9

High-Level Petri Nets—Extensions, Analysis, and Applications

Xudong He* and Tadao Murata°
*School of Computer Science**
Florida International University, Florida, USA
Department of Computer Science°
University of Illinois at Chicago, Chicago, Illinois, USA

9.1 Introduction

Petri nets are an excellent formal model for studying concurrent and distributed systems and have been widely applied in many different areas of computer science and other disciplines (Murata, 1989). There have been over 8000 publications on Petri nets (refer to Website *http://www.daimi.au.dk/PetriNets/*). Since Carl Adam Petri originally developed Petri nets in 1962, Petri nets have evolved through four generations: the first-generation low-level Petri nets primarily used for modeling system control (Reisig, 1985a), the second-generation high-level Petri nets for describing both system data and control (Jensen and Bozenburg, 1991), the third-generation hierarchical Petri nets for abstracting system structures (He and Lee, 1991. He, 1996; Jensen, 1992), and the fourth-generation object-oriented Petri nets for supporting modern system development approaches (Agha, 2001). Petri nets have also been extended in many different ways to study specific system properties, such as performance, reliability, and schedulability. Well-known examples of extended Petri nets include timed Petri nets (Wang, 1998) and stochastic Petri nets (Marsan *et al.*, 1994; Haas, 2002). In this article, we present several extensions to Petri nets based on our own research work and provide analysis techniques for these extended Petri net models. We also discuss the intended applications of these extended Petri nets and their potential benefits.

Au: Please check to make sure this website still works-sites are easily outdated and replaced.

9.2 High-Level Petri Nets

In the past few years, a concerted effort by the worldwide Petri net community has resulted in an international standard on defining high-level Petri nets (HLPN) (ISOLTEC, 2002) that will profoundly help to facilitate and promote the research and applications of these nets. In the following sections, we follow as closely as possible the notations, concepts, and definitions given in the standard documentation to introduce high-level Petri nets, which are used later to define our extensions.

9.2.1 The Syntax and Static Semantics of High-Level Petri Nets

An **HLPN** is a structure:

$$N = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0).$$

Here:

- $NG = (P, T; F)$ is a net graph, with:
  - P a finite set of nodes, called places;
  - T a finite set of nodes, called transitions, disjoint from $P (P \cap T = \varnothing)$; and
  - $F \subseteq (P \times T) \cup (T \times P)$ a set of directed edges called arcs, known as the flow relation.
- $Sig = (S, O)$ is a Boolean signature, where S is a set of sorts and where O is a set of operators defined in the Annex A of ISO/JEC (2002).
- V is an S-indexed set of variables, disjoint from O.
- $H = (S_H, O_H)$ is a many-sorted algebra for the signature Sig, defined in this list:
- Type: $P \rightarrow S_H$ is a function that assigns types to places.
- $AN = (A, TC)$ is a pair of net annotations.
  - A: $F \rightarrow \text{TERM}(O \cup V)$ such that for all (p, t), $(t, p) \in F$ and all bindings α, $Val_\alpha(A(p, t))$, $Val_\alpha(A(t', p)) \in \mu\text{Type}(p)$. TERM $(O \cup V)$, α, Val_α and $\mu\text{Type}(p)$ are defined in Annex A of ISO/IEC (2002). A is a function that annotates each arc with a term that when evaluated (for some binding) results in a multiset over the associated place's type.
  - TC: $T \rightarrow TERM(O \cup V)_{\text{Bool}}$ is a function that annotates transitions with Boolean expressions.
- M_0: $P \rightarrow \cup_{p \in P} \mu$ Type(p) such that $\forall p \in P$, $M_0(p) \in \mu$ Type (p) is the initial marking function that associates a multiset of tokens (of the correct type) with each place.

Au: Do you think that equations in this chapter need a number assigned to them, as in (9.1), (9.2), (9.3), etc? If so, please add these throughout start from 9.1

The above definitions are directly from ISO/IEC 2002. Basically, HLPN has three essential parts: (1) a net graph NG defining the syntax, (2) an underlying algebraic specification (Sig, V, H) defining the semantic domain, and (3) a net inscription (Type, AN, M_0) mapping syntactic entities to their semantic denotations. By restricting the underlying algebraic specification and/or the net inscription, different variations of high-level Petri nets can be obtained.

9.2.2 Dynamic Semantics

Marking

The marking M of the HLPN is defined in the same way as the initial marking:

M: $P \rightarrow \cup_{p \in P} \mu$ Type(p) such that $\forall p \in P$, $M(p) \in \mu$ Type(p).

Enabling

- **Enabling of a Single Transition Mode**

A transition $t \in T$ is enabled in a marking M for a particular assignment of α_t to its variables and satisfies the transition condition, $\text{Val}_{\text{bool}}(TC(t)) = \text{true}$ known as a **mode** of t, if:

$$\forall p \in P Val_{\alpha_t}(\overline{p, t}) \leq M(p),$$
$$\text{where for} (u, v) \in (P \times T) \cup (T \times P),$$
$$\overline{u, v} = A(u, v) \text{ for } (u, v) \in F, \text{ or } u, v = \varnothing \text{ for } (u, v) \notin F.$$

- **Concurrent Enabling of Transition Modes**

Let α_t be an assignment for the variables of transition $t \in T$ that satisfies its transition condition, and then denote the set of all assignments for transition t, by Assign_t. Define the set of all transition modes to be $TM = \{(t, \alpha_t) | t \in T, \alpha_t \in \text{Assign}_t\}$ and a step to be a finite nonempty multiset over TM.

A step X of transition modes is enabled in a marking, M, if:

$$\forall p \in P \sum_{(t, \alpha_t) \in X} Val_{\alpha_t}(\overline{p, t}) \leq M(p).$$

Thus, all of the transition modes in X are concurrently enabled if X is enabled. Enabling of a single transition mode is a special case of concurrent enabling of transition modes.

The enabling condition of a transition specifies that enough right tokens are available to make the transition happen.

Transition Rule

If $t \in T$ is enabled in mode α_t for marking M, t may occur in mode α_t. When t occurs in mode α_t, the marking of the net is transformed to a new marking M', denoted in $M[t, \alpha_t > M'$, according to the following rule: Au: end bracker?

$$\forall p \in P (M'(p) = M(p) - Val_{\alpha_t}(\overline{p, t}) + Val_{\alpha_t}(\overline{t, p})).$$

If a step X is enabled in marking M, then the step can occur, resulting in a new marking of M' denoted by $M[X > M'$, where M' is given by:

$$\forall p \in P (M'(p) = M(p) - \sum_{(t, \alpha_t) \in X} Val_{\alpha_t}(\overline{p, t}) + \sum_{(t, \alpha_t) \in X} Val_{\alpha_t}(\overline{t, p}))$$

The firing rule defines the effect of firing a transition, which consumes specific tokens according to the mode in the preset (input places) of the transition and generates new tokens in the postset (output places) of the transition.

Behavior of A HLPN N

An **execution sequence** $M_0[X_0 > M_1[X_1 > \ldots$ of N is either finite when the last marking is terminal (no more enabled transition in the last marking) or infinite, in which each X_i is

a step. The **behavior** of N is the set of all execution sequences starting from the initial marking. The set of all reachable markings from the initial marking is denoted by $[M_0>$.

Although ISO/IEC (2002) has defined the concepts of markings, transition enabling, and firing rules, it stopped short in defining the dynamic semantics of an HLPN. In the existing Petri net literature, there have been several semantic models of Petri nets, such as interleaving, concurrent, and causal executions (Reisig, 1985b). From our own experience, we feel that the interleaving set semantics model defined above is adequate for studying many useful system behavioral properties.

9.3 Temporal Predicate Transition Nets

Petri nets are a model-oriented formal method and are well suited for modeling the dynamic behaviors of concurrent and distributed systems. Petri net specifications reveal system design structures and thus provide guidelines for system implementation. Furthermore, Petri net specifications are executable and support system simulation and testing in addition to formal analysis. It is not easy, however, to directly and explicitly express behavioral properties using Petri nets. On the other hand, property-oriented formal methods, such as temporal logic, are ideal for specifying and analyzing behavioral properties. It would be great to have a hybrid formal method by integrating the strengths of model-oriented and property-oriented formal methods. In recent years, integrating formal methods has become a major research area (Clark and Wing, 1996). There have been several published results on integrating Petri nets and temporal logic (Anttila *et al.*, 1983; Diaz *et al.*, 1983; He and Lee, 1990; He, 1992; Mandrioli *et al.*, 1996; Suzuki and Lu, 1989). Most of the earlier work was either lacking a systematic approach or using a low-level Petri net model. In the following subsections, we describe our results on integrating predicate transition nets [GL81] and first-order linear time temporal logic [MP92].

9.3.1 Definition of Temporal Logic

In this section, we define a linear time first-order temporal logic (LTFOTL) in the style of Lamport (1994a) based on a given PrTN. Let $N = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0)$ be a PrTN.

Values, State Variables, and States

The **set of values** is the multiset of tokens defined by the ground terms TERM(O) of N. Multisets can be viewed as partial functions. For example, multiset $\{3a, 2b\}$ can be represented as $\{a \mapsto 3, b \mapsto 2\}$.

The **set of state variables** is the set P of places of NG, which change their meanings during the executions of N. The arity of a place p is determined by its type Type (p).

The **set of states** St is the set of all reachable markings $[M_0>$ of N. A marking is a mapping from the set of state variables into the set of values. We use $M[\![x]\!]$ to denote the value of x under state (marking) M.

Since state variables take partial functions as values, they are *flexible* function symbols. We can access a particular component value of a state variable. There is a problem, however, associated with partial functions (i.e., many values are undefined). The above problem can easily be solved by extending state variables into total functions in the following way: for any n-ary state variable p, any tuple $c \in \text{TERM}(O)$, and any state M, if $p(c)$ is undefined under M, then let $M[\![p(c)]\!] = 0$. The above extension is consistent with the semantics of PrTN. Furthermore we can consider the meaning $[\![p(c)]\!]$ of the function application $p(c)$ as a mapping from states to **Nat** using a postfix notation for function application $M[\![p(c)]\!]$.

Rigid Variables, Rigid Function, and Predicate Symbols

Rigid variables are individual variables that do not change their meanings during the executions of N. All rigid variables occurring in our temporal logic formulas are bound (quantified), and they are the only variables that can be quantified. Rigid variables are variables appearing in the label expressions and constraints of N.

Rigid function and **predicate symbols** do not change their meanings during the executions of N. The set of rigid function and predicate symbols is defined in V of N.

State Functions, Predicates, and Transitions

A **state function** is an expression built from values, state variables, rigid function, and predicate symbols. For example, $[\![p(c) + 1]\!]$ is a state function, where c and 1 are values, p is a state variable, and $+$ is a rigid function symbol. Since the meanings of rigid symbols are not affected by any state, for any given state M, $M[\![p(c) + 1]\!] = M[\![p(c)]\!] + 1$.

