Manufacturing Systems Modeling Tools
Based on Object-oriented Petri Nets

The paper proposes an approach, called OPNets, for modeling and validating manufacturing systems that is based on object-oriented high-level Petri nets. In OPNets, modeling components of Petri net are constructed into hierarchical objects that communicate with each other by passing messages. To enhance the reusability and maintainability, OPNets organizes a system into hierarchical objects that inherit attributes and behavioral properties from the object of super class and object-interaction relations are separated from the internal structure of object. The modeling scheme of OPNets tries to resolve the complexity problems of Petri net. To illustrate the modeling schemes of OPNets, a storage/retrieval example has been proposed.
I. INTRODUCTION

Petri nets have been found to be useful for describing and analyzing real-time systems such as manufacturing and robotics systems [Baldassari and Bruno 1988; Camuri and Franchi 1990] that are usually characterized by distributed and concurrent nature. However, the complexity of the model is drastically increased as the number of reachable states and events in Petri nets grows [Murata 1989]. Therefore the complexity problem is one of the main obstacles in applying Petri nets to large complex manufacturing systems.

As attempts to resolve the problem, high level Petri nets [Genrich and Lautenbach 1981; Sibertin-Blanc 1985] and net reduction methods [Lee and Favrel 1985] are developed. High level Petri nets, which generalize tokens as identifiable objects and inscribe expressions on transitions, have been recognized as suitable formalisms to integrate the phases of specification, simulation, prototyping of discrete event dynamic systems such as process control systems, and real-time system modeling. High level Petri nets, however, often fail to reduce the systems into of manageable size. Petri net reduction is another way of solving the problem that reduces the system model to a simpler one, while preserving the system properties to be analyzed. However, they basically reduce the graphic structure of the net, disregarding the structural properties of the system: a system is usually composed of its subparts that have their own behaviors and communicate with each other through exchanging messages.

More fundamental solution to the complexity problem can be provided by object-oriented approach in which objects are regarded as entities that have their own data and actions to be carried out in response to incoming messages. Object-oriented approach also emphasizes the importance of inter-object behavior. It is recognized that the well organized inter-object behavior enhances the maintainability and reusability. The inter-object behavior should be designed such that each object is organized as independent as possible by decoupling the communication knowledge. An object-oriented high level Petri nets called OPNets is proposed in the paper to resolve the complexity problem. It enhances the maintainability by organizing a system into concurrent objects and separating the synchronization constraints from the internal structure of each object.
II. OBJECT ORIENTED HIGH LEVEL PETRI NETS

Object-oriented high-level Petri nets called OPNets is developed to manage the complexity problem and hence to increase the maintainability of system modeling. To briefly introduce the OPNets, concepts of system, object, external and internal structures of object, and relation defined in the OPNets are explained in the following sections. The detailed explanations of OPNets are given in [Lee and Park 1993].

1. Systems

In OPNets, a system is composed of mutually communicating hierarchical objects and their interconnection relations. Objects in OPNets are hierarchically organized entities and incorporate aggregation and classification concepts of object-oriented approach. Communications between objects are supported by a set of links called interconnection relations that connect related objects. In Figure 1, objects are represented by O and the interconnection relations R are represented by gates g and their input and output flow relations.

\[ SYSTEM = (O, R), \]

where
\[ O : \text{a set of objects}, \]
\[ R : \text{a set of interconnection relations}. \]

2. Objects

Each object has an external structure and an internal structure that are separated for information hiding. External structure is designed for the message communications between objects, whereas the internal control flow of each object is represented by the internal structure. As shown in Figure 1, the internal structures of objects, except O2, are not identified from outside while the interconnection relations between objects are represented externally, where objects are represented by rounded boxes. The internal control flows of O2 are also externally hidden, but shown in Figure 1 for the illustration of the internal structure.

2.1 External Structure of Objects
The external structure of an object \( O_i \subset O \) is represented by the 6-tuple,
\[ O_i = (H_i, IG_i, OG_i, MI_i, OM_i, F_i), \]
where
\[ H_i : \text{an object hierarchy}, \]
\[ IG_i : \text{a set of input gates of object } O_i. \]
OG: a set of output gates of object O.
IM: a set of input message queues of object O.
OM: a set of output message queues of object O.
F: a set of flow relations of object O.

