bigcup_{i=0}^{n-1} P^i\} \cup \{P_0, \mathrm{ARM}, \mathrm{VM}\}$, $T = \bigcup_{i=0}^{n-1} (T^i \cup \{ts_i\} \cup \{tf_i\})$, $A \subseteq \{(P \times T)\} \cup \{(T \times P)\}$, $L = \bigcup_{i=0}^{n-1} L^i$,

$$\Lambda : \begin{cases} \left(\bigcup_{i=0}^{n-1} P^i\right) \cup \{P_0\} \to X^*, \\ \mathrm{ARM} \to \{c_a\}^*, \quad M_0 = \{\Lambda(P_0), \Lambda(\mathrm{ARM}), \Lambda(\mathrm{VM}), M_o^0, \ldots, M_o^{n-1}\}, \\ \mathrm{VM} \to \{c_o\}^*, \end{cases}$$

where ARM is a place to store activation records of invocations, VM a place to store object variables, c_a a token of an activation record, and c_o is a token of variables in an object.

The detailed description of the CGSPN model DS is shown in Fig. 2. Function f is a probability function defined by source program to enable/disable the firing of a transition. This model consists of the descriptions of classes (we use the term "subnet ND_i" as the CGSPN model of a node N_i). The DS model also includes additional places VM and ARM. Tokens in place VM are used as the synchronization mechanisms for variable accesses. A token with color c_o in VM represents the variables of an object. Tokens with color in place ARM represent the activation records of invocations. These records are used to hold the context of parent invocations. Hence X' includes X, c_o and c_a. Moreover, tokens in VM and ARM are managed by the operating system. In an object-oriented program, objects are activated by invocations. Every invocation



(a) The node model ND i (b) The system model DS

Fig. 2. CGSPN description of the activity of a node N_i and the system resources.

represents a token in the DS model, and is initially placed in P_0. Thus, the initial marking $\Lambda(P_0)$ is determined by the initial object assignment in a distributed system. When the execution starts, these tokens enter the places $P1_i$'s of associated subnets through transitions ts_i's according to the codes of the program.

Without loss of generality, assume that a token tk_0 is in the place $P1_i$ of subnet ND_i and it invokes an invocation of method m to object O_{j1}. This token tk_0, along with a variable token released from VM, causes the firing of transition $t1_i$ after a time duration (phase 1: *Issue*). This duration is assumed to be an exponential distribution with parameter μ_i^m which depends on the type of method m and the target class i. Moreover, a new token tk_i, which replaces tk_0, enters place $P2_i$.

At this moment, token tk_i enables all transitions $tm_{i,j}$'s, as shown in Fig. 2. However, as tk_i is transmitted to the target place $P1_j$ of subnet (phase 2: *Transmit*), only one transition will fire. The firing of transition is determined by function f, where $f: T \rightarrow \{0, 1\}$. The duration time for transmission is also an exponential distribution with parameter λ.

However, tk_i has to wait in $P1_j$ of subnet ND_j if no token of object variables is released from VM (i.e., the target object is accessed by other object, which causes mutual exclusion of critical section in object variables). When a token in VM is released, both transitions $t1_i$ and $t2_i$ can be enabled. At this time, firing function f determines whether token tk_i completes execution or further activates non-local invocations. For the former case, that is, tk_i completes execution, $f(t2_i)$ will become one and transition $t2_i$ will fire accompanied with an associated activation record retrieved from ARM. After a duration time of execution (phase 3: *Execute*), tk_i enters place $P3_j$. At this time, function f causes transition $tr_{j,i}$ to fire and tk_i traverses back to its parent subnet ND_i (phase 5: *Return*).

For the latter case, that is, tk_i activates further non-local invocations, $t1_i$ will fire (phase 3: *Execute*) and tk_i is stored in ARM as a token of activation record. A new token, namely tk_j, indicating further non-local invocation, replaces tk_i (phase 4: Branch) and repeats the process described above. After further non-local invocation completes, token tk_i, retrieved from ARM, enters place $P1_j$ of subnet ND_j to continue the remaining process. The process of *Branch* phase repeats until all the non-local invocations complete. At this time, *Return* phase starts and token tk_i goes back to parent subnet ND_i through $P3_j$ and transition $tr_{j,i}$.

The proposed DS model can be constructed from the codes of an object-oriented program and the associated class hierarchy. In Section 3 we present the semantic constructs of our GSPN model to assist the performance analysis.

3. Semantic constructs of CGSPN model DS

As mentioned earlier, we have proposed a CGSPN model to describe the behavior of object invocations. In our proposed model, every token of invocation has an attribute to show its own behavior. This attribute can be represented by an attribute function, denoted as I_{attr}. This function is defined as follows:

$$I_{\text{attr}} : Token \rightarrow \langle Obj \times Obj \times Method \times (Token \cup \{null\})\rangle, \tag{1}$$

where *Token* is the set of tokens in places $P_0, P1_i$'s, $P2_i$'s and $P3_i$'s $(i = 0, \ldots, n-1)$, *null* the empty set in places $P_0, P1_i$'s, $P2_i$'s and $P3_i$'s, *Obj* the set of objects, and *Method* is the types of method invocations.

The values of these functions are determined by the source program. Moreover, the last term *Token* in function I_{attr} indicates the token of parent invocation.