A **predicate** is a Boolean-valued state function. A predicate p is satisfied by a state M if and only if $M[\![p]\!]$ is true.

A **transition** is a particular kind of predicate that contains primed state variables (e.g., $[\![p'(c) = p(c) + 1]\!]$). A transition relates two states (an old state and a new state), where the unprimed state variables refer to the old state and where the primed state variables refer to the new state. Therefore, the meaning of a transition is a relation between states. The term **transition** used here is a temporal logic entity. Although it reflects the nature of a transition in a PrTN net N, it is not a transition in N. For example, given a pair of states M and M': $M[\![p'(c) = p(c) + 1]\!]M'$ is defined by $M'[\![p(c)]\!] = M[\![p(c)]\!] + 1$. Given a transition t, a pair of states M and M' is called a "transition step" if $M[\![p]\!]M'$ equals true.

We can easily generalize any predicate p without primed state variables into a relation between states by replacing all unprimed state variables with their primed versions such that $M[\![p]\!]M'$ equals $M'[\![p]\!]$ for any states M and M'.

Temporal Formulas

Temporal formulas are built from elementary formulas (predicates and transitions) using logical connectives $\neg$ and $\wedge$ (and

derived logical connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$), universal quantifier $\forall$ (and derived existential quantifier $\exists$), and temporal operator **always** $\Box$-(and derived temporal operator sometimes $\diamond$).

The semantics of temporal logic is defined on infinite sequences of states that are extracted from the execution sequences of PrTNs, where the last marking of a finite execution sequence is repeated infinitely many times at the end of the execution sequence. For example, for an execution sequence $M_0[X_0 > M_1[X_1 > \ldots M_n$, the following infinite sequence behavior $\sigma = << M_0 \ldots M_n, M_n, \ldots >>$ is obtained. We denote the set of all possible behaviors obtained from a given PrTN as St^∞. Let u and v be two arbitrary temporal formulas; p be an m-ary place; t be a transition; x, $x_1 \ldots x_n$ be rigid variables; $\sigma = << M_0, M_1, \ldots >>$ be a behavior; and $\sigma^k = << M_k, M_{k+1}, \ldots >>$ be a k-step-shifted behavior sequence. We define the semantics of temporal formulas recursively as follows:

(1) $\sigma[\![p(x_1, \ldots, x_n)]\!] \equiv M_0[\![p(x_1, \ldots, x_n)]\!]$
(2) $\sigma[\![t]\!] \equiv M_0[\![t]\!]M_1$
(3) $\sigma[\![\neg u]\!] \equiv \neg\sigma[\![u]\!]$
(4) $\sigma[\![u \wedge v]\!] \equiv \sigma[\![u]\!] \wedge \sigma[\![v]\!]$
(5) $\sigma[\![\forall xu]\!] \equiv \forall x.\sigma[\![u]\!]$
(6) $\sigma[\![\Box u]\!] \equiv \forall n \in \mathbf{Nat}\ \sigma^n[\![u]\!]$

Intuitively temporal operator always $\Box$ means every state in a state sequence, and its dual operator sometimes $\diamond$ means some future state in a state sequence. The relationship between these two temporal operators is:

$$\neg\Box \equiv \neg \diamond .$$

A temporal formula u is said to be **satisfiable** denoted as $\sigma| = u$, if there is an execution σ such that $\sigma[\![u]\!]$ is true (i.e., $\sigma| = u \Leftrightarrow \exists\sigma \in ST^\infty.\sigma[\![u]\!]$). A temporal formula u is **valid** with regard to N, denoted as $N| = u$, if it is satisfied by all possible behaviors St^∞ from N: $N| = u \Leftrightarrow \forall\,\sigma \in St^\infty.\sigma[\![u]\!]$.

9.3.2 Temporal Predicate Transition Net

A temporal predicate transition net (TPrTN) is a tuple $TN = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0, f)$, where $N = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0)$ is a PrTN, and f is a LTFOTL formula that constrains the execution of TN. The semantics of TN are defined to be the set of execution sequences $\Sigma = \{\sigma|\sigma \in St^\infty \wedge \sigma[\![f]\!]\}$.

It is easy to see that a TPrTN TN is a PrTN N with a temporal logic formula f. The temporal logic formula is defined according to the net graph NG using logical connectives and operators, which can be viewed as a part of the underlying algebraic specification (Sig, V, H). The temporal logic formula f is evaluated using the dynamic behaviors of N. By incorporating a temporal formula into the definition of a PrTN, we are able to explicitly specify and verify many system properties such as fairness.

9.3.3 An Example of TPrTN

The **Five Dining Philosophers** problem is a classical example for studying behavioral properties of concurrent processes. There are five philosophers sitting around a round table, and there is one chopstick between two adjacent philosophers. Each philosopher is either thinking or eating. In order for a philosopher to eat, he needs to pick up two chopsticks next to him. There are several interesting issues with regard to a system model of this simple problem:

- Is the system deadlock free? A deadlock occurs when the system cannot progress anymore due to a formation of a waiting cycle.
- Has the system enforced mutual exclusion? Mutual exclusion ensures that no two neighboring philosophers can eat at the same time.
- Is the system live-lock free? A live lock occurs when some philosopher has no chance to eat anymore from a certain point and thus starves to death.

The following TPrTN models the Five Dining Philosophers problem, shown in Figure 9.1, in which five philosophers are denoted by five integer tokens 0 to 4 and five chopsticks are also denoted by five integer tokens 0 to 4. Places p_1 and p_3 stand for the **Thinking** and **Eating** states of philosophers, respectively. Place p_2 defines the availability of chopsticks. Transitions t_1 and t_2 stand for the actions of **Pick up** and **Put down** chopsticks, respectively.

The algebraic definition of the net $TN = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0, f)$ is as follows:

$P = \{p_1, p_2, p_3\}$, $T = \{t_1, t_2\}$,
$F = \{(p_1, t_1), (p_2, t_1), (t_1, p_3), (p_3, t_2), (t_2, p_1), (t_2, p_2)\}$,
$\text{Type}(p_1) = \text{Type}(p_2) = \text{Type}(p_3) = \text{INT}$,
$A(p_1, t_1) = x$, $A(p_2, t_1) = \{x, y\}$, $A(t_1, p_3) = x$,
$A(p_3, t_2) = x$, $A(t_2, p_1) = x$, $A(t_2, p_2) = \{x, y\}$,
$TC(t_1) = TC(t_2) = y = x \oplus 1$,
$M_0(p_1) = M_0(p_2) = \{0, 1, 2, 3, 4\}$, and $M_0(p_1) = \{\ \}$,

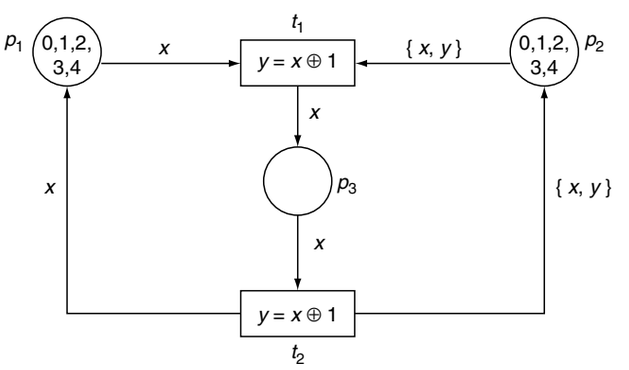

FIGURE 9.1 A TPrTN Specification of the Five Dining Philosophers Problem.

where INT is a sort defined in Sig, x and y are variables in V, $\oplus$ is a modulo 5 addition operation given in Sig and defined in the algebra H, and $y = x \oplus 1$ is a term in algebra H.

Temporal formula f is defined as follows:

$$f = \forall x(\Box \diamond (p_1(x) = 1 \wedge p_2(x) = 1 \wedge p_2(x \oplus 1) = 1) \Rightarrow \Box \diamond (p_3'(x) = 1 \wedge p_1'(x) = 0 \wedge p_2'(x \oplus 1) = 0)).$$

Temporal formula f defines the strong fairness (Manna and Pnveli, 1992) that captures the enabling condition and the firing result of transition t_1 with every mode (defined by $\forall x$). Intuitively, it states that any philosopher who wants to eat **infinitely many times** (defined by the temporal operator sequence $\Box\diamond$) will eat **infinitely many times**.

9.3.4 Analysis of TPrTNs

System behavioral properties can be divided into two major categories: **safety** properties and **liveness** properties (Manna and Pnveli, 1992). Widely accepted formal definitions of safety and liveness properties were given in [AS85 (Alpern and Schneider, 1985)]. A safety property, which is different from the concept **safeness** in Petri net literature (Murata, 1999) needs to hold in every state and in every state sequence. The canonical form of a safety property is characterized by a temporal formula $\Box w$. A liveness property, which is different from the concept transition **liveness** in Petri net literature (Murata, 1989), needs to hold in some future state in every state sequence. The canonical form of a liveness property is characterized by a temporal formula $\diamond w$.

In He and Lee (1990) and He and Ding (1992), we developed an axiomatic approach of using temporal logic to analyze PrTNs. The essential ideas are to derive system-dependent temporal inference rules from PrTN transitions. These inference rules capture the causal relationships of transition firings. Thus, executions of a PrTN become temporal logic deductions in the derived axiomatic system. We have proved that safety properties (Manna and Prueli, 1992) can be effectively analyzed using this axiomatic approach. Furthermore, we have shown how to analyze liveness properties (Manna and Prueli, 1992) using the above temporal formula by the (1991).

Given $TN = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0, f)$, we can obtain the following system-dependent temporal axiom system L:

(1) A system-dependent axiom captures the initial marking M_0: $\wedge_{p \in P}(IC_p)$, (IC)

where IC_p represents the initial condition of place p under M_0

(2) or each place p of arity k, let $y_1, \ldots, y_k$ be new variables not used in any label and t be any transition in the postset $p\bullet$ with label $L(p, t) = \{m_1(x_{11}, \ldots, x_{1k}), \ldots, m_n(x_{n1}, \ldots, x_{nk})\}$. We construct an inference rule as follows: $p(y_1, \ldots, y_k) > p'(y_1, \ldots, y_k) \mid - \vee_{t \in p\bullet}((ET_t) \wedge (\vee_{l \le i \le n} (\wedge_{l \le j \le k} (y_j = x_{ij}))))$, (PR_p)

where ET_t represents the causal relationship of firing transition t. Intuitively, each inference rule states if a place loses tokens from one marking to another, then some relevant transition must have fired (He and Ding, 1992) The variables are universally quantified and are omitted for simplicity. The above inference rules only reflect the local state changes. Together with system independent axioms and inference rules (Manna and Prueli, 1992, 1995), we can use IC and PR_p to prove a variety of safety properties.