Parent objects are specified in an object hierarchy to incorporate inheritance across the object hierarchy. Gates, which are non-empty subset of transitions in Petri nets, execute message communications between objects by firing. Input gates and output gates perform incoming and outgoing message communications, respectively. Graphically, gate gi is represented by a thick solid bar as shown in Figure 1. A message queue is a place and can be regarded as an input and output window through which communications between outside objects and the actions of the object are possible. Message queues are composed of reply queue to model wait—and-reply mechanism and synchronization queues to restrict the transition firing sequences between objects. Graphically, message queues are represented by small ovals coming out from objects where the single and double ovals represent the synchronization and reply queues, respectively. Flow relations are represented by arrows that connect input gates and input message queues, or output message queues and output gates.

![Figure 1. Graphical Representation of OPNet Structure](image)

2.2 Internal Structure of Objects

Two types of objects are defined in OPNets: primitive object and composite object. A primitive object is a basic entity for behavior representation in which static
properties and dynamic behaviors are defined. A composite object is an aggregate of more than one primitive objects and/or other composite objects. Figure 1 depicts a composite object \( O_1 \) that is an aggregate of a primitive object \( O_{21} \) and a composite object \( O_{22} \). Detailed discussions of internal structures of a composite object and a primitive object will be given in the following section.

2.3 Internal Structure of a Composite Object

Internal structure of a composite object defines a set of objects contained in the composite object and their interconnection relations. Let \( CO_i \) be a composite object \( i \) and \( PO_i \) denotes a primitive object \( i \). Then object \( O \) equals \( CO \cup PO_i \), where \( CO = \bigcup_i CO_i \) and \( PO = \bigcup_i PO_i \). The internal structure of a composite object \( CO_n, ICO_n \), is characterized by the following 3-tuples:

\[ ICO_i = (X, Y, R_i), \]

where
\[ X = \{ X \mid X \in P(CO), CO_i \in X \}, \]
\[ Y = \{ Y \mid Y \in P(PO_i) \}, \]
\[ R_i : a \text{ set of interconnection relations}. \]

\( X \) is an element of power set of \( CO \), excluding sets containing \( CO_i \) itself, and \( Y \) is an element of power set of \( PO \). Interconnection relations between these objects are represented by gates and their input and output flow relations.

2.4 Internal Structure of a Primitive Object

For each primitive object, static properties and dynamic behaviors must be clearly specified for the complete and explicit modeling of control structures. Static properties include algebraically specified data structures, while dynamic behaviors show the partial ordering of actions and the influence of the object state on its actions, which implies that actions of an object may only be enabled when the object is in a specific state, and must be delayed until the object is in a state consistent with the execution of the actions. For example, if a buffer is empty, then the state of the buffer is inconsistent with the execution of a deque operation to remove an item from the buffer. The dynamic behavior of a primitive object \( O_{21} \) is shown in Figure 1.

An internal structure of a primitive object \( PO_i \) can be defined as follows:

\[ IPO_i = (D_i, SV_i, S_i, AT_i, LF_i, IN_i, M_i) \]

where
\[ D_i : a \text{ set of attributes of } PO_i, \]
\[ SV_i : a \text{ set of state variables } PO_i, \]
$S_i : \text{a set of states of } PO_i$,

$AT_i : \text{a set of action transition of } PO_i$,

$LF_i : \text{a set of local flow relations of } PO_i$,

$IN_i : \text{a set of instances of } PO_i$,

$M_0 : \text{initial marking of } PO_i$.

A set of attribute and a set of state variables constitute a data structure of a primitive object. Attributes and state variables are similar in semantics, however, the values of attributes are static and may be stored in files while the state variables are changing according to the state changes of instances. State is an non empty subset of places which represents the current status of the object. Each state is associated with an unary state predicate characterizing the state variables. The state predicate is defined by a function mapping from a particular state into a tuple of state values of a primitive object.

An action transition, which is a subset of transitions, plays a role of synchronization and performs predefined actions when precondition of action transition is met. An action in action transition represents an execution of a sequential program. Action is classified as external or internal depending on whether it provides a service to other objects or not. External actions are also divided into asynchronous actions, synchronous actions, and response actions. Asynchronous action is a side-effect free action that is instantly triggered upon receipt of request message disregarding the current state of the object, therefore, it doesn't need to be sequenced with other actions. Message queues are not required to be connected for an asynchronous action since it is invoked by other objects without any explicit interconnection relation between the server and the client. A synchronous action of an object is invoked synchronously with the external actions of the other object to which synchronization queue is connected. In addition to synchronization, a response action, to which reply queue is connected, also returns the result to the client. To internal actions, no message queues are connected since they do not provide any service to external objects. However, partial sequences between internal actions and synchronous/response actions should be established for the complete modeling of dynamic behavior.