If an object O_{i1} is assigned to node N_i, the mapping function can be defined as $Map(O_{i1}) = N_i$. Moreover, a token tk_0 in P_0 has an attribute which indicates the initial status of program execution, such as $I_{\text{attr}}(tk_0) = \langle \_O_{i1}, m, null\rangle$, where tk_0 is an invocation of method m to object O_{i1}. When the program starts execution, tk_0 in P_0 moves to $P1_i$ of subnet ND_i through transition ts_i. After tk_0 enters place $P1_i$, transitions $t1_i$ and $t2_i$ are enabled. The firing of $t1_i$ and $t2_i$ is determined by the firing function f. This function decides whether such invocation returns back to parent subnet or further activates inter-node invocations, that is $f(t1_i) + f(t2_i) = 1$, for all $i, i = 0, \ldots, n-1$. If $f(t2_i)$ is one, that is, the invocation completes execution, $t2_i$ will fire by retrieving an associated token of activation record from ARM and the resulting token will enter $P3_i$ (see Fig. 2). However, if $f(t1_i)$ is one, $t1_i$ will fire since and tk_0 is stored in ARM as a token of activation record, which will be discussed later. At the same time, a new token tk_i is created in $P2_i$ to represent further invocation.

In general, the semantic of transition $t1_i (i = 0, \ldots, n-1)$ is as follows:

For a token tk selected from $P1_i$ of subnet ND_i, and $I_{\text{attr}}(tk) = \langle O_{i1}, O_{j1}, m, ptk\rangle$, where $Map(O_{i1}) = N_i$

$$\begin{aligned} &\text{if } f(t1_i) = 1 \quad tk \rightarrow tk', \quad \text{and} \quad I_{\text{attr}}(tk') = \langle O_{j1}, O_{k1}, m', tk\rangle, \text{where} \\ &\qquad tk, tk' \in Token,\ ptk \in Token \cup \{null\}. \end{aligned} \tag{2}$$

Whenever a token of an invocation moves from $P1_i$ to $P2_i$ through $t1_i$, its attribute determines the value of firing function. This firing function can be defined as follows:

Let tk be the token in $P2_i$ of *subnet* ND_i, $I_{\text{attr}}(tk) = \langle O_{i1}, O_{j1}, m, ptk\rangle, N_i = Map(O_{i1})$ and $ptk \in Token \cup \{null\}$,

$$
\begin{aligned}
&f(tm_{i,j}) = 1 \quad \text{if } N_j = Map(O_{j1}),\\
&f(tm_{i,j}) = 0 \quad \text{otherwise},\\
&\quad \text{where } \sum_{i=0}^{n-1} f(tm_{i,j}) = 1, \quad \text{for all } i, i = 0, \cdots n-1.
\end{aligned}
\tag{3}
$$

If $f(t2_i)$ is one, the firing function f also determines the return path via transitions $tr_{j,i}$ by the attribute of tk_0, which will be discussed later.

As mentioned in our five-phase protocol, two phases need to access tokens in VM: *Issue* and *Execute*. The access control occurs in the transitions $t1_i$'s and $t2_i$'s. To fire these transitions, the corresponding variable tokens should be found in VM. Each object is associated with a token in VM. The attribute function V_{attr} for the tokens in VM is

$$V_{\text{attr}} : Token1 \rightarrow Obj, \tag{4}$$

where $Token1$ is set of tokens in place VM .

Each object O_i thus has a token v_i in VM with attribute $V_{\text{attr}}(v_i) = O_i$. The conditions required to enable transitions $t1_i$'s and $t2_i$'s can be described as

Let tk be a token of invocation, and $I_{\text{attr}}(tk) = \langle O_{i1}, O_{j1}, m, ptk \rangle$,

For transitions $t1_i$ and $t2_i$: $[tk \in \Lambda(P1_i)]$ and $[O_{j1} \in S_{\text{VM}}]$,

where $S_{\text{VM}} = \{O_i | \quad \forall v_i \in \text{VM}, V_{\text{attr}}(v_i) = O_i\}, tk \in Token, ptk \in Token \cup \{null\}$, and

$$v_i \in Token1. \tag{5}$$

When the condition is satisfied, the associated transition retrieves the variable token from VM and, after it has fired, the transition releases this token back to VM.

A token in ARM represents the activation record of an invocation. ARM could be a stack or hash table. Activation records can be retrieved by the function Acc, denoted as

$$Acc : Token \rightarrow Token \cup \{null\}. \tag{6}$$

Initially, we assume that $Acc(tk) = null$ for any token tk in P_0. Whenever the transition $t1_i$ fires, token tk of method m in $P1_i$ will be stored in ARM and create a token of child method m' invocation tk' in $P2_i$. Hence the semantic of transition $t1_i$ can be depicted as $Acc(tk') = tk$, where

$$I_{\text{attr}}(tk) = \langle O_{i1}, O_{j1}, m, ptk \rangle \quad \text{and} \quad I_{\text{attr}}(tk') = \langle O_{j1}, O_{k1}, m', tk \rangle. \tag{7}$$

As stated previously, if $f(t2_i)$ is one, transition $t2_i$ will try to retrieve a token from ARM via function Acc. The resulting token then enters $P3_i$ and the function f determines the designated transition which the resulting token will traverse, either back to P_0 or its upper-level subnet. Function f is defined as