The following is the derived system-dependent temporal axiom system of the Five Dining Philosophers problem shown in Figure 9.1:

$$p_1(0) = 1 \wedge p_1(1) = 1 \wedge p_1(2) = 1 \wedge p_1(3) = 1 \wedge p_1(4) = 1 \wedge p_2(0) = 1 \wedge p_2(1) = 1 \wedge p_2(2) = 1 \wedge p_2(3) = 1 \wedge p_2(4) = 1. \quad (IP)$$

$$p_1(x) > p_1'(x) \mid - p_1(x) \ge 1 \wedge p_2(x) \ge 1 \wedge p_2(y) \ge 1 \wedge y = x \oplus 1 \wedge p_1'(x) = p_1(x) - 1 \wedge p_2'(x) = p_2(x) - 1 \wedge p_2'(y) = p_2(y) - 1 \wedge p_3'(x) = p_3(x) + 1. \quad (PR_{p_1})$$

$$p_2(x) > p_2'(x) \mid - p_1(x) \ge 1 \wedge p_2(x) \ge 1 \wedge p_2(y) \ge 1 \wedge y = x \oplus 1 \wedge p_1'(x) = p_1(x) - 1 \wedge p_2'(x) = p_2(x) - 1 \wedge p_2'(y) = p_2(y) - 1_3'(x) = p_3(x) + 1. \quad (PR_{p_2})$$

$$p_3(x) > p_3'(x) \mid - p_3(x) \ge 1 \wedge p_1'(x) = p_1(x) + 1 \wedge p_2'(x) = p_2(x) + 1 \wedge p_2'(y) = p_2(y) + 1 \wedge p_3'(x) = p_3(x) - 1 \wedge y = x \oplus 1. \quad (PR_{p_3})$$

A TPrTN is **deadlock free** if and only if there is at least one transition being enabled in any reachable state or if a normal terminal state has been reached. Deadlock freedom is a safety property. The deadlock freedom property of the Five Dinning Philosophers problem is formulated as follows:

$$\Box \exists x, y(p_3(x) \ge 1 \vee (p_1(x) \ge 1 \wedge p_2(x) \ge 1 \wedge p_2(y) \ge 1 \wedge y = x \oplus 1)). \quad (*)$$

A TPrTN net is **livelock free** if and only if for any transition enabled infinitely often with a mode α, the same transition with mode α will fire infinitely often. Livelock freedom is a liveness property. One of the liveness properties of the Five Dining Philosophers problem is that every individual philosopher will eventually get a piece of meal (starvation free), which can be expressed by the following temporal formula:

$$\forall x \Box \diamondsuit (p_3(x) = 1). \quad (**)$$

The proofs of the above formulas involve logical deductions using inference rules of ordinary first-order logic and temporal logic as well as system-dependent inference rules listed previously. Because logical deductions are machine-oriented and not suitable for human comprehension, we omit the proofs of the

above properties (*) and (**). Interested readers can find the proofs of (*) and (**) in He (1991) and He and Ding (1992).

9.4 *PZ* Nets

A widely accepted formal notation for specifying the functionality of sequential systems is *Z* (Spivey, 1992). The *Z* is based on typed set theory and first-order logic and, thus, offers rich type definition facility and supports formal reasoning. However, *Z* does not support an effective definition of concurrent and distributed systems, and *Z* specifications do not have explicit operational semantics. Many researchers have attempted to combine *Z* with other concurrent models, including CSP (Benjamin, 1990), CCS (Taguchi and Araki, 1997), and temporal logic (Duke and Smith, 1989; Clarke and Wing, 1996; Evans, 1997; Lamport, 1994b) in recent years. Several works attempted to integrate Petri nets and *Z* (Van Hee *et al.*, 1991; He, 2001). In van Hee *et al.* (1991); *Z* was used to specify (1) the metamodel of restricted hierarchical colored Petri nets (that allows super transitions but not super places) and (2) the transitions of specific colored Petri nets. A complete specification consists of a hierarchical net structure, one global state schema for defining all places, and one operation schema for each transition. Schemas of transitions are piped to obtain an operational semantics through the use of input and output variables. In He (2001), a formal method of integrating Petri nets with *Z* was presented with the objectives to extend Petri nets with data and function definition capabilities through an underlying *Z* specification and to extend *Z* with an explicit operational semantics and concurrency mechanisms through Petri nets. In the following subsections, we briefly describe the basics of *Z* notations and introduce the results given in He (2001).

9.4.1 A Brief Overview of *Z*

Besides elementary data types, *Z* allows the user to introduce other primitive types, called given types. Given types are in capital letters and enclosed by a pair of brackets that do not require further definitions. For example, [IDEN] introduces a given type IDEN.

Part of *Z* is an essential notation called *schema* from which new types and their properties can be defined. A *Z* schema has a boxed structure often consisting of two major parts: the **declaration part** and the **predicate part** (optional) as follows:

Name
Declaration part
Predicate part

The name of a schema is global and can be used in the declaration part of other schemas to reuse the variable definitions.

The declaration part defines local variables of the schema in the form: **var: Type** (the collection of these variable definitions forms the **signature** of the schema), and thus a reference of this variable in another schema needs to be prefixed by its schema name followed by a period (i.e., **Name. var**: A variable name can be a simple name (defining the old value), a name with a dash′ (defining the new value of the same name without the dash), a name with a question mark? (an input from the external environment), or a name with an exclamation point! (an output to the external environment). If a schema includes both the name of a schema *S* and its dashed version *S*′, the following notation is used: ΔS when the value of some variable defined in *S* has been changed by the required processing or ΞS when nothing in *S* is changed by the required processing.

Au: This is not a dash. Do you mean to say a "prime symbol", as the description, or do you want to change this to a dash "–"?

The predicate part specifies the constraint (invariant) of a definition or the precondition (subformulas without dashed variables) and postcondition (subformulas containing dashed variables) of some processing in terms of first-order logic formulas. Subformulas on separate lines in the predicate part are conjoined by default. The predicate thus defines the **property** of the schema.

State Schemas and Operation Schemas

A schema can be used to define the abstract state of a system, called a **state schema** in the sequel, when the state machine model is used. The predicate part in this case defines the data invariant among involved variables.

Each state schema *S* needs an initial value that is defined by an initialization schema that enumerates the initial values of state variables in the predicate part.

A schema *SC* can be used to define an operation, called an **operation: schema** in the sequel, which includes both a state schema name *S* and its version *S*′ in its declaration part. However, ΔS or ΞS is often used instead of *S* and *S*′. The predicate part can be divided into the precondition part (denoted by pre-*SC* in the sequel) and the postcondition part (denoted by post-*SC* in the sequel).

Operations on Schemas

New schemas can be built using existing schemas besides the inclusion mentioned in the previous paragraphs.

(1) **Schema disjunction**: Schema disjunction has the form New $\hat{=}$ Old1 $\vee$ Old2, where New, Old1, and Old2 are schema names. The declaration part of new schema New is obtained by merging the declaration parts of Old1 and Old2, and the predicate part of New is obtained by making a disjunction of the predicate part of Old1 with that of Old2.

(2) **Schema conjunction**: Schema conjunction has the form New $\hat{=}$ Old1 $\wedge$ Old2, where New, Old1, and Old2 are schema names. The declaration part of new schema New is obtained by merging the declaration parts of Old1 and Old2, and the predicate part of New is obtained by making a conjunction of the predicate part of Old1 with that of Old2.

Au: Besides? Not clear here – Schemes can be built except those restricted by the inclusion?

(3) **Schema composition**: Let Old1 and Old2 be two schemas; a new schema of the form New $\hat{=}$ Old1; Old2 can be defined and has the following meaning:
- The signature of New is the inputs and outputs of both Old1 and Old2, together with the nondashed variables in Old1 and dashed variables of Old2;
- The property of Old1 is included in New, but all the dashed names are redecorated with a decorator not used in either Old1 and Old2;
- The property of Old2 is included in New, but all the nondashed names are decorated with the same decorator as was used in Old1; and
- The newly decorated names are hidden with an existential quantifier.

Other operations related to schemas can be found in work by Spivey (1992), including using a schema as a type in the declaration part of other schemas, hiding declarations in a schema, and reusing the operations defined in one schema in other related schemas through promotion.

9.4.2 Definition of *PZ* Nets

Let $Z = (Z_P, Z_T, Z_I)$ be a collection of Z schemas. If we let z be a Z schema, we use **name**(z), **sig**(z), and **prop**(z) to denote the name, the signature part (as a typed set of mappings), and property part of z, respectively in the sequel. The Z_I schemas in Z satisfy the following conditions:

- $\forall z1, z2 \in Z_P \cdot (z1 \neq z2 \Rightarrow \mathbf{sig}(z1) \cap \mathbf{sig}(z2 = \varnothing)$;
- $\forall z1, z2 \in Z_I \cdot (z1 \neq z2 \Rightarrow \mathbf{sig}(z1) \cap \mathbf{sig}(z2) = \varnothing)$; and
- $|Z_P| = |Z_I|$.

The first two conditions state that the signatures of z schemas are pair-wise disjoint, and the last condition states that the number of z schemas in Z_P is the same as that in Z (i.e., one-to-one correspondence).

A *PZ* net is a tuple $ZN = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0, Z)$.

Here, Z is a collection of Z schemas satisfying the above conditions, $AN = (\text{Type}, A, TC)$, and:

- *Type*: $P \rightarrow Z_P$ is one-to-one mapping, giving the data definition of *ZN*. For each place $p \in P$, Type maps p to a unique Z schema $z \in Z_P$ such that $p =$**name**(z). The type of p is defined by the signature of z.
- $TC: T \rightarrow Z_T$ is one-to-one mapping, providing the functionality definition of *ZN*. For each transition $t \in T$, C maps t to a Z schema $z \in Z_T$ such that $t=$**name**(z). The functional processing of t is defined by the property of z. Furthermore, the following constraint is satisfied. For any $p \in P$, $t \in T$:
  (1) If $(p, t) \in F$, then **sig** (Type (p)) $\cap$**sig**(TC(t)) $\neq \varnothing$.
  (2) If $(t, p) \in F$, then **sig** (Type (p)) $\cap$ **sig**($TC(t)$) $\neq \varnothing$.
- $A: F \rightarrow pVar$ is the control flow definition of *ZN*, where *Var* is the set of all hidden variables (through quantification) in Z_T. Let $Var(t)$ denote the set of hidden variables in $TC(t)$ for any transition t, let $S(v)$ denote the sort (or type) of a variable v, and let $S(V)$ denote the set of the sorts of the variables in the sorted set V. Then, the following constraints are satisfied for any $p \in P$:
  (1) $\bar{A}(p, t) \subseteq Var(t)$ and $\bar{A}(t, p) \subseteq Var(t)$, where
  $$\bar{A}(x, y) = \begin{cases} A(x, y) & \text{if (x, y)} \in \text{F} \\ \varnothing & \text{otherwise} \end{cases}$$
  (2) For any $x \in \bar{A}(p, t)(\text{or}\bar{A}(t, p)) \cdot S(x) \in S(\mathbf{sig}(\text{Type}(p)))$.
  (3) $\mathbf{prop}(TC(t)) \Rightarrow \mathbf{sig}(\text{Type}(p)') = \mathbf{sig}(\text{Type}(p)) - \bar{A}(p, t) \cup \bar{A}(t, p)$.
- $M_0: P \rightarrow Z_I$ is a Type-respecting (i.e., $\mathbf{sig}(\text{Type}(p)) = \mathbf{sig}(M_0(p))$ initial marking of *ZN*, where Z_1 is a set of Z schemas defining the initial state of the system. $M_0(p)$ is the Z schema in Z_1, defining the initial marking of place p.