Local flow relations are internal control flows of a primitive object with the following four types: flows from input message queues to action transitions, flows from action transitions to output message queues, flows from states to action transi-
tions, and flows from action transitions to states.

Each object has instances that are uniquely referenced by identifiers of the instances such as names. Instances are represented by tokens that are initially given to primitive objects. Tokens of instance type stand for specific instances of the object, therefore, they reside within the boundary of object and are not allowed to be created nor destroyed during the net execution. On the other hand, tokens of message type represent the messages for communications between objects that are allowed to cross the boundary of objects and hence allowed to be created or destroyed.

3. Object Interconnection Relations

In order to decouple the communication knowledge as much as possible from each object, we have adopted a scheme in which both the sender and receiver of messages may not need to know the exact communication type of the other side and the data type adaptation between the communication channels are partly supported by an intermediate transition. Therefore the communications between objects are performed by firing the intermediate transitions, i.e., gates. In Figure 1, firing of $g_i$ removes a message from the message queue of object $O_i$ and puts it into that of object $O_{2i}$.

The interconnection relation $R$ is a binary relation on the Cartesian product of the objects:

$$R \subseteq O \times O.$$

The actual interconnection of objects is established through the gates, by selecting $IG_i$’s and $OG_i$’s such that if $(O_i, O_j) \in R$, then $OG_i \cap IG_j \neq \emptyset$, where $IG_i$’s are the input gates of object $O_i$, and $OG_i$’s are the output gates of object $O_i$. That is, if $g \in IG_i$ and $g \in OG_i$ then $O_i$ and $O_i$ are connected through gate $g_i$ and $g_i$ is connected to $omq_i$ and $imq_i$, where $omq_i \in g \cdot$ and $imq_i \in g \cdot$. The $omq_i$ and $imq_i$ are called output message queue and input message queue respectively. The $g \cdot t(p \cdot)$ denotes the set of all input places (transitions) of a transition $t$ (place $p$) and $t \cdot (p \cdot)$ denotes the set of all output places (transitions) of a transition $t$ (place $p$). Then $O_i$ and $O_j$ are called a sender and a receiver of message, respectively. Thus the interconnection relations of objects can be defined as follows:

$$R_{ij} = \{(O_i, g_i, O_j) \mid g_i \in OG_i \cap IG_i\}.$$

In Figure 1, $O_i$ and $O_j$ are connected through $g_i$. 
III. AN ILLUSTRATIVE EXAMPLE

1. Problem Description

To illustrate the modeling of OPNets, an automatic storage retrieval system [Son et al. 1989] is described that furnishes working stations with appropriate part boxes stored in part box storage as shown in Figure 2. A worker picks up part boxes from a part box storage and puts them down on conveyor 1 or 2 according to the cell of an automatic storage to which the part boxes are sent. An automatic storage has 15 cells to store 6 different types of part boxes and each cell can contain about 15 to 20 boxes. Part boxes that are about to go to the cells of number 8 to 15 or 1 to 7 are laid on conveyor 1 or 2 respectively.

The worker also takes up empty boxes from conveyor 2 and then stores them on empty box storage. The part types and their appropriate cell number of an automatic storage are summarized in Table 1. When the same kind of part boxes are stored on more than one cell like part type 1, the part boxes are stored or drawn according to priority. This makes box handling operations more convenient since high priority cells are more efficient to work with. For each part box, the former cells listed in Table 1 have priority over the latter ones. When the current number
of empty part boxes in cell 5 of an automatic storage exceeds certain level, the worker is notified not to lay filled part boxes from part box storage to conveyor 2. The empty boxes in an automatic storage are laid on conveyor 2 if the conveyor 2 is emptied and is moved backward to store empty part boxes to an empty box storage.

<table>
<thead>
<tr>
<th>Part Type</th>
<th>Cell Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>11, 15, 7, 4</td>
</tr>
<tr>
<td>#2</td>
<td>12, 13, 14</td>
</tr>
<tr>
<td>#3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>#4</td>
<td>8, 9</td>
</tr>
<tr>
<td>#5</td>
<td>10</td>
</tr>
<tr>
<td>#6</td>
<td>6</td>
</tr>
<tr>
<td>empty box</td>
<td>5</td>
</tr>
</tbody>
</table>

The movement of part boxes from conveyor 1 or 2 to the appropriate cells of an automatic storage is guaranteed by the transporter 1. Transporter 1 also picks up empty boxes from cell 5 of an automatic storage and then put them down on conveyor 2 that moves them to empty box storage by reversing the moving direction. Transporter 2 releases the boxes from all the cell of an automatic storage (except cell 5) and deposit them on supply line that furnishes a part box queue of an appropriate working station with boxes. The empty boxes that have finished operations on a working station are sent to cell of number 5 by transporter 2.