Let tk' be a token in $P3_i$ *of subnet* ND_i, $I_{\text{attr}}(tk') = \langle O_{j1}, O_{k1}, m', tk, N_x \rangle = Map(O_{k1})$ and $N_j = Map(O_{j1})$,

$$
\begin{aligned}
&f(tr_{x,j}) = 1 \quad \text{if } Acc(tk') = tk \text{ and } I_{\text{attr}}(tk) = \langle O_{i1}, O_{j1}, m, ptk \rangle,\\
&f(tr_{x,j}) = 0 \quad \text{otherwise}.\\
&f(tf_x) = 1 \quad \text{if } Acc(tk) = null,
\end{aligned}
$$

$f(tf_x) = 0$ otherwise,

$$\text{where} \sum_{i=0}^{n-1} f(tr_{x,j}) + f(tf_x) = 1 \quad \text{for all } x \text{ and } j, x, j = 0, \ldots, n-1. \tag{8}$$

After the designated transition has been determined, the resulting token tk' is replaced by the token tk in transition $t2_i$ if $Acc(tk') = tk$. Otherwise, tk' remains unchanged if $Acc(tk')$ is *null*.

With the above constructs, we can formally describe the behavior of an object invocation. In Section 4, we will prove that our semantic constructs are correct in the DS model and analyze the effectiveness of an assignment strategy by our DS model.

4. Discussions about the CGSPN model DS

4.1. Correctness of the semantic constructs

In Section 2 we have shown the five-phase protocol of an invocation. This five-phase protocol can also be viewed as a syntax term *Invoc(i,j,m)*, defined as follows:

```
Invoc(i,j,m)::= Issue(i,j,m) Transmit(i,j,m) Execute(i,j,m)
             Branch(i,j,m) Return(i,j,m),
Execute(i,j,m )::= Lookup(i,j,m) M-Execute(i,j,m)
Branch(i,j,m) ::=∅ | Invoc(j,k,m′ ) R(i,j,m),
R(i,j,m) ::= ∅ | M-Execute(i,j,m) | Branch(i,j,m) | M-Execute(i,j,m)
             Branch(i,j,m),
```

where i, j, k are source and target object indices, m, m' types of methods, *Issue( )* an atomic operation for phase *Issue*, *Transmit( )* an atomic operation for phase *Transmit*, *Execute( )* a composite operation for phase *Execute*, *Lookup( )* an atomic operation for method lookup in phase *Execute*, *M-Execute( )* an atomic operation for code execution in phase *Execute*, *Branch( )* a composite operation for phase *Branch*, *R( )* a composite operation for phase *Branch*, and *Return( )* an atomic operation for phase *Return*.

Using this syntax, we deduce three lemmas to prove the correctness of our constructs.

Lemma 1. *Semantics of variable accesses is correct for all types of invocations.*

The accesses of tokens in VM occur at *Execution* phase since the variables of target object could be collected and updated. The accesses are caused by the firing of transitions $t1_i$'s or $t2_i$'s. Suppose an invocation of method m to object O_{j1} of node N_j activates, a token tk_0 enters place $P1_j$ of subnet ND_j. According to our semantic constructs, condition (5) states that if tk_0 is in place $P1_j$ and variable token of O_{j1} is contained in VM, both $t1_j$ and $t2_j$ are enabled and one of them is fired by function f. Thus, we can see that the semantic constructs of VM are correct.

In Lemma 2 we prove that the semantic constructs of accessing ARM are also correct. Since the accesses of tokens in place ARM occur only at transitions $t1_i$'s and $t2_i$' s, it is thus sufficient to show that the semantics is correct in transitions $t1_i$'s and $t2_i$'s.

Lemma 2. *Semantics of accessing ARM is correct for all types of invocations.*

Invocations can be classified into two types: one is initial invocations contained in the main program, and the other is activated by other invocations. For the first type of invocations, the initial values of *Acc( )* are *null*, while the values for the second type are not. Suppose that an invocation of method m to object O_{j1} of node N_j, denoted as *Invoc(i1, j1, m)*, is activated and a token tk_i enters place $P1_j$ of subnet ND_j. At this time, both $t1_j$ and $t2_j$ are enabled if condition (5) is satisfied. The behavior of tk_i depends on the following values of $f(\ )$:

Case 1 ($f(t2_j) = 1$): $t2_j$ fires by retrieving a token from ARM. If tk_i is an initial invocation, $Acc(tk_i)$ is *null* and function f causes the firing of transition tf_j by statement (8), that is

$$f(tf_i) = 1 \qquad \because Acc(tk_i) = null, \quad N_j = Map(O_{j1}).$$

Token tk_i will go back to P_0 through transition tf_j and terminates its execution.

If tk_i is the second type of invocations, $Acc(tk_i)$ is not *null,* and suppose that it is tk_0. Function f causes the firing of transition $tr_{j,i}$ by statement (8), that is

$$f(tr_{j,i}) = 1 \quad \because Acc(tk_i) = tk_0,\ I_{\mathrm{attr}}(tr_0) = \langle O_{i1}, O_{j1}, m, ptk\rangle,\ Map(O_{i1}) = N_i.$$

Token tk_0 will replace tk_i and go back to $P1_i$ of subnet ND_i through transition $tr_{j,i}$ to continue the remaining process of tk_0.

Case 2 ($f(t1_j) = 1$): $t1_j$ fires to further activate a non-local invocations. Suppose in node N_j, tk_i further invokes method m' to object O_{k1} of node N_x, denoted as *Invoc* $(j1, k1, m')$, where $N_j \neq N_x$. At this time, tk_i is stored in ARM and a token tk_j of $Invoc(j1, k1, m')$ is created, and $Acc(tk_i)$ is set as tk_i. After the firing of transition $t1_j$, token tk_j enters $P1_x$ of subnet ND_x through $tm_{j,x}$.