The following is a simple example of a *PZ* net specification of the familiar Five Dining Philosophers problem. The overall system structure is shown in Figure 9.2, which is the same as Figure 9.1 except for the renaming.

The Z schemas (Z_P, Z_T, Z_I) are the following:

(1) Z_P = {thinking, chopstick, eating}:

Thinking
  tphil: p PHIL

Chopstick
  chop: p CHOP

Eating
  ephil: p (PHIL × CHOP × CHOP)
  left: PHIL → CHOP
  right: PHIL → CHOP

(2) Z_T = pickup, putdown:

Pick up
  Δ Thinking
  Δ Chopstick

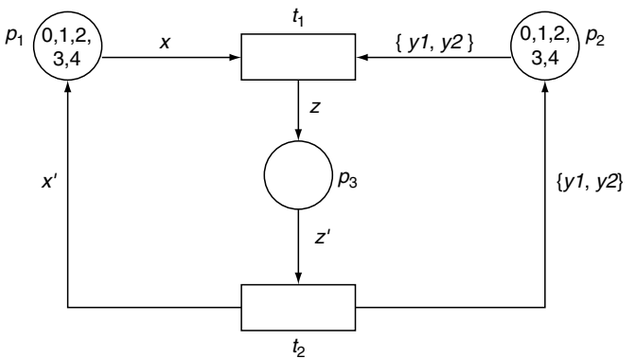

FIGURE 9.2 A *PZ* Net Specification of the Five Dining Philosophers Problem.

Δ Eating

$\exists x$: PHIL • $\exists y_1, y_2$: CHOP • $\exists z'$: PHIL × CHOP × CHOP•

$(x \in \text{tphil} \wedge y_1 \in \text{chop} \wedge y_2 \in \text{chop} \wedge y_1 = \text{left}(x) \wedge$

$y_2 = \text{right}(x) \wedge$

$(z' = < x, y_1, y_2 >$

$\text{tphil}' = \text{tphil} \backslash \{x\}$

$\text{chop}' = \text{chop} \backslash \{y_1, y_2\}$

$\text{ephil}^{\bullet} = \text{ephil} \cup \{z'\}$

$\text{left}' = \text{left} \wedge \text{right} = \text{right}))$

Put down

Δ Thinking

Δ Chopstick

Δ Eating

$\exists x'$: PHIL • $\exists y_1', y_2'$: CHOP • $\exists z$: PHIL × CHOP × CHOP•

$(z \in \text{ephil} \wedge z = < x', y_1', y_2' >$

$\text{tphil}' = \text{tphil} \cup \{x'\}$

$\text{chop}' = \text{chop} \cup \{y_1', y_2'\}$

$\text{ephil}' = \text{ephil} \backslash \{z\}$

$\text{left}' = \text{left} \wedge \text{right} = \text{right})$

(3) Z_1 = Init_Thinking, Init_Chopstick, Init_Eating:

Init_Thinking

$\text{tphil} = \{0, 1, 2, 3, 4\}$

Init_Chopstick

$\text{chop} = \{0, 1, 2, 3, 4\}$

Init_Eating

Eating

$\text{ephil} = \varnothing$

$\text{left} = \{0 \mapsto 4, 1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 2, 4 \mapsto 3\}$

$\text{right} = \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4\}$

$\text{Type} = \{p_1 \mapsto \text{Thinking}, p_2 \mapsto \text{Chopstick}, p_3 \mapsto \text{Eating}\}$

$TC = \{t1 \mapsto \text{Pickup}, t2 \mapsto \text{Putdown}\}$,

$A = \{(p1, t1) \mapsto \{x\}, (p2, t1) \mapsto \{y1, y2\}, (t1, p3) \mapsto \{z'\},$

$(p3, t2) \mapsto \{z\}, (t2, p1) \mapsto \{x'\}, (t2, p2) \mapsto \{y1', y2'\}\}$,

$M_0 = \{p1 \mapsto \text{Init\_Thinking}, p2 \mapsto \text{Init\_Chopstick},$

$p3 \mapsto \text{Init\_Eating}\}$.

9.4.3 PZ Net Analysis

In He (1995, 2001), a structural induction technique was proposed for analyzing safety properties of *PZ* nets, which is based on an invariance inference rule from work by Manna and Prueli (1995). This technique is briefly discussed in the following subsections.

Formalizing Invariant Properties

Let $[M_0 >^{\omega}$ denote the set of all valid marking sequences extracted from all execution sequences of a given *PZ* net, and let σ be a valid marking sequence with $|\sigma|$ as the length of the sequence and $\sigma(i)$ as the ith state (marking) in σ. W (a first order logic formula) is an invariant property if and only if the following holds: $\forall \sigma: \sigma \in [M_0 >^{\omega} .(\forall i: 0 \leq i \leq |\sigma|.(\sigma(i) \mid= W))$, where $\sigma(i) \mid= W$ denotes that marking $\sigma(i)$ satisfies W (i.e., the evaluation of W under marking $\sigma(i)$ yields true). Thus, a safety property holds in every state (marking) of every valid marking sequence. The above formulation can be simplified to the following equivalent version in terms of the set of all reachable markings only:

Au: end bracker?

$$\forall M: M \in [M_0 > .(M \mid= W).$$

Proving Invariant Properties

In Manna and Prveli (1995), several temporal logic-based inference rules for invariance (or safety properties) were given. Among the rules, the following basic invariance rule (in its state validity form) is reformulated in terms of *PZ* nets and is essential.

The Basic Invariance Rule

B1. $M_0 \Rightarrow W$

B2. $$\frac{C(t): \alpha \wedge W \Rightarrow W' \text{ for every } t \in T \text{ and occurrence } \alpha}{M \mid= W \text{ for every } M \in [M_0 >}$$

In the basic invariance rule, premise B1 requires that the initial marking M_0 imply property W, and premise B2 requires that all transitions preserve W. $C(t)$ is the Z schema associated with t that defines the enabling condition (precondition) and the firing result (postcondition) of t. W' is obtained from W by changing the names of variables to their dashed version. Based on premises B1 and B2, we conclude that W is valid or is satisfied under any reachable marking from M_0.

We provide the following procedure to use the above inference rule:

Step 1. Prove the initial marking as satisfying a system property formula.

Step 2. Assume the system property formula holds after k events in a state M.

Step 3. Prove the system property formula holds after $k + 1$ events in any directly reachable state M' from M.

It is easy to see that steps 2 and 3 in the temporal induction proof technique fulfill premise B2 of the invariance rule. Furthermore, we only need to consider the firing of a relevant transition and the associated Z schema with regard to the given property under the guidance of the net structure during step 3.

For example, a system deadlock may occur when a particular place has a special marking. To show the system does not have the deadlock, we only need to show that transitions connected to this place cannot result in this special marking. Therefore, the proof is in general local in the sense that only a subset of transitions needs to be considered. Similar ideas have been also explored in other temporal analysis techniques (He and Lee, 1990; He and Ding, 1988). In general, logical, net structural, and net behavioral reasonings are needed to prove an invariant property. The above temporal induction technique is demonstrated in the following example.

The deadlock freedom property of the Five Dining Philosophers problem is formulated as follows:

$$\Box(\exists x \in \text{ephil}(p_3(x) \geq 1 \vee \exists x \in \text{tphil}.(\text{right}(x) \in p_2 \wedge \text{left}(x) \in p_2)).$$

This problem explains that at any state, a philosopher is eating, which ensures the enabledness of transition Put down, or a philosopher is thinking, which ensures the enabledness of transition (Pick up)

Au: Not a word in the dictionary-please change word.

The above formula can be rewritten as follows without using the temporal operator:

$$\forall M \in [M_0 >.(\exists x \in \text{ephil}.(p_3(x) \geq 1 \vee \exists x \in \text{tphil}\ (\text{right}(x) \in p_2 \wedge \text{left}(x) \in p_2)).$$

Thus, we need to prove the following formula using the structural induction technique:

$$\exists x \in \text{ephil}(p_3(x) \geq 1 \vee \exists x \in \text{tphil right}(x) \in p_2 \wedge \text{left}(x) \in p_2) \quad (*)$$

Here is the proof outline of the formula (*):

Step 1 Under the initial marking M_0, Init_Thinking, Init_Chopstick, and Init_Eating endures:

$$\exists x \in \text{tphil}.(\text{right}(x) \in p_2 \wedge \text{left}(x) \in p_2) \text{ and thus}(*).$$

Step 2: Assume (*) holds after k transitions in a state M.

Step 3: Prove (*) holds after $k+1$ transitions in a state M' such that $M[t/\alpha > M'$.

Au: end bracker?

Case 1: Firing transition Pick up with $\alpha = \{x/ph_1, y_1/ch_1, y_2/ch_2\}$ and from the postcondition of schema Pick up with $ch_1 \in$ ephil′in in M' Thus, $\exists x \in \text{ephil}(p_3(x) \geq 1)$ is true in M', and hence (*) holds in M'.

Case 2: Firing transition putdown $\alpha = \{z/< ph_1, ch_1, ch_2 >\}$:

From the precondition of put down with left $(ph_1) = ch_1$ and right $(ph_1) = ch_2$; from the postcondition of schema put down with $ph_1 \in$ tphil, $ch_1 \in$ chop′, and $ch_2 \in$ chop'. Thus, $\exists x \in$ tphil. right$(x) \in p_2 \wedge$ left$(x) \in p_2)$ is true in M', and hence (*) holds in M'.

Therefore, (*) has been proven.

9.5 Hierarchical Predicate Transition Nets

The development of hierarchical predicate transition nets (HPrTNs) was motivated by the need to construct specifications for large systems using Petri nets (Reisig, 1987) and inspired by the development of modern high-level programming languages and other hierarchical and graphical notations, such as data flow diagrams (Yourdon, 1989) and statecharts (Harel, 1988). Similar work on introducing hierarchies into colored Petri nets was given in Jensen (1992, 1995). With the introduction of hierarchical structures into predicate transition nets, the resulting net specifications are more understandable, and the specification construction process becomes more manageable. HPrTNs were used in specifying several systems, including an elevator system (He and Lee 1991), a library system (He and Yana, 1992), and a hurried dining philosophers system (He and Ding, 2001). HPrTNs can be analyzed directly by using a structural induction technique combining structural, behavioral, and logical reasoning (He, 2001) and can be translated into program skeletons in a concurrent and parallel object-oriented programming language CC++ (He 2000c) and Java (Lewandowski and He, 1998, 2000). A complete formal definition of HPrTNs was given in He (1996). In the following subsections, basic concepts and notation of HPrTNs are briefly introduced.