Six working stations are numbered from 1 to 6 and the ith working station processes part of type i. The elapsed processing times required for the working stations are ranged from station 1 to static 6 in an increasing order. To improve the system efficiency, higher priority is given to the request for the parts that consumes less time in working station. Therefore priority is given firstly on the requests for the part boxes of type 1, secondly on type 2, ..., and finally on type 6. After the processing of an working station has been completed, a worker in the working station lays the
empty box on return line and loads filled boxes on station from part box queue, which automatically signals a request for the part box to transporter 2. The type of part box to be delivered to supply line is determined by transporter 2 according to priority. Any request for the part boxes that are not provided by an automatic storage is backlogged.

2. Modeling Process

An object-oriented development method to organize the system as autonomous objects is presented in this section. The primary criterion for the decomposition of a system is that each module in the system denotes an object. The object has its own set of applicable operations and communicates with other through message passing. The modeling process recognizes the importance of abstraction and information hiding. For each object, the external part and the internal part are developed but the internal part is not identified from outside. The steps of modeling process are explained below.

2.1 Identification of Objects

The first step is to identify the objects and their attributes in the target system. The objects are usually derived from nouns in describing the problem space. The objects, their attributes, and state variables in an automatic storage retrieval system are extracted as shown in Tables 2. Working station has two attributes (no, priority) and a state variables (states). For each instance of a working station, priority is valued according to the no of the instance and lower priority value means higher priority. There are six instances of working station which have priority values equal to their no's, that is $s_i$ with priority equals to 1, $s_i$ with priority to 2, and so on. The addresses of a part box represent the cell numbers of an automatic storage to which the part box is sent.

The address—rule of part boxes is as follows:

if type = 1 and contents = full then address = \{11, 15, 7, 4\},
if type = 2 and contents = full then address = \{12, 13, 14\},
if type = 3 and contents = full then address = \{1, 2, 3\},
if type = 4 and contents = full then address = \{8, 9\},
if type = 5 and contents = full then address = \{10\},
if type = 6 and contents = full then address = \{6\},
if contents = empty
  then address = {5} ;

There also exist simple queues to hold part boxes; they are part box storage, empty box storage, supply line, return line, and part box queue. Each queue has an attribute (queue—size) to indicate the current number of part boxes in the queue. An automatic storage is composed of 15 cells to store part boxes. Each cell has attribute (no) to identify the cell and capacity to limit the total number of part boxes allowed in the cell. Therefore the current total number of part boxes in the cell (\#of\_part) does not exceed the capacity of the cell, which is described in the construct (with constraints) attached to the end of \#of\_part. The contents—rule1 of cell (which is invoked whenever the value of \#of\_part is modified) is defined as follows :

\[
\text{if } \#\text{of}_\text{part} < \text{capacity then contents} = \text{empty} \\
\text{else contents} = \text{full} ;
\]

Conveyor 1 and conveyor 2 share the same properties: they are of same kind, and have same attributes, state variables, etc. However, each one has some specific properties: directions specified to conveyor 1 or 2. In order to reduce redundancy in modeling such objects, a super class that have common properties of its subclass is introduced and each subclass has only specific attributes and inherits the common properties from the super class. Conveyor is a super class of conveyor 1 and conveyor 2. The isA construct establishes the hierarchy. The properties of a conveyor are inherited to conveyor 1 and conveyor 2. Transporters also have the same hierarchical structure: transporter 1 and transporter 2 inherit the properties of their super class transporter.

The contents_rule2 of conveyor is declared as follows :

\[
\text{if } \#\text{of}_\text{part} = 0 \text{ then contents} = \text{empty} , \\
\text{if } 0 < \#\text{of}_\text{part} \text{ and } \#\text{of}_\text{part} < \text{capacity then contents} = \text{something} , \\
\text{if } \#\text{of}_\text{part} = \text{capacity then contents} = \text{full} ;
\]

2.2 Identification of Actions

The identification step serves to characterize the actions of each object. In order to model control structure of each object as explicitly as possible, an action represents an execution of a sequential program; hence no concurrency is allowed in
each action. The actions of objects are identified and listed in Table 3. Actions to increase the value of queue_size in objects holding part boxes are omitted in Table 3 but shown in Figure 3.