When $Invoc(j1, k1, m')$ completes, tk_j enters $P3_x$ of subnet ND_x through transition $t2_x$ (since $f(t2_x)$ is one). By statement (8) function f causes the firing of transition $tr_{x,j}$ since $Acc(tk_j)$ is tk_i, not *null.* Therefore, tk_i replaces tk_j, and correctly goes back to $P1_j$ of subnet ND_j.

After $Invoc(i1, j1, m)$ completes *Branch* phase, $f(t2_j)$ becomes one and repeats the process of case 1. Thus these semantic constructs are correct for all types of invocations.

Since the *Branch* phase includes a composite operation *R( )* in syntax, we need to prove that *R( )* works correctly for all types of invocations.

Lemma 3. *The proposed semantics is correct in R( ).*

By the definition of invocations, *R( )* represents the remaining process of an arbitrary invocation. There are four cases for the derivation of *R( )*: ∅, *M-Execution( )*, *Branch( )* and *M-Execution( ) Branch( )*. Since the time duration of *Execution( )* can be zero, ∅ is thus a special case of *M-Execution( )*, and *Branch( )* is also a special case of *M-Execution( ) Branch( )*. For a subnet ND_i, the first two cases happen when $f(t2_i)$ is one and a token of an invocation will directly enter $P3_i$ by firing transition $t2_i$. At this time, *R( )* completes its process and our semantic constructs are thus correct.

The latter two cases happen when $f(t1_i)$ is one, $t1_i$ will fire and activate further invocation. Since the firing of $t1_i$ indicates the process of a further *Branch( )*, which causes another process of *R( )*, our constructs are thus also correct.

With the preceding lemmas, we can prove the correctness of our semantics with the following theorem.

Theorem 1. *The proposed semantic constructs are correct for all types of invocations.*

Proof. Without loss of generality, let us examine the behavior of an invocation which activates arbitrary levels of cascading non-local invocations, as shown in Fig. 3. Let us examine an *a*th level invocation, denoted as $Invoc(i_{a-1}, i_a, m_a)$, which is represented by a token tk_a in our DS model. This token indicates a non-local invocation of method m_a from object $O_{i_{a-1}}$ to object O_{i_a}, where $Map(O_{i_a}) = N_{x_a}$, $Map(O_{i_{a-1}}) = N_{x_{a-1}}$, and $N_{x_a} \neq N_{x_{a-1}}$.

We prove this theorem by induction on *a*.

(i) *Case* ($a = 1$): We discuss the behavior of $Invoc(i_0, i_1, m_1)$ with the five-phase protocol.

(a) *Issue, Transmit* and *Execution*: By Lemmas 1 and 2, since the access of VM and ARM works correctly, the process of these phases is correct.

(b) *Branch*: After the process of *Execution* phase, token tk_1 enters $P1_{x_i}$ of subnet ND_{x_1}. If tk_1 does not activate further invocation, the *Branch* phase will be skipped and *Return* phase starts directly. If tk_1 does, $f(t1_{x_1})$ becomes one and $t1_{x_1}$ fires. At this time, tk_1 is stored in ARM and token tk_2, which represents the invocation $Invoc(i_1, i_2, m_2)$, enters $P2_{x_1}$ of subnet ND_{x_1} (by Lemma 2). *Acc(tk₂)* is then set to be tk_1. After $Invoc(i_1, i_2, m_2)$ is completed, token tk_1 replaces tk_2 and enters $P1_{x_1}$ of subnet ND_{x_1} through transition tr_{x_2,x_1} by statement (8) ($\because Acc(tk_2) = tk_1$). Thus, the process of *Branch* phase is correct.

(c) *R( )*: By Lemma 3, since there is no further invocation in $Invoc(i_0, i_1, m_1)$, we know that our semantics is correct in the process of *R( )* for all types of invocations.

(d) *Return*: After the execution of *Branch*, $t2_{x_1}$ fires and tk_1 enters $P3_{x_n}$ of subnet ND_{x_n}. By Lemma 2 and statement (8), $Invoc(i_0, i_1, m_1)$ thus completes correctly and token of invocation moves back to P_0 or its parent subnet.

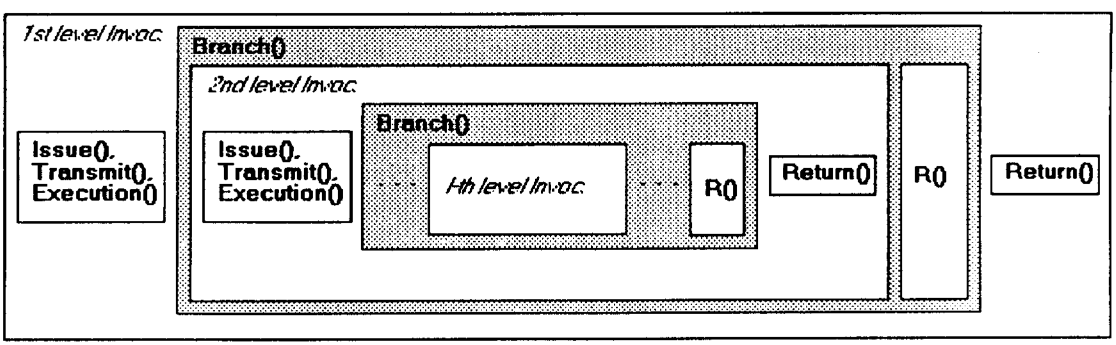


Fig. 3. Structure of an object invocation.

