Large Scale Manufacturing Systems Modeling Tools Based on Object-oriented Petri Nets

Yang Kyu Lee(서원대 경영정보학과) and Sung Joo Park (한국과학기술원 테크노경영대학원)

Abstract

Manufacturing systems are usually large, complex, and concurrent in nature that makes it difficult to model and simulate the behavior in advance. The paper proposes an approach, called OPNets, for modeling and validating manufacturing systems. The approach is based on object-oriented high-level Petri nets in which modeling components of Petri net are constructed into object hierarchy. The objects communicate with each other by passing messages. To enhance the reusability and maintainability, a system are developed by object hierarchy. Inheritance among object hierarchy is also supported in OPNets. The modeling scheme of OPNets tries to resolve the complexity problems of Petri net.

I. INTRODUCTION

Petri nets have been found to be useful for describing and analyzing real-time systems such as manufacturing and robotics systems [1,2,3,4,5,7,15] 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 [10]. Therefore the complexity problem is one of the main obstacles in applying Petri nets to large complex manufacturing systems.

One promising solution to the complexity problem is object-oriented approach. An object-oriented high level Petri nets called OPNets is proposed 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 the system. Brief description of OPNets is give in here. More detailed explanations of OPNets are given in [9].

1. Systems

As ordinary object models, a system is modeled by objects and their relations. Objects in OPNets are hierarchically organized. The relationships between objects are modeled by a set of links called *interconnection relations*. 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 : a set of *objects*,

R : 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 O_2, 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 O_2 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 $O_i \in O$ is represented by the 6-tuple,

$O_i = (H_i, IG_i, OG_i, IM_i, OM_i, F_i)$,

where

H_i : *an object hierarchy,*

IG_i : *a set of input gates of object* O_i.

OG_i : *a set of output gates of object* O_i.

IM_i : *a set of input message queues of object* O_i.

OM_i : *a set of output message queues of object* O_i.

F_i : *a set of flow relations of object* O_i.

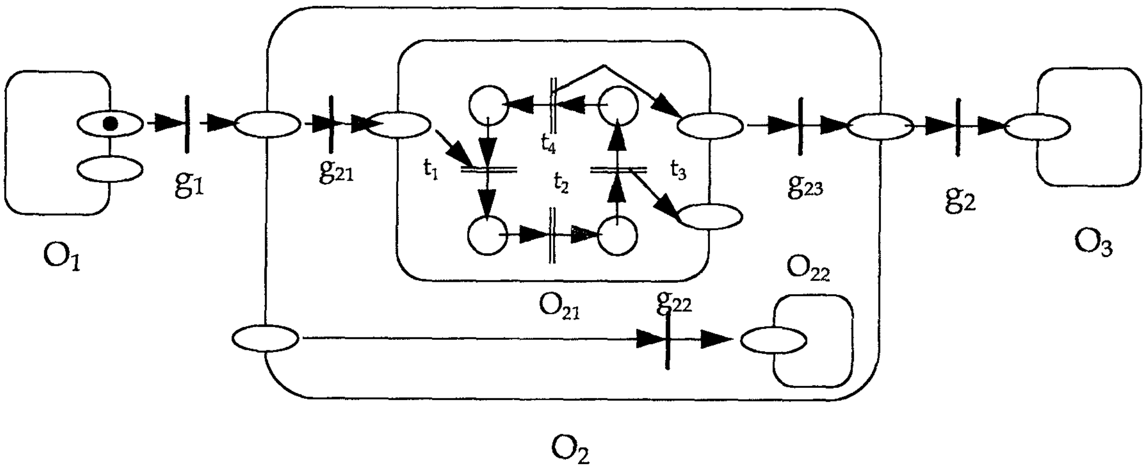

Figure 1. Graphical Representation of OPNet Structure

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 g_i 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.

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_2 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 $CO \cup PO$, where $CO = \cup_i CO_i$ and $PO = \cup_i PO_i$. The internal structure of a composite object CO_i, ICO_i, is characterized by the following 3-tuples.

$ICO_i = (X, Y, R_i)$,

where

$X = \{X \mid X \in \boldsymbol{P}(CO),\ CO_i \notin X\}$,

$Y = \{Y \mid Y \in \boldsymbol{P}(PO)\}$,

R_i : *a 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:

$IPO_i = (D_i, SV_i, S_i, AT_i, LF_i, IN_i, M_0)$

where

D_i : a set of attributes of PO_i,

SV_i : a set of state variables PO_i,

S_i : a set of states of PO_i,

AT_i : a set of action transition of PO_i,

LF_i : a set of local flow relations of PO_i,

IN_i : a set of instances of PO_i,

M_0 : 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. Table 1 shows the types, names, connected message queue types and control flows of actions in action transition. 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 transitions, 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_1 removes a message from the message queue of object O_1 and puts it into that of object O_2.