9.5.1 Definition of HPrTNs

An HPrTN is a structure:

$$HN = (NG, \text{Sig}, V, H, \text{Type}, AN, M_0, \rho).$$

Here:

- $NG = (P, T, F)$ is a net graph. P and T are finite sets of places and transitions such that $P \cap T =$. Elements in P are represented by solid and dotted circles. Similarly, elements in T are represented by solid and dotted boxes. Solid circles or boxes are elementary nodes, and dotted circles and boxes are super nodes. In particular, we identify two subsets $IN \subseteq P \cup T$ and OUT $\subseteq P \cup T$ such that IN contains the heads of all incoming **nonterminating arcs** (an arc inside a super node is a nonterminating arc if one of its ends is connected to the boundary of the super node) and OUT contains the tails of all outgoing nonterminating arcs. Nodes in IN $\cup$ OUT are called **interface** nodes. We use $\bullet$ IN to denote the set of the presets of all elements in IN (i.e., $\bullet\text{IN} = \{\bullet n | n \in \text{IN}\}$), and OUT $\bullet$ to denote the set of the postsets of all elements in OUT. F is the set of arcs and is called the **flow relation**, satisfying the conditions: $P \cap F =$, $F \cap T =$, and $F \subseteq (\bullet\text{IN} \times \text{IN} \cup P \times T \cup T \times P \cup \text{OUT} \times \text{OUT} \bullet)$. An arc f can be uniquely identified by a pair of nodes (n_1, n_2) denoting its source and sink, in which $n_1(n_2)$ may denote the preset (postset) of $n_2(n_1)$ when f is a nonterminating arc.

- Sig, *V*, *H*, Type are defined as in HLPN in Section 9.2.
- $AN = (A, TC)$ is a pair of net annotations.

A is a function that annotates each are with a term that when evaluated (for some binding) results in a multiset over the associated place's type. Furthermore, all simple labels of a compound label must have distinct identifiers, and all simple labels of arcs connected to the same node must have distinct identifiers. Because compound labels define data flows as well as control flows, the following basic control flow patterns (He and Lee, 1991) must be correctly labeled: (1) data flowing into and out of an elementary transition must take place concurrently, and (2) data flowing into and out of an elementary predicate can occur at different times. Furthermore, data flows between different levels of hierarchies must be balanced (i.e., a simple label occurs in a nonterminating arc if and only if it also appears in an arc with the same direction connected to the enclosing super node).

- *TC* is a function that annotates transitions with Boolean expressions. A super transition is an abstraction of low-level actions, and its meaning is thus completely defined by the low-level refinement. Therefore, the constraint of a super transition is true by default (it is conceivable that a nontrivial constraint for a super transition might be useful; however, in general it is very difficult to define such a constraint and also very difficult to interpret the constraint with regard to the operational (dynamic) semantics of the super transition).
- M_0 is the initial marking function.
- $\rho\dot{P} \cup T \rightarrow p(P \cup T)$ is a hierarchical mapping that defines the hierarchical relationships among the nodes in *P* and *T*; this mapping also satisfies the constraint that the interface nodes $\in$ IN $\cup$ OUT be all predicates if their parent node is a predicate or all transitions if their parent node is a transition. For any node *n*, $\rho(n)$ defines the immediate descendant nodes of *n*. The ancestor and descendants of any node can be easily expressed by using well-known relations, such as transitive closure on ρ A node in an HPrTN is local to its parent and can be uniquely identified by prefixing its ancesters' names separated with periods to its own name; however, often its own name is referred whenever a no name clash occurs.

The enabling condition of an elementary transition is defined exactly the same as that of an *HLPN*'s in Section 9.2. A super transition is enabled if at least one of its interface child transitions in IN is enabled and its firing is defined by an execution sequence of its child transitions; thus, its behavior is fully defined by its child transitions. The firing rule of a transition is formally defined in He (1996). Two transitions (including the same transition with two different occurrence modes) can fire concurrently if they are not in conflict (the firing of one of them disables the other). Conflicts are resolved nondeterministically. The firing of an elementary transition is atomic, and the firing of a super transition implies the firing of some elementary transition and may not be atomic. We define the behavior of an HPrTN to be the set of all possible maximal execution sequences containing only elementary transitions. Each execution sequence represents consecutively reachable markings from the initial marking in which a successor marking is obtained through a step (firing of some enabled transitions) from the predecessor marking.

Au: Is this part of 9.3 Figure? If not, please provide an introductory sentence

Figure 9.3 shows an HPrTN specification of the Five Dining Philosophers problem.

Type(Thinking) $= \varphi$ Eating $= p$ (PHIL), Type(Avail) $=$ Type (Used) $= p$ (CHOP),

Type(Chop)) $= \varphi$(Avail) $\cup \varphi$(Used), Type(Relation) $= p$ (PHIL $\times$ CHOP $\times$ CHOP),

$A(f_3) = < 1, re >$, $A(f_4) = < 2, re >$, $A(f_5) = < 3, ph >$,
$A(f_6) = < 4, ph >$, $A(f_7) = < 5, ph >$, $A(f_8) = < 6, ph >$,
$A(f_{13}) = < 7, \{ch_1, ch_2\} >$, $A(f_{14}) = < 8, \{ch_1, ch_2\} >$,
$A(f_{15}) = < 9, \{ch_1, ch_2\} >$, $A(f_{16}) = < 10, \{ch_1, ch_2\} >$,
$A(f_9) = A(f_3) \times A(f_{13})$, $A(f_{10}) = A(f_4) \times A(f_{16})$,
$A(f_{11}) = A(f_3) \times A(f_{15})$, $A(f_{12}) = A(f_4) \times A(f_{14})$,
$A(f_1) = A(f_{13}) \times A(f_{15})$, $A(f_2) = A(f_{14}) \times A(f_{16})$,
$TC(\text{Pickup}) = (ph = re[1]) \wedge (ch1 = re[2]) \wedge (ch2 = re[3])$,
$TC(\text{Putdown}) = (ph = re[1]) \wedge (ch1 = re[2]) \wedge (ch2 = re[3])$,
$TC(\text{Phil}) = \text{True}$,
$M_0(\text{Thinking}) = \{1, 2, \ldots, k\}$, $M_0(\text{Eating}) = \{\}$,
$M_0(\text{Avail}) = \{1, 2, \ldots, k\}$, $M_0(\text{Used}) = \{\}$,
$M_0(\text{Chop}) = M_0(\text{Avail}) \cup M_0(\text{Used}) = \{1, 2, \ldots, k\}$,
$M_0(\text{Relation}) = \{(1, 1, 2), (2, 2, 3), \ldots, (k, k, 1)\}$

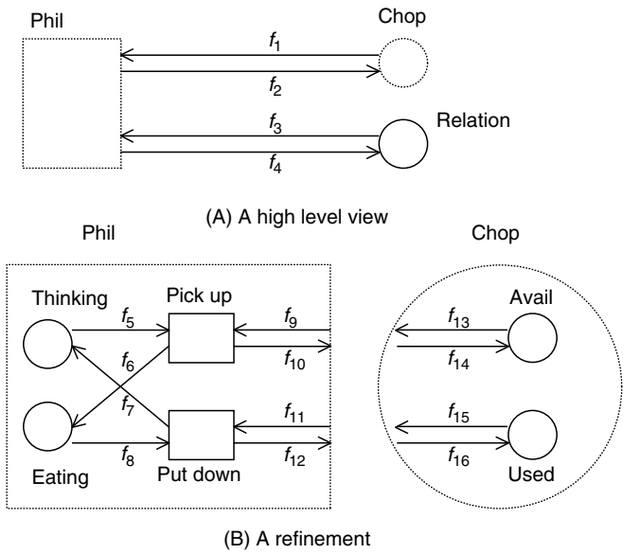

FIGURE 9.3 an HPrTN specification of the Five Dining Philosophers Problem.

9.5.2 System Modeling Using HPrTNs

HPrTNs can be used to model systems using the traditional structured approach (Yourdon, 1989) or modern object-oriented approach (Booch, 1994). He and Ding (2001) proposed an approach to realize various object-oriented (OO) concepts in HPrTNs. Furthermore, He (2000a, 2000b) and Dong and He (2001) applied HPrTNs to formalize several diagrams in the Unified Modeling Language (UML)—a software industry standard second-generation object-oriented modeling language. In the following sections, we briefly describe how to model several key OO concepts in HPrTNs.

Classes

One of the central ideas of OO paradigm is data encapsulation captured by the **class** concept. A class is essentially an abstract data type with a name, a set of related data fields (or attributes), and a set of operations on the data fields. It is straightforward to use a predicate to denote a data field (structure) and a transition to represent an operation in Petri nets. The current value of a data field is determined by the tokens of the denoting predicate under the current marking. The meaning or definition of an operation is specified by the constraint associated with the denoting transition.

HPrTNs were originally developed for structured analysis that provides separate mechanisms for data abstraction and processing abstraction through super predicates and super transitions, respectively. Therefore, we can use a super predicate and super transition pair in an HPrTN to capture the notion of a class although it is adequate to define a class by using a super predicate when there is no externally visible operation or using a super transition when there is no externally visible attribute. This view of class is a major improvement over the view in (He and Ding, 1996), where a class was represented by a super predicate only. In this view, the interface of the class is defined by the super predicate and the super transition. The super predicate defines data and internal operations of the class, while the super transition mainly defines the externally visible operations of the class. The corresponding subnets further define the internal structures of the data and the operations. The net inscription defines the meanings of net components through predicate types, token values, and transition constraints. When the resulting HPrTN is simple enough, there is no need to separate the super nodes from their subnets (i.e., the subnets are directly embedded inside the super nodes). An attribute or operation is externally visible if the corresponding denoting predicate or transition is an interface node (i.e., connected with a nonterminating arc). It should be noted, however, that not every super predicate or transition needs to be considered as a class. A super predicate or transition may simply denote a data abstraction or operation abstraction as originally intended; for example, a super predicate can be used to hide the internal states of an attribute that is defined by several related predicates, and a super transition can be used to define alternative implementations of an operation to realize operation overloading or overriding. Thus, our approach supports the coexistence of various modeling paradigms.

Based on the above analysis, we use the following C + +-like class schema to document a class defined by the super node(s) in an HPrTN (it is worth noting that the class schema is only used for understanding purpose and does not add functionality to the given HPrTN:

```
class Name [:superclass(es)]
{ public:
   predicates and transitions
[private:
   predicates and transitions],
}
```

where brackets [. . .] denote optional items. Predicates and transitions listed in both public and private sections are those contained in the super node(s). The name(s) of the super node(s) are used to form the class name.