(ii) *Case* ($a = n$): Assume that our constructs work correctly for n-level invocation $Invoc(i_{n-1}, i_n, m_n)$.
(iii) *Case* ($a = n + 1$): Suppose that $Invoc(i_{n-1}, i_n, m_n)$ further activates an invocation $Invoc(i_n, i_{n+1}, m_{n+1})$. By case ($a = 1$) $Invoc(i_n, i_{n+1}, m_{n+1})$ works correctly.

Thus we conclude that our DS model can describe all kinds of invocations in an object-oriented program. □

Theorem 1 can formally prove the correctness of our semantic constructs. In Section 4.2, we formulate a performance model of assignment strategies and derive the guidelines for designing effective object assignment strategies.

4.2. Effectiveness of assignment strategies

The effectiveness of an assignment strategy can be measured by the communication and computation costs, which are denoted as C_{comm} and C_{comp}, respectively. By examining our DS model, it is obvious that C_{comm} is incurred by the *Transmit* and *Return* phases (time duration of firing transitions $tm_{i,j}$'s, and $tr_{i,j}$'s), while C_{comp} by the *Execute* phase (time duration of firing transitions $t1_i$'s and $t2_i$'s). For a distributed object-oriented system, these costs play an important role for the overall system performance. Hence, an effective assignment strategy should minimize these costs.

In this section, we use the analytical measurement to measure the costs based on our DS model. We first assume the following probability values for firing function f as (m indicates type of method):

1. $p\{f(t2_i) = 1\} = q$, and $p\{f(t1_i) = 1\} = (1 - q)$,
2. $p\{f(tm_{i,j}) = 1\} = p^m_{i,j}$, where $\sum_{j=0}^{n-1} p^m_{i,j} = 1$,
3. $p\{f(tf_i) = 1\} = r_m$,
4. $p\{f(tr_{i,j}) = 1\} = t^m_{i,j}$, where $\sum_{j=0}^{n-1} t^m_{i,j} + r_m = 1$.

With the definitions of DS model stated in Section 2, we assume that if the code of the associated method cannot be found locally, this invocation will be by-passed to other nodes as a non-local invocation. The probability of this by-passed invocation is defined as p_{bypass}. Thus the mean value of $f(t2_i)$ becomes $q/(1 + p_{\text{bypass}})$, which is denoted as q'. With the above probability values, we can transform our DS model into Markov chains to evaluate C_{comm} and C_{comp}. The states of these Markov chains are defined as follows:

$$S = (\omega_{a,b})_{n \times 3},$$

where $\omega_{a,b} = (\sigma^1_{a,b}, \sigma^2_{a,b}, \ldots, \sigma^k_{a,b})$ is the set of color tokens in the place of subnet ND_a, k = number of colors and $\sigma^m_{a,b}$ is the number of tokens of color m in the place of subnet ND_a, $\sigma^m_{a,b} \geqslant 0$, m indicates type of method.

In Fig. 4, we give an example to illustrate the states and the transitions of the Markov chains for an invocation. This example indicates an invocation of a method m from node N_i to node N_j which further activates a method m' to node N_k. As shown in Fig. 4, there are seven states, namely $S1 - S7$. The contents of these seven states are expressed as follows:

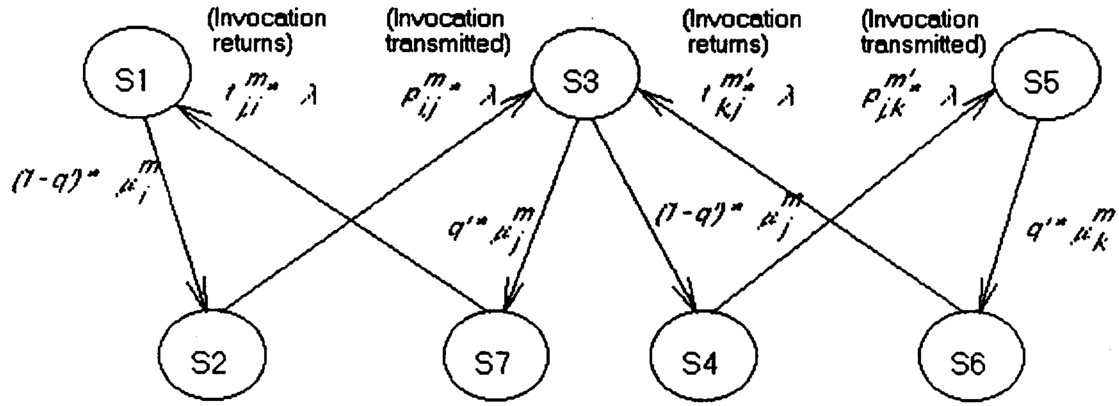


Fig. 4. Markov chains for an invocation m issued from node N_i to node N_j, $(S1, \ldots, S7 \in S)$.