2.3 Establishment of Visibility
Once the objects and their actions have been identified, the external visibility of the objects are established as shown in Figure 3. The static dependencies among objects are identified to consider how the objects are related to one another. Internal actions are not identified in Figure 3 since they does not provide any service to other objects and hence hidden from outside.

Table 2. Object of an Automatic Storage Retrieval System.

<table>
<thead>
<tr>
<th>Objects</th>
<th>Attributes</th>
<th>State Variables</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>working station</td>
<td>n: integer[1..6]; priority: integer</td>
<td>status: one of {wait, setting, processing};</td>
<td>s1, s2, s3, s4, s5, s6</td>
</tr>
<tr>
<td></td>
<td>with rule: priority = no;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>part box</td>
<td>n: integer; type: integer[1..6]; address: set of integer</td>
<td>status: one of {empty, full};</td>
<td>p_i = 1,...,n</td>
</tr>
<tr>
<td></td>
<td>with rule: address_rule;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>part box storage</td>
<td>queue_size: integer;</td>
<td></td>
<td>part_box_storage</td>
</tr>
<tr>
<td>automatic storage</td>
<td>cells: set of cell;</td>
<td></td>
<td>automatic_storage</td>
</tr>
<tr>
<td>cell</td>
<td>n: integer; capacity: integer;</td>
<td>contents: one of {empty, full} with rule: contents_rule1;</td>
<td>p_i = 1,...,n</td>
</tr>
<tr>
<td></td>
<td>#of_part: integer; if modified invoke</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>contents_rule1 with constraints:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#of_part &lt;= capacity;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>worker</td>
<td>worker_id: integer;</td>
<td>status: one of {busy, idle};</td>
<td>worker_1</td>
</tr>
<tr>
<td>conveyor</td>
<td>address: set of integer;</td>
<td>content: one of {empty, something, full} with rule:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>capacity: integer;</td>
<td>contents Rule2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#of_part: integer;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>conveyor 1</td>
<td>isa: conveyor;</td>
<td></td>
<td>conveyor_1</td>
</tr>
<tr>
<td>conveyor 2</td>
<td>isa: conveyor; direction: one of {forward,</td>
<td></td>
<td>conveyor_2</td>
</tr>
<tr>
<td></td>
<td>backward};</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty box storage</td>
<td>queue_size: integer;</td>
<td></td>
<td>empty_box_storage</td>
</tr>
<tr>
<td>transporter</td>
<td>transporter_no: integer;</td>
<td>status: one of {order, out_of_order};</td>
<td></td>
</tr>
<tr>
<td>transporter 1</td>
<td>isa: transporter;</td>
<td></td>
<td>transporter_1</td>
</tr>
<tr>
<td>transporter 2</td>
<td>isa: transporter;</td>
<td></td>
<td>transporter_2</td>
</tr>
<tr>
<td>supply line</td>
<td>queue_size: Integer;</td>
<td></td>
<td>supply_line</td>
</tr>
</tbody>
</table>
### Table 3. Actions of Objects in an Automatic Storage Retrieval System.