In the HPrTN shown in Figure 9.3, both super transition Phil and super predicate Chop can be viewed as classes. Thus, the following class definitions can be obtained:

```
class Chop
{ public:
   Avail, Used
},
class Phil
{ public:
   Pick up, Put down
   private:
   Thinking, Eating
}.
```

Objects

An instance or object of a class has its own copy of data while sharing operations with other objects of the same class. To distinguish an object from other objects, a unique identifier is needed for each object.

In an HPrTN, an object is essentially defined by a set of tokens related through the same identifier, thus, the sort of any predicate p needs to contain a component sort of relevant identifiers (i.e., Type $(p) = p(ID \times \ldots)$. Different objects of a class share the same class data structures (i.e., tokens with different identifiers can reside in a predicate at the same time in an HPrTN). In general, however, objects of the same class cannot interact with each other directly. The above problem can be easily solved by defining a subexpression comparing token identifiers in the constraint of each transition. Movements of tokens and/or changes of token values while maintaining the object identifier indicate state changes of the object.

In the HPrTN shown in Figure 9.3, there are k philosopher objects with identifiers of sort PHIL, and there are k chopstick objects with identifiers of sort CHOP.

Class Reference Relation

Classes work together to fulfill the functionality of the underlying system. A class can use the operations and/or data provided by other classes.

In HPrTNs, the interface of a transition includes a box with a name and the labels of relevant arcs (the label identifiers determining the calling context and the flow expressions specifying parameters). The meaning of an elementary transition is defined by its constraint, and the meaning of a super transition is defined through its corresponding subnet.

It is easy to model a class reference by adding some arc when a class needs to access some public attribute of another class. For example, Figure 9.4 illustrates simple class reference relationships where some operation in class C_1 (p_1 and t_1) uses some public attributes defined in class *C2* (p_2 and t_2), and some operation in class C_2 uses some public attributes defined in class C_1. Figure 9.3 contains simple reference relationships between classes Phil and Chop.

To define an operation in one class using another operation in a different class, we cannot simply add an arc since Petri nets do not allow direct connections between transitions. As discussed earlier, there are two main ways to handle class reference relationships in the existing research works: (1) to fuse the two operations in two classes into one such that only synchronized communication is allowed (Biberstein and Buchs, 1995; Battison *et al.*, 1995; Lakos, 1995b) and (2) to create some places inside one class to hold parameters to simulate message passing and function calls (Bastida, 1995; Lakos, 1995b) which supports asynchronous communication. HPrTNs support both synchronous and asynchronous communications through reference places to model different communication protocols. These reference predicates do not belong to any class and can be viewed as connectors in software architecture languages (Shaw and Garlan, 1996). It is quite easy to model a function call through passing two messages by using one reference predicate to hold the input values and another to hold output results; another easy way to model a function call is by defining the calling operation (function) as a super transition whose subnet has at least two transitions to handle sending and receiving values.

Figure 9.5 shows the general pattern of a function call from class C_1 containing t_1 to class C_2 containing t_2, in which p_1 and p_2 are reference places. The above pattern defines a one-way **synchronized communication** (i.e., the caller must wait for the callee to continue its execution), whereas a simple message passing from an operation in one class to an operation in another class in general defines an **asynchronous communication**.

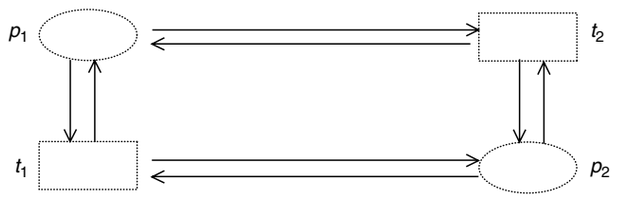

FIGURE 9.4 Simple Public Attributes Access.

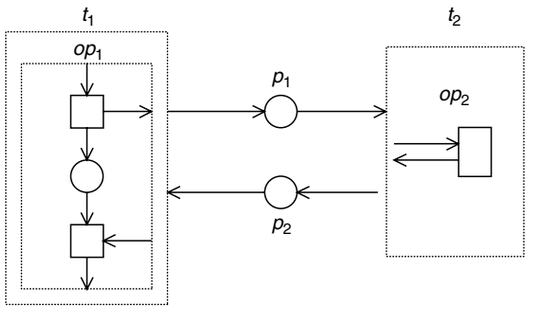

FIGURE 9.5 Reference Through Function Call.

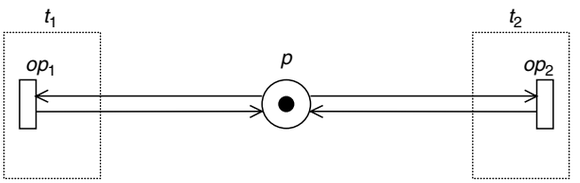

FIGURE 9.6 Synchronized Communication.

Figure 9.6 defines a general synchronization pattern such that two operations in class C_1 containing t_1 and C_2 containing t_2 must execute mutual exclusively, where p is a reference predicate with an initial dummy token.

It is quite natural and easy to define class reference relationships by using the decomposition and synthesis techniques of HPrTNs discussed in He and Lee (1991).

Class Inheritance Relation

Another major feature of the OO paradigm is **class inheritance relation** that captures the generalization–specialization relationships in the real world (Coad and Yourdon, 1991). A class inheritance relationship exists between a superclass and a subclass such that the subclass inherits data structures as well as operation definitions from the superclass without the need to define them again. Thus, class inheritance relation supports a flexible and managed way to reuse existing data structures and operations.

A class inheritance relation is realized in HPrTNs through the reuse of the net structures of inherited super nodes and the net inscription of inherited elementary nodes (the sorts of predicates, the label expressions of relevant arcs, and the constraints of transitions) defined in an existing HPrTN denoting a class. The inherited predicates and transitions, however are explicitly represented or embedded in the subclass to clearly define its role, the same convention was used by Lakos (1995b). An inherited element in a subclass has a name of the form

super_node.element_name, where super_node is the partial name of the superclass and element_name is the internal name of the element in the superclass. Renaming of relevant arcs are also necessary to reflect the current context and to ensure flow balance. It is clear that inheritance does not reduce the size of an HPrTN specification since inherited elements are embedded; an alternative way to embedding is through delegation, as explained by Abade and Cardelle (1996). However, the advantages are obvious since the meaning or structure (the most difficult part in writing an HPrTN specification) of an inherited element is already available and is obtained without any additional effort, furthermore, many known properties of the inherited element might be maintained through inheritance (structural properties are surely kept, but behavioral properties may need additional validation). It is worth noting that (1) only public components of a superclass can be inherited; (2) inheritances from multiple superclasses are supported, and an element can be inherited by multiple subclasses because no ambiguity will occur due to the naming convention; and (3) a redefined (overriding) operation is considered as a new operation in a subclass and is distinguished from an inherited operation such that an overriding operation in a subclass has the same name as the overriden operation in the superclass, this distinction between inheritance and overriding was also made by Abadi and Cardelli (1996).

Figure 9.7 shows a class inheritance relationship defined as follows:

class p_1 and t_1
{ public:
p_2, t_2, t_3
};
class p_3 and t_3: p_1 and t_1
{ public:
t_3, $p_1 \cdot p_2$, $t_1 \cdot t_2$
private:
p_4, p_5, t_6
}.

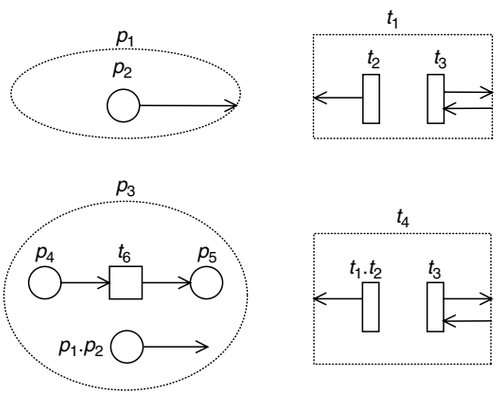

FIGURE 9.7 An Example of Class Inheritance Relation.

Polymorphism

OO paradigm also supports **polymorphism** such that an operation's name (with possibly different signatures) may have different meanings or behaviors (implementations) through inheritance or overriding.

Polymorphism can be achieved in HPrTNs in two different yet related ways. First, polymorphism is a major feature of the underlying many-sorted algebra H of an HPrTN; detailed discussions of algebraic specifications and polymorphism can be found in work by Ehrig and makr, (1985). The same operation symbol in H is used for many derived sorts. A simple example is the overloaded equality ($=$) operator when an algebraic specification H contains two elementary data types (or classes) INT and CHAR with a single parameterized definition of the equality ($=$). Second, polymorphism can be accomplished through net structure and inscription. An operation provides overriding capability if its constraint distinguishes a superclass object and a subclass object (or two objects from two different subclasses with the same superclass) and processes them differently. To realize polymorphism in an HPrTN, a shared predicate can be used to hold tokens of a superclass as well as tokens of subclasses, and the shared predicate is connected to the transition defined in the superclass and its inherited (or overriding) versions in the subclasses. The constraint of the original transition is only satisfied by the tokens of the superclass, and the constraint of each inherited (or overriding) transition is only satisfied with tokens of the subclass containing the transition.

Au: Correct term?

Figure 9.8 shows the general pattern of realizing polymorphism through net structure in HPrTNs, in which p is a shared place, t_1 is a part of the superclass, and t_2 is a part of the subclass, and op_1^* is either an inherited or an overriding version of op_1, as shown in Figure 9.8.

Furthermore, operation overriding can be achieved through a super transition in an HPrTN such that the firing of a particular component transition is determined partly by the (dynamic) instantiation of an object identifier (and thus its sort). The use of the above case-like net structure in realizing polymorphism can be avoided in the implementation of an HPrTN.

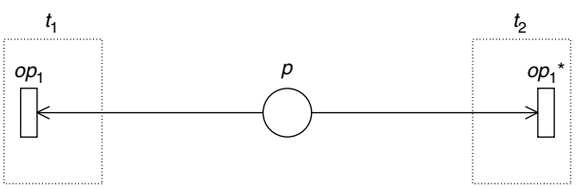

FIGURE 9.8 Polymorphism Through Choice Structure.