$$S1 = \begin{bmatrix} & \vdots & \\ (\ldots, \sigma^m_{i,1}, \ldots) & (\ldots, \sigma^m_{i,2}, \ldots) & (\ldots) \\ (\ldots, \sigma^m_{j,1}, \ldots) & (\ldots, \sigma^{m'}_{j,2}, \ldots) & (\ldots, \sigma^m_{j,3}, \ldots) \\ (\ldots, \sigma^{m'}_{i,1}, \ldots) & (\ldots) & (\ldots, \sigma^m_{i,1}, \ldots) \\ & \vdots & \end{bmatrix},$$

$$S2 = \begin{bmatrix} & \vdots & \\ (\ldots, \sigma^m_{i,1} - 1, \ldots) & (\ldots, \sigma^m_{i,2} + 1, \ldots) & (\ldots) \\ & \vdots & \\ (\ldots, \sigma^m_{j,1}, \ldots) & (\ldots, \sigma^{m'}_{j,2}, \ldots) & (\ldots, \sigma^m_{j,3}, \ldots) \\ (\ldots, \sigma^{m'}_{i,1}, \ldots) & (\ldots) & (\ldots, \sigma^m_{i,1}, \ldots) \\ & \vdots & \end{bmatrix},$$

$$S3 = \begin{bmatrix} & \vdots & \\ (\ldots, \sigma^m_{i,1}, \ldots) & (\ldots, \sigma^m_{i,2}, \ldots) & (\ldots) \\ (\ldots, \sigma^m_{j,1} + 1, \ldots) & (\ldots, \sigma^{m'}_{j,2}, \ldots) & (\ldots, \sigma^m_{j,3}, \ldots) \\ & \vdots & \\ (\ldots, \sigma^{m'}_{i,1}, \ldots) & (\ldots) & (\ldots, \sigma^m_{i,1}, \ldots) \\ & \vdots & \end{bmatrix},$$

$$S4 = \begin{bmatrix} & \vdots & \\ (\ldots,\sigma^{m}_{i,1},\ldots) & (\ldots,\sigma^{m}_{i,2},\ldots) & (\ldots) \\ (\ldots,\sigma^{m}_{j,1},\ldots) & (\ldots,\sigma^{m'}_{j,2}+1,\ldots) & (\ldots,\sigma^{m}_{j,3},\ldots) \\ & \vdots & \\ (\ldots,\sigma^{m'}_{i,1},\ldots) & (\ldots) & (\ldots,\sigma^{m}_{i,1},\ldots) \\ & \vdots & \end{bmatrix},$$

$$S5 = \begin{bmatrix} & \vdots & \\ (\ldots,\sigma^{m}_{i,1},\ldots) & (\ldots,\sigma^{m}_{i,2},\ldots) & (\ldots) \\ (\ldots,\sigma^{m}_{j,1},\ldots) & (\ldots,\sigma^{m'}_{j,2},\ldots) & (\ldots,\sigma^{m}_{j,3},\ldots) \\ & \vdots & \\ (\ldots,\sigma^{m'}_{i,1}+1,\ldots) & (\ldots) & (\ldots,\sigma^{m}_{i,1},\ldots) \\ & \vdots & \end{bmatrix},$$

$$S6 = \begin{bmatrix} & \vdots & \\ (\ldots,\sigma^{m}_{i,1},\ldots) & (\ldots,\sigma^{m}_{i,2},\ldots) & (\ldots) \\ (\ldots,\sigma^{m}_{j,1},\ldots) & (\ldots,\sigma^{m'}_{j,2},\ldots) & (\ldots,\sigma^{m}_{j,3},\ldots) \\ & \vdots & \\ (\ldots,\sigma^{m'}_{i,1},\ldots) & (\ldots) & (\ldots,\sigma^{m}_{i,1}+1,\ldots) \\ & \vdots & \end{bmatrix},$$

$$S7 = \begin{bmatrix} & \vdots & \\ (\ldots,\sigma^{m}_{i,1},\ldots) & (\ldots,\sigma^{m}_{i,2},\ldots) & (\ldots) \\ (\ldots,\sigma^{m}_{j,1},\ldots) & (\ldots,\sigma^{m'}_{j,2},\ldots) & (\ldots,\sigma^{m}_{j,3}+1,\ldots) \\ & \vdots & \\ (\ldots,\sigma^{m'}_{i,1},\ldots) & (\ldots) & (\ldots,\sigma^{m}_{i,1},\ldots) \\ & \vdots & \end{bmatrix}.$$

Since we only concern the computation of C_{comm} and C_{comp}, we eliminate the places $P2_i$'s and $P3_i$'s to simplify the analysis. The set of states S can thus be reduced to a new set of states, namely S', depicted as follows:

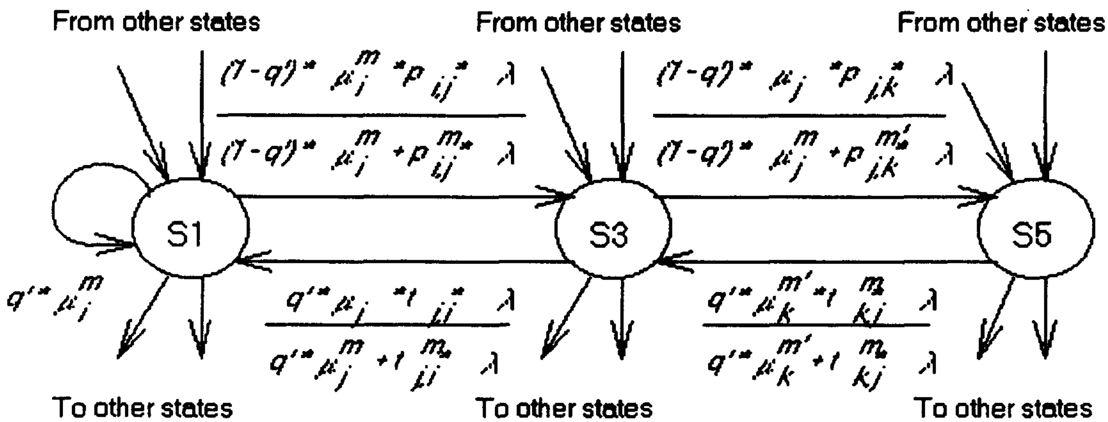


Fig. 5. Simplified Markov chains for the example in Fig. 4.

$$S' = (\omega_{a,1})_{n\times 1}.$$

The associated simplified Markov chains for the example stated in Fig. 4 are shown in Fig. 5.