<table>
<thead>
<tr>
<th>Object</th>
<th>Actions (abbreviation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>working station</td>
<td>*(1) work (work)</td>
</tr>
<tr>
<td></td>
<td>*(2) unload empty box to return line (unload)</td>
</tr>
<tr>
<td></td>
<td>*(3) load filled part box to station (load)</td>
</tr>
<tr>
<td>automatic storage</td>
<td>**(4) determine cell to which part box is stored (determine)</td>
</tr>
<tr>
<td></td>
<td>**(5) choose cell from which part boxes are drawn (choose)</td>
</tr>
<tr>
<td></td>
<td>*(6) select empty boxes to be drawn (select)</td>
</tr>
<tr>
<td>worker</td>
<td>*(7) pick up part boxes from part box storage (pick up)</td>
</tr>
<tr>
<td></td>
<td>*(8) lay part boxes on conveyor 1 (lay down)</td>
</tr>
<tr>
<td></td>
<td>*(9) lay part boxes on conveyor 2 (lay down)</td>
</tr>
<tr>
<td></td>
<td>*(10) move empty boxes to empty box storage (move)</td>
</tr>
<tr>
<td>conveyor</td>
<td>*(11) move part boxes forward (move for)</td>
</tr>
<tr>
<td>conveyor 2</td>
<td>*(12) move part boxes backward (move back)</td>
</tr>
<tr>
<td></td>
<td>*(13) reverse direction (reverse)</td>
</tr>
<tr>
<td>transporter 1</td>
<td>*(14) store part box on automatic storage from conveyor 1 (store)</td>
</tr>
<tr>
<td></td>
<td>*(15) store part box on automatic storage from conveyor 2 (store)</td>
</tr>
<tr>
<td></td>
<td>*(16) deposit empty part boxes from cell 5 to conveyor 2 (deposit)</td>
</tr>
<tr>
<td>transporter 2</td>
<td>*(17) deposit part boxes on supply line (deposit)</td>
</tr>
<tr>
<td></td>
<td>*(18) store empty boxes on automatic storage from return line (store)</td>
</tr>
<tr>
<td></td>
<td>*(19) determine type of part box to supply line (determine)</td>
</tr>
<tr>
<td>supply line</td>
<td>*(20) move part box of type i to part box queue of station i (move)</td>
</tr>
<tr>
<td>part box queue</td>
<td>*(21) signal request for the part box to transporter 2 (signal)</td>
</tr>
</tbody>
</table>

*: internal actions  
**: response actions   
other: synchronous actions

#### 2.4 Establish External Interfaces and Implement Each Object

A suitable representation for the interfaces and for each object are selected and implemented in this step. The specification of external interface serves as a contract between the "clients" of an object and the object itself that forms the boundary between the outside view and the inside view of an object. In OPNets, interfaces are established by message queues and gates that relate message queues. Therefore the specifications of message queue and gates constitute the external interfaces.

Dynamic behavior of each object is also established in this step using high level Petri nets that identify the partial sequences of actions in each object and the influ-
ence of object states on its actions. Some of the critical objects to describe the system behavior are shown in Figure 4.

![Diagram of object interactions](image)

**Figure 3. External Visibility of Objects**

### 3. Analysis of the Inter—Object Behavior

Since analyzing a large complex net in a single step often produces erroneous results and is computationally inefficient, we have developed a two-step analysis method [Lee and Park 1993] which validates each object in a first step and then checks the synchronization constraints among the objects as a global analysis scheme. The procedure provides a way to manage the complexity by dividing the net into the smaller nets and then applies the analysis in two steps keeping the global validation intact. Briefly, the two-step validation procedure is as follows (detailed discussions are given in [Lee and Park 1993]).
In the first step, a local analysis is performed to validate the internal behavior of each object and to draw an interface equivalent net that shows only the firing sequence of the input and output gates, i.e., the synchronization constraints. In the second step, a synchronization analysis is performed to validate the interface equivalent net, which is constructed in the first step, to check the consistency of communications between objects. The interface equivalent net of an automatic storage retrieval system in which no deadlock detected is given in [Lee and Park 1993].

IV. CONCLUDING REMARKS

OPNets integrates the formalities and elegant expressions for concurrent control structures of the Petri nets, and the abstraction and powerful structuring schemes of the object—oriented approach. OPNets particularly focuses on the independent structure of objects and hence on the maintainability and reusability. With a view to improve the independence of objects, the communication knowledges are decoupled as much as possible from each
object and synchronization constraints are clearly separated from the internal control logic of each object. Validation of the whole system is much simpler when the partitioned nets are analyzed separately and then the communications are checked as a second step.

Contrary to the benefits, the structure of OPNets increases the number of places and transitions because message queues and gates should be added in order to separate the internal structure from the external structure. The maintainable and reusable structure can, however, outweigh the burden of simple increase in the number of places and transitions. A mechanism to inherit a behavior in addition to attributes and actions is being developed that will preserve the integrity of each objects and the synchronization constraints. An execution to the timed Petri nets that incorporates timing constraints into the OPNets is remained as a further study.

참 고 문 현


◇ 저자소개 ◇


공동저자 박성주는 서울대 산업공학공학사, 한국과학기술원 산업공학공학석사를 취득하고 미국 미시간주립대학교에서 시스템공학을 전공하여 공학박사를 취득하였다. 현재 한국과학기술원 테크노경영대학원 교수와 한국과학기술원 경영정보연구센터 소장으로 재직하고 있다. 주요 관심분야는 경영정보시스템, 정보시스템 통합, CSCW, 객제지향기술, 경영혁신기술, 시뮬레이션 등이다.