9.6 Fuzzy-Timing High-Level Petri Nets

9.6.1 Definition of FTHN

A Fuzzy-timing high-level petric net (FTHN) is a structure:

$$FN = (N, D, FT),$$

where N is an HLPN defined in Section 9.2; D is the set of all fuzzy delays, $d_{tp}(\tau)$, associated with arcs (t, p) from each transition $t \in T$ to its output place p; and FT is the set of all **fuzzy time stamps**. A **fuzzy time stamp** $\pi(\tau) \in FT$ is associated with each token and each place. A fuzzy time stamp $\pi(\tau)$ is a **fuzzy time function** or **possibility distribution** giving the numerical estimate of the possibility that a particular token arrives at time τ in a particular place. Any type of possibility distribution can be represented or approximated by using a number of trapezoidal distributions. Thus, we use the trapezoidal **possibility distribution** specified by the five parameters, $h(a, b, c, d)$ as shown in Figure 9.9, where h is the height having the following properties: $0 \leq h \leq 1$, $h = 1$ for an event (arrival of a token) that has occurred or will occur and $h < 1$ for an event that will not necessarily occur. A so-called **fuzzy number** is represented by the triangular distribution $h(a, b, b, d)$, which is a special case of the trapezoidal form with $b = c$. A deterministic interval between a and d denoted $[a, d]$ can be represented by (a, a, d, d), a special case of trapezoidal form with $a = b$, $c = d$, and $h = 1$. In addition, given an arbitrary-shaped possibility distribution, we can approximate it with the union of a number of trapezoidal distributions (Zhou and Murata, 1999).

In Zhou *et al.* (2000), *FTHN* is extended by adding a time interval with a possibility value p in the form of $p[\alpha, \beta]$ to each transition. That is, each transition is associated with a firing interval denoted $p[\alpha, \beta]$, where the default interval is $1[0, 0]$ (a transition definitely fires as soon as it is enabled). If a transition t is enabled at time instant τ, it may not fire before time instant $\tau + \alpha$ and must fire before or at time instant $\tau + \beta$. Possibility p is a value in the interval [0,1], where p is 1 if transition t is not in structural conflict with any other transition, and p can be less than 1 when we want to assign different chances to transitions in conflict. Here, '**structural conflict**' means that a transition t shares some input place with another transition that can be enabled simultaneously with transition t; firing one transition will disable the other transition.

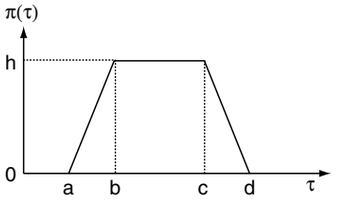

FIGURE 9.9 Trapezoidal Possibility Distribution.

9.6.2 Computation for Updating Fuzzy Time Stamps

Suppose that a transition t is enabled by n tokens and the fuzzy enabling time $e_t(\tau)$ of transition t is computed by $e_t(\tau) = \text{latest}\ \{\pi_i(\tau),\ i = 1, 2, \ldots, n\}$, where latest is the operator that constructs the "latest-arrival-lowest-possibility distribution" from n distributions (Murata, 1996; Murata *et al.*, 1999). The $\pi_i(\tau)$ is the fuzzy time stamp to which the enabling token arrives at the ith input place of transition t. When there are m transitions in structural conflict that are enabled with their fuzzy enabling times, $e_i(\tau)$, $i = 1, 2, \ldots, t, \ldots, m$, and with their possibility intervals, $p_i[\alpha_i, \beta_i]$, we compute the fuzzy occurrence time $o_1(\tau)$ of transition t whose fuzzy enabling time is $e_t(\tau)$ as follows:

$$o_t(\tau) = \min\{e_t(\tau) \oplus p_i(\alpha_t, \alpha_t, \beta_t, \beta_t),\ \text{earliest}\ \{e_i(\tau) \oplus p_i(\alpha_i, \alpha_i, \beta_i, \beta_i),\ i = 1, 2, \ldots, t, \ldots, m\}\},$$

where **earliest** is the operator that constructs the "earliest-arrival-highest-possibility distribution" from m distributions, (Murata, 1996; Murata *et al.*, 1999), **min** denotes the minimum or intersection operation, and $\oplus$ is the extended addition (Dubois and Prade, 1989). We compute the fuzzy time stamp $\pi_{tp}(\tau)$, which is the fuzzy time distribution at which a token arrives at the transition t output place p, as follows:

$$\pi_{tp}(\tau) = o_t(\tau) \oplus d_{tp}(\tau) = h_1(o_1, o_2, o_3, o_4) \oplus h_2(d_1, d_2, d_3, d_4) = \min\{h_1, h_2\}(o_1 + d_1, o_2 + d_2, o_3 + d_3, o_4 + d_4),$$

where $d_{tp}(\tau)$ is the fuzzy delay associated with the arc (t, p) (Murded, 1996). When there are no transitions in conflict with transition t with its possibility interval $p_t[\alpha t, \beta_t]$, the fuzzy occurrence time is given by $o_t(\tau) = e_t(\tau) \oplus (\alpha_t, \alpha_t, \beta_t, \beta_t)$, where we set $p_t = 1$ since no conflict exists. We use the following formulas as approximate computations of the earliest and latest operations:

$$\text{earliest}\ \{e_i(\tau),\ i = 1, 2, \ldots, n\} = \text{earliest}\{h_i(e_{i1}, e_{i2}, e_{i3}, e_{i4}),\ i = 1, 2, \ldots, n\} = \max\{h_i\}(\min\{e_{i1}\},\ \min(e_{i2},\ \min\{e_{i3},\ \min\{e_{i4}\}),\ i = 1, 2, \ldots, n$$

$$\text{latest}\{\pi_i(\tau),\ i = 1, 2, \ldots, n\} = \text{latest}\{h_i(\pi_{i1}, \pi_{i2}, \pi_{i3}, \pi_{i4}),\ i = 1, 2, \ldots, n\} = \min\{h_i\}(\max\{\pi_{i1}\}, \max\{\pi_{i2}\}, \max\{\pi_{i3}\}, \max\{\pi_{i4}\}),\ i = 1, 2, \ldots, n.$$

Using the above procedure, we compute and update fuzzy time stamps $\pi(\tau)$, fuzzy enabling times $e(\tau)$, and fuzzy occurrence times $o(\tau)$ each time a transition firing (atomic action) occurs, starting from the initial (given) fuzzy timestamps of tokens in the initial marking M_0 and initially specified fuzzy delays. $d_{tp}(\tau)$. (See Zhou *et al.* [2000].)

9.6.3 Intended Application Areas and Application Examples of FTHNs

As seen in Subsections 9.6.1 and 9.6.2, the essence of FTHNs is the computation of updating fuzzy time stamps, and this computation involves only additions and comparisons of real numbers. Thus, computation can be done very fast, and FTHNs are suitable for performance analysis in real-time applications. The notion of fuzzy timing is highly flexible in that it can capture imprecise or incomplete knowledge regarding time behavior with specified or given possibility distributions, as well as the more conventional deterministic and probabilistic knowledge. However, it is expected that the traditional probabilistic approach and the *FTHN* method using possibility theory are complementary, rather than competitive, as possibility theory is considered to be complementary to probability theory (Zadeh, 1995). We expect that the FTHN method is more scalable than the traditional stochastic approaches because the *FTHN* computations are done using real arithmetic operations.

Some examples of FTHN applications include the following. A real-time network protocol used in local area networks (LANs) is modeled using FTHNs in (Murata, *et al.*, 1999). Here we are interested in evaluating the worst-case performance in a given network, where propagation delays, times for processing message frames, and other such processes are specified as trapezoidal fuzzy-time functions. The fuzziness is due to the uncertain length (thus uncertain delay) of each message. In addition, FTHN models are used for performance evaluation of manufacturing systems, where the abilities of machines and/or workers involved in a manufacturing process are fuzzily known (Watanuki, 1999). Another area in which FTHN models are applied is the synchronization of multimedia systems (Zhou and Murata, 1998, 2001). Multimedia systems integrate a variety of media with different temporal characteristics (e.g., time-dependent media, such as video, audio, or animation) and time-independent media (e.g. text, graphics, and images). Thus, synchronization is a critical issue in these systems. The temporal specification has to be properly represented for presentation reviewing and planning by the user as well as for storing purposes. Multimedia synchronization has a time-critical problem because it must guarantee the temporal constraints for presenting the media items. Thus, there is a benefit in using FTHN methods to specify and analyze the temporal relations and specifications. The FTHN method has been used to present a new fine-grained temporal FTHN model for distributed multimedia synchronization (Zhou and Murata, 2001). The FTHN method has also been applied to a video-on-demand (VOD) system, which is a continuous playback multimedia application in which constant real-time media data are required for smooth real-time presentation. Thus, timing is a critical issue, and the response time is the only timing that has direct interaction with the subscribers and also impacts the quality-of-service (QoS), as discussed by Murata and Chen (2000). A nontrivial example of FTHN application is found in Zhou *et al.* (2000), where the FTHN method has been used to model networked virtual-reality systems, such as Cave Automatic Virtual Environment (CAVE) and Narrative Immersive Constructionist/Collaborative Environments (NICE) projects at the University of Illinois at Chicago. Using the FTHN models, various simulations have been conducted to study real-time behaviors, network effects, and performance (latencies and jitters) of the *NICE*. The simulation results are consistent with data and measurements obtained experimentally. This study shows how powerful FTHN models are in specifying and analyzing performance and how helpful the performance analysis is in improving a real-time networked-virtual-environment system design process.

References

Abadi, M., and Cardelli, L. (1996). *A theory of objects*. Springer-Verlag. Au: City & State Abbrev.?

Agha, G., De Cindio, F., Rozenberg, G. (Eds.). (2001). Concurrent object-oriented programming and Petri nets—Advances in Petri nets. *Lecture Notes in Computer Science*. Springer Verlag,

Anttila, M., Eriksson, H., and Ikonen, J. (1983). Tools and studies of formal techniques—Petri nets and temporal "logic." In H. Rudin and C.H. West (Eds.) *Protocol specification, testing, verification.* Elsevier Science. Au: City & State Abbrev.? Au: City & State Abbrev.?

Alpein, B., and Schneider, F. B. (1985). Defining liveness. *Information Processing Letters 21*, 181–185.

Bastide, R. (1995). Approaches in unifying Petri nets and the object-oriented approach. *Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency* Vol. 000–000.

Biberstein, O., and Buchs, D. (1995). Structured algebraic nets with object-orientation. *Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency* Vol. 000–000.

Battiston, E., Chizzoni, A., and Cindio, F. D. Inheritance and concurrency in CLOWN. *Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency.* Vol. 000–000.

Benjamin, M. (1990). A message passing System: An example of combining CSP and Z. *Proceedings of the 5th Annual Z Users Workshop* Vol. 000–000, 221–228.

Booch, G. (1994). *Object-oriented analysis and design with applications.* (2d. ed.). Benjamin/Cummings. Au: City & State Abbrev.?

Booch, G., Rumbaugh, J., and Jacobson, I. (1997). *Unified modeling language user guide.* Reading, MA: Addison-Wesley.

Clarke, E., and Wing, J. (1996). Formal methods: State of the art and future. *ACM Computing Surveys 28*(4), 626–643.

Coad, P., and Yourdon, E. (1991). *Object-oriented analysis.* Yourdon Press. Au: City & State Abbrev.?

Diaz, M., Guidacci, G., and Silveriral, D. (1983). On the specification and validation of protocols by temporal logic and nets. *Information Processing 83*, 47–52.

Dong, Z., and He, X. (2001). Integrating UML statechart and collaboration diagrams using hierarchical predicate transition nets. *Lecture Notes in Informatics*, *P-7*, 99–112.