With the above descriptions, we can construct the whole simplified Markov chains for a given assignment strategy. Moreover, based on these Markov chains, we can obtain the cost of an invocation of method m from node N_i to node N_j, denoted as $C_{\text{inv}}(i,j,m)$, as follows:

$$C_{\text{inv}}(i,j,m) = 2/\lambda + 1/\mu_i^m + 1/\mu_j^m. \tag{9}$$

Besides, in the main program, we can also compute the startup cost of an initial method m invocation in node N_i, denoted as $C_{\text{start}}(i,m)$

$$C_{\text{start}}(i,m) = 1/\mu_i^m. \tag{10}$$

Moreover, we can further obtain the average total cost of an object-oriented program, namely $\overline{C}_{\text{prg}}$, as

$$\overline{C}_{\text{prg}} = \sum_{l_r=0}^{\infty}\{p(l_r)l_r \sum_{i,j,m} C_{\text{inv}}(i,j,m)\} + \sum_{l_r=0}^{\infty}\{p(l_s)l_s \sum_{i,m} C_{\text{inv}}(i,m)\}, \tag{11}$$

where $p(l_s)$ is the probability of l_s startup invocations in the main program and $p(l_r)$ is the probability of l_r inter-node invocations running in the target system. From the above cost functions, we can observe from the Markov chains that C_{comp} comes from the duration of μ_i^m and C_{comm} comes from the duration of λ. We can also conclude that if the probability p_{bypass} decreases, that is, the probability of finding required method codes locally increases, q' also increases, and cost of invocations thus decreases.

In our DS model, the parameters $p_{i,j}^m$'s, $t_{i,j}^m$'s and λ can be determined by the number of nodes n and the topology structure G in the target system, while the other parameters can be determined by the assignment strategies. Therefore, $p_{i,j}^m$'s, $t_{i,j}^m$'s and λ can be depicted as $p_{i,j}^m(n,G)$, $t_{i,j}^m(n,G)$ and $\lambda(n,G)$. For a static assignment strategy, the parameters μ_i^m and q' are fixed since objects are assigned to nodes before execution.

However, for a dynamic assignment strategy, since objects are created, assigned or destroyed in run-time, μ_i^m and q' also vary in run-time. Hence, μ_i^m and q' can be viewed as time-varying functions, denoted as $\mu_i^m(t)$ and $q'(t)$.

To reduce $\overline{C}_{\text{prg}}$ caused by these by-passed invocations, there are two approaches: duplicating all the necessary method codes invoked in a node, or grouping objects with sub- or super-class relation in a node. For the first approach, the by-passed invocations can be eliminated, however the total space cost will be increased due to redundant code duplication. For the second approach, the space cost will be minimized, however the cost of inter-node invocations may not be minimized since objects that interact frequently are usually of different classes (without superclass or subclass relation). Therefore we should minimize the combined cost of space and by-passed invocations in designing an effective object assignment strategy.

5. Conclusions

In this paper we propose a model DS to describe the behavior of object invocations when an assignment strategy is applied. This model also depicts the detailed phase transitions for various kinds of object invocations. It should be noted that our model permits multiple invocations runs in parallel to simulate the behavior of a distributed object-oriented system.

There are many ways to measure the costs of a system. In this paper, we applied the analytical measurement by our CGSPN model DS for an object-oriented program in a distributed manner. In our five-phase protocol, we observed that the communication cost results from the *Transmit* and *Return* phases (caused by transitions $tm_{i,j}$'s, and $tr_{i,j}$'s), whereas the computation cost results from the *Execute*phase (caused by transitions $t1_i$'s and $t2_i$'s). Finally, we provided guidelines to evaluate a given assignment strategy. Such guidelines are helpful in designing an effective object assignment strategy.

References

[1] Wen-Tsung Chang, Chien-Chao Tseng, The Object Support via FIFO Links on Message-Passing Systems, 11th International Conference on Technology of Object-Oriented Languages and Systems, Santa Babara, 1993, pp. 231–238.

[2] Wen-Tsung Chang, Chien-Chao Tseng, Supporting Distributed Objects in FIFO-Based Message-Passing Systems, Journal of Object-Oriented Programming (2) (1995) 56–64.

[3] Wen-Tsung Chang, Chien-Chao Tseng, Analytical Modelling on Object Invocations and Assignment Strategies in Message-Passing Systems, International Computer Symposium, Hsinchu, Taiwan, December 1994, pp. 1203–1208.

[4] S.H. Bokhari, Assignment Problems in Parallel and Distributed Computing, Kluwer Academic Publishers, Dordrecht, 1987.

[5] Lo, Virginia Mary, Task Assignment in Distributed Systems, Ph.D. Thesis, University of Illinois, Urbana-Champaign, 1983.