Dubois, D., and Prade, H. (1989). Processing fuzzy temporal knowledge. *IEEE Transactions On Systems, Man and Cybernetics*, *19*(4), 729–744.

Duke, R., and Smith, G. (1989). Temporal logic and Z specifications. *Australian Computer Journal 21*(2), 62–69.

Ehrig, H., and Mahr, B. (1985). *Fundamentals of algebraic specification 1—Equations and initial semantics.* Pringer-Verlag.

Evans, A. S. (1997). An improved recipe for specifying reactive systems in Z. *Proceedings of the 10th International Conference of Z Users* (*Lecture Notes in Computer Science*) *1212*, 273–294.

Genrich, H. J., and Lautenbach, K. (1981). System modeling with high-level Petri nets. *Theoretical Computer Science 13*, 109–136.

Haas, P. (2002). *Stochastic Petri nets: Modeling, stability, simulation.* Springer-Verlag.

Harel, P. (2002). On visual formalisms. *Communications of the ACM 31*, 514–530.

He, X., and Ding, Y. (1992). A temporal logic approach for analyzing safety properties of predicate transition nets. *Proceedings of the 12th IFIP World Computer Congress (Information Processing '92)* Vol. 000–000, 127–133.

He, X., and Ding, Y. (1996). Object-oriented specification using hierarchical predicate transition nets. *Proceedings of the 2nd International Workshop on Object-Oriented Programming and Models of Concurrency* Vol. 000–000, 72–79.

He, X., and Ding, Y. (2001). Object orientation in hierarchical predicate transition nets. *Lecture Notes in Computer Science* Vol. 000–000, 196–215.

He, X. (1991). Specifying and verifying real-time systems using time Petri nets and real-Time temporal logic. *Proceedings of the 6th Annual Conference on Computer Assurance* Vol. 000–000, 135–140.

He, X. (1992). Temporal predicate transition nets—A new formalism for specifying and verifying concurrent systems. *International Journal of Computer Mathematics 45* (1/2), 171–184.

He, X. (1995). A method for analyzing properties of hierarchical predicate transition nets. *Proceedings of the 19th Annual International Computer Software and Applications Conference* (*COMPSAC* '95) Vol. 000–000, 50–55.

He, X. (1996). A formal definition of hierarchical predicate transition nets. *Proceedings of the 17th International Conference on Application and Theory of Petri Nets (ICATPN'96), Lecture Notes in Computer Science 1091*, 212–229.

He, X. (1998). Transformations on hierarchical predicate transition nets: Abstractions and refinements. *Proceedings of the 22nd International Computer Software and Application Conference* (*COMPSAC* '98) Vol. 000–000, 164–169.

He, X. (2000a). Formalizing use case diagrams in hierarchical predicate transition nets. *Proceedings of the IFIP 16th World Computer Congress* Vol. 000–000, 484–491.

He, X. (2000b). Formalizing class diagrams using hierarchical predicate transition nets. *Proceedings of the 24th International Computer Software and Application Conference* (*COMPSAC* '2000). Vol. 000–000.

He, X. (2000c). Translating hierarchical predicate transition nets into CC++ programs. *Information and Software Technology 42*(7), 475–488.

He, X. (2001). PZ nets—A formal method integrating Petri nets with Z. *Information and Software Technology. 43*, 1–18.

He, X., and Lee, J.A.N. (1990). Integrating predicate transition nets and first-order temporal logic in the specification of concurrent systems. *Formal Aspects of Computing 2*(3), 226–246.

He, X., and Lee, J.A.N. (1991). Methodology for constructing predicate transition net specifications. *Software—Practice & Experience 21*(8), 845–875.

He, X., and Yang, C. H., Structured analysis using hierarchical predicate transition nets. *Proceedings of the 16th International Computer Software and Applications Conference (COMPSAC'92)* Vol. 000–000, 212–217.

ISO/IEC. (2002). High-level Petri nets–Concepts, definitions, and graphical notation. *Final Draft International Standard* 15909, version 4.7.1.

Jensen, K. (1992). *Coloured Petri nets—Basic concepts, analysis methods, and practical use.* Springer-Verlag.

Jensen, K. (1995). *Coloured Petri nets—Basic concepts, analysis methods, and practical use.* Springer-Verlag.

Jensen, K., and Rozenberg, G. (Eds.). (1991). *High-level Petri nets—Theory and applications.* Springer-Verlag.

Kan, C., and He, X. (1995). High-level algebraic Petri nets. *Information and Software Technology 37* (1), 23–30.

Kan, C., and He, X. (1996). A method for constructing algebraic Petri nets. *Journal of Systems and Software 35*, 12–27.

Kappel, G., and Schrefl, M. (1991). Using an object-oriented diagram technique for the design of information systems. In *Dynamic modeling of information systems.* Elsevier Science Publishers.

Lewandowski, S., and He, X. (1998). A Java framework for implementing hierarchical predicate transition nets. *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering (SEKE'98)* Vol. 000–000, 261–268.

Lewandowski, S., and He, X. (2000). Automating the generation of code for a hierarchical predicate transition net-based design. *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering.* Vol. 000–000 pp 000–000.

Lakos, C. (1995a). From colored Petri nets to object Petri nets: *Proceedings of the 16th International Conference on the Application and Theory of Petri Nets.* Vol. 000–000 pp 000–000.

Lakos, C. (1995b). The object orientation of object Petri nets. *Proceedings of the 1st Workshop on Object-Oriented Programming and Models of Concurrency* Vol. 000–000 pp 000–000.

Lamport, L. (1994a). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16* 872–923.

Lamport, L. (1994b). TLZ *Proceedings of the 1994 Z User Workshop* Vol. 267–268.

Lee, Y.K., and Park, S.J. (1993). OPNets: An object-oriented high-level Petri net model for real-time system modeling. *Journal of Systems and Software 20*, 69–86.

Mandrioli, D., Morzenti, A., Pezze, M., Pietro, P., and Silva, S. (1996). A Petri net and logic approach to the specification and verification of real-time systems. *Formal Methods for Real-time Computing* Vol. 000–000 pp 000–000.

Manna, Z., and Pnueli, A. (1995). *The temporal verification of reactive systems—safety.* Springer-Verlag, Vol. 000–000 pp 000–000.

Manna, Z., and Pnuell, A. (1992). *The temporal logic of reactive and concurrent systems—specification.* New York: Springer-Verlag Vol. 000–000 pp 000–000.

Marsan, M., Balbo, G., Conte, G., Donatelli, S., and Franceschinis, G., (1994). *Modeling with generalized stochastic Petri nets.* New York: John Wiley & Sons Vol. 000–000 pp 000–000.

Au: City & State Abbrev.?
Au: City & State Abbrev.?
Au: City & State Abbrev.?
Au: City & State Abbrev.?
Au: City & State Abbrev.?

Matsuoka, S., and Yonezawa, A. (1993). Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, (Eds.). *Research directions in concurrent object-oriented programming*. MIT Press. Vol. 000–000 pp 000–000.

Medvidovic, N., and Taylor, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering 26* (1), 70–93.

Murata, T. (1996). Temporal uncertainty and fuzzy-timing high-level Petri nets. *Application and theory of Petri nets: Lecture Notes in Computer Science, 1091*, 11–28.

Murata, T. (1989). Petri nets, properties, analysis and applications. *Proceedings of IEEE, 77*(4), 541–580.

Murata, T., and Chen, C.P. (2000). Fuzzy-timing Petri-net modeling and analysis of video-on-demand system response times. *Proceedings of the 5th World Conference on Integrated Design & Process Technology*, Vol. 298–306.

Murata, T., Suzuki, T., and Shatz, S. (1999). Fuzzy-timing high-level Petri nets (FTHNs) for time-critical systems. In J. Cardoso and H. Camargo (Eds.). *Fuzziness in Petri nets*, Vol. 22: *Studies in Fuzziness and Soft Computing*. New York: Springer Verlag.

Queille, I.P., and Sifakis, J. (1982). Specification and verification of concurrent systems in CESAR. *Lecture Notes in Computer Sciences 137*, 337–351.

Reisig, W. (1987). Petri nets in software engineering. *Lecture Notes in Computer Science 255*, 63–96.

Shaw, M., and Garlan, D. (1996). *Software architecture*. Englewood Cuffs, NJ: Prentice-Hall.

Suzuki, L., and Lu, H. (1989). Temporal Petri nets and their application to modeling and analysis of a handshake daisy chain arbiter. *IEEE Transactions on Computer 38*(5), 696–704.

Spivey, J.M. (1992). *The Z notation: A reference manual*. Englewood Cliffs, NJ: Prentice-Hall.

Reisig, W. (1985a). Petri nets—An introduction. *EATCS Monographs on Theoretical Computer Science 4*,

Reisig, W. (1985b). On the semantics of Petri nets. In J. Neuhold and G. Chroust (Eds.), *Formal models in programming* North-Holland.

Au: City & State Abbrev.?

Taguchi, K., and Araki, K. (1997). The state-based CCS semantics for concurrent Z specification. *Proceedings of the 1st International Conference on Formal Engineering Methods* Vol. 283–292.

Stroustrup, B. (1991). *The C++Programming Language*. (2nd ed.). Reading, MA: Addison-Wesley.

van Hee, K.M., Somers, L.J., and Voorhoeve, M. (1991). Z and high-level Petri nets. *Lecture Notes in Computer Science 551*, 204–219.

Wang, J. (1998). *Timed Petri nets, theory and application*. Kluwer Academic Publisher.

Watanuki, K., and Murata, T. (1999). Evaluation method for assembly. Norwell, MA: disassembly by Petri nets. *Proceedings of the International Conference on Engineering Design* (ICED'99) 1, 519–522.

Yourdon, E. (1989). *Modern structured analysis*. Englewood Cliffs, NJ: Prentice Hall.

Zadeh, L.A. (1995). Discussion: Probability theory and fuzzy logic are complementary rather than competitive. *Technometrics of American Statistical Association and American Society for Quality Control 37*(3), 00

Zhou, Y., and Murata, T. (2001). Modeling and analysis of distributed multimedia synchronization by extended fuzzy-timing Petri nets. *Journal of Integrated Design and Process Science 4*(4), 23–38.

Zhou, Y., and Murata, T. (1998). Fuzzy-timing Petri net model for distributed multimedia synchronization. *Proceedings of the 1998 IEEE International Conference on Systems, Man, and Cybernetics*. Vol. 244–249.

Zhou, Y., and Murata, T. (1999). Petri net model with fuzzy-timing and fuzzy-metric temporal logic. *International Journal of Intelligent Systems 14*(8), 719–746.

Zhou, Y., Murata, T., and DeFanti, T. (2000). Modeling and performance analysis using extended fuzzy-timing Petri nets for networked virtual environments. *IEEE Transactions on Systems, Man, and Cybernetics 30*(5), 737–756.