[6] V.B. Gylys, J.A. Edwards, Optimal Partitioning of Workload for Distributed Systems, IEEE COMPCON'76, 1976, pp. 353–357.

[7] A.N. Tantawi, D. Towsley, Optimal Static Load Balancing in Distributed Computer Systems, Journal of Association for Computing Machinery 32 (2) (1985) 445–465.

[8] F. Ercal, Heuristic Approaches to Task Allocation for Parallel Computing, Ph. D. Dissertation, Ohio State University, 1988.

[9] B. Meyer, Object-Oriented Software Construction, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[10] L. Kleinrock, Queueing Systems – Volume I: Theory, Wiley-Interscience, New York, 1975.

[11] P.G. Harrison, N.M. Patel, Performance Modeling of Communication Networks and Computer Architectures, Addison-Wesley, Reading, MA, 1992.

[12] F. Baskett, K.M. Chandy, R.R. Muntz, F.G. Palacios, Open, closed, and mixed networks of queues with different classes of customers, Journal of Association for Computing Machinery 22 (2) (1975) 248–260.

[13] M.A. Marsan, G. Balbo, G. Conte, Performance Models of Multiprocessor Systems, The MIT Press, Cambridge, MA, 1986.

[14] J.L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[15] J.-S. Song, The Modeling, Analysis, and Design of Distributed Systems Based on Communicating Petri Nets, Ph.D. Dissertation, University of California, Berkeley, 1988.

[16] Y.U. Kim, S. Moon, Object-relationship diagrams for object-oriented modeling with concurrency feature, Microprocessing and microprogramming 33 (1991/1992) 207–221.

[17] J. Engelfriest, G. Leih, G. Rozenberg, Formalizing the behavior of parallel object-based systems by Petri nets, Semantics for Concurrency, Leicester 1990: Workshops in Computing, Springer, Berlin, 1990, pp. 204–221.

[18] D.N. Christodoubkis, Petri net semantics of smalltalk-80, Microprocessing and microprogramming 24 (1988) 267–272.

[19] R. Bastide, C. Sibertin-Blanc, P. Palanque, Cooperative Objects: A Concurrent Petri-Net Based Object-Oriented Language, Proceedings of IEEE Conference on Systems, Man and Cybernetics, vol. 3, 1993, pp. 286-291.

[20] Yang Kyu Lee, Sung Joo Park, OPNets: An object-oriented high-level petri net model for real-time system modeling, Journal of Systems and Software 20 (1993) 69–86.

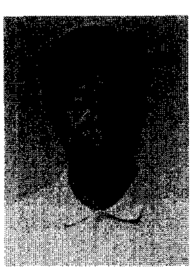
[21] J. Billington, Extensions to Coloured Petri Nets, PNPM' 89, Kyoto, Japan, December 1989, pp. 61–70.

[22] K. Jenson, Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, vol. 1, Springer, Berlin, 1992.

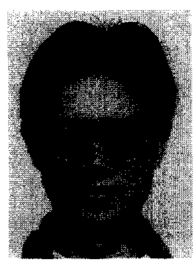
[23] A. Snyder, Modeling the C++ Object Model, European Conference on Object-Oriented Programming (ECOOP'91), 1991, pp. 1–20.

[24] C. Tomlinson, M. Scheevel, W. Kim, Sharing and organizational protocols in object-oriented systems, Journal of Object-Oriented Programming 2 (6) (1989) 25–36.

Wen-Tsung Chang is currently a senior engineer and a project manager in Multimedia Laboratory, Institute of Information Insustry, Taipei, Taiwan, ROC. He received his B.S, M.S and Ph.D. degrees in Computer Science from National Chiao-Tung University, in 1989, 1991 and 1995, respectively. His research interests are object-oriented design and programming, performance evaluation and distributed systems. Currently, he is interested in applications of multimedia and computer-aided architectural design.



Chien-Chao Tseng is currently a professor in the Department of Computer Science and Information Engineering at National Chiao-Tung University, Hsin-Chu, Taiwan. He received his B.S. degree in Industrial Engineering from National Tsing-Hua University, Hsin-Chu, Taiwan, in 1981; M.S. and Ph.D. degrees in Computer Science from the Southern Methodist University, Dallas, Texas, USA, in 1986 and 1989, respectively. His research interests are in Mobile Computing, and Parallel and Distributed Processing.



Wen-Kuang Chou is an associate professor of the Department of Computer Science and Information Management at Providence University, ShaLu, Taiwan, ROC. He received B.S. from the National Chiao-Tung University in Computer Engineering, M.S. in Computer Science from National Taiwan University, and Ph.D. from the Southern Methodist University in Computer Science and Engineering, in 1983, 1986, and 1991, respectively. From 1984 to 1986, he was a research assistant in Institute of Information Science at Academic Sinica, Taipei, ROC. Also from 1986 to 1987, he was an assistant research fellow in the same institute. From 1989 to 1991, he was an assistant researcher of Pacific International Center of High Technology Research (PICHTR) in Information Technology Division, Honolulu, Hawaii. At the same time, he was also an assistant researcher of Lab. of Intelligent and Parallel Systems (LIPS) at University of Hawaii. His research interests include applications of artificial intelligence and neural networks, theorems of neural networks, speech and image processing and recognition, parallel architectures in discrete transforms, computer architectures and multimedia systems. Currently, he is interested in the network management and mobile computing.