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_j's and OG_i's such that if $(O_i, O_j) \in R$, then $OG_i \cap IG_j \neq \varnothing$, where IG_j's are the input gates of object O_j and OG_j's are the output gates of object O_j. That is, if $g \in IG_j$ and $g \in OG_j$ then the O_i and O_j are connected through gate g, and g is connected to omq_i and imq_j, where $omq_i \in \bullet g$ and $imq_j \in g\bullet$. The $\bullet t$ ($\bullet p$) denotes the set of all input *places* (*transitions*) of a transition t (place p) and $t\bullet$ ($p\bullet$) 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:

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

In Figure 1, O_1 and O_2 are connected through g_1 and g_2.

Type	Name	Connected MQ	Control Flow
Internal	internal action	none	sequenced
External	asynchronous action	none	not sequenced
	synchronous action	Synchronization Queue	sequenced
	response action	Reply queue	sequenced

Table 1. Types of Action

III. INHERITANCE

Since large manufacturing systems are usually complex, parallel and distributed, they are difficult to specify, simulate, and anlyze. Therefore sharing previously developed specifications or code will greatly reduce efforts to build target systems.

1. Underlying Issues of Inheritance

Inheritance mechanism can be used as a code or specificaiton sharing mechanism among objects. When a class inherits from other class, it also should inherit behavior in addition to attributes and methods. Therefore some kind of behavior equivalence has to be guranteed across inheritance hierarchy. Otherwise, the inherited behavior will collide with the behavior of object which want to inherit attributes and methods from higher class. We call this situation as *interfere.*

The key points of inheritance in OPNets are as follows:

- ❑ Reduce required modeling efforts by sharing previously developed specifications
- ❑ Preserve the behavior of each object after inheriting the behavior
- ❑ Preserve inter-object behavior constraints
- ❑ Avoid interference of synchronization constraints with inheritance

2. Constraints of Behavior Inheritance

The following constraints should be considered in inheriting behavior among class hierarchy.

- ❑ The integrity of each object
- ❑ Synchronization constraints among objects
- ❑ Contractual obligations

The integrity constraint is the most fundamental one among constraints. The integrity of object is assured if the object's partial sequences of actions are preserved after inheritance. Synchronization constraints represent the sequence of message communications among objects that shows an external action selection sequence. They are represented by the firing sequences of gates. Contractual obligations define semantically meaningful constraints that restrict the sequences between the original actions and the inherited actions.

3. Temporal Logic for Defining Transition Firing Sequence

The firing sequence of transitions is defined by the temporal specification language that is revised by Uchihara from propositional temporal logic. The synchronization constraints among objects are restricted by constraining the firing sequences of gates. The contractual obligations are defined by firing sequence of the action transitions between object hierarchy. Therefore both of them can be defined by the temporal logic. The basic definitions of temporal specification language are as follows.

A set of formulas are defined from a finite set of elements E and a finite set of states S inductively as follows:

- ❑ If $e \in E$ and $s \in D \subset S$, then e(s,D) is a formula. D is called the domain of an element e.
- ❑ If f1 and f2 are formulas, then $\neg$ f1, f1$\wedge$f2, f1$\vee$f2, f1$\Rightarrow$f2, f1, □ f1 and ◇f1 and \$f2 are also formulas

The operators have the following meanings:

$\neg$ means NOT,

$\wedge$ means AND,

$\vee$ means OR,

$\Rightarrow$means IMPLY,

□ f means f is true for all future states

◇f means f is true for some future states

@f means f is true for the next state

f1 \$f2 means f1 is true until f2 becomes true

For example, if switch $\in$ E and D={on, off}, □ (switch(on, {on,off}) $\Rightarrow$@switch(off, {on, off})) is a formula. The formula means that when the state of switch is on, the next state will always be off.

In specifying the synchronization parts, the following abbreviations are introduced for readability:

- ❑ As domain is uniquely determined by each element, a domain can be omitted, ex. e(s,D) can be simply written as e(s)
- ❑ Fire(<gate name>) is abbreviated as e=s.
- ❑ If e$\in$E and s$\in$D, e(s) is abbreviated as e=s. ex, g1(full) can be written as g1=full

For example, a specification either *g1* or *g3* are fired in turns after firing of *g1* is expressed as follows:

□ (g1$\Rightarrow$@((g2$\Rightarrow$@g3) $\vee$(g3$\Rightarrow$@g2)))

4. Inheritance Procedures

The prcedure to inherit behavior, attributes, and methods consists of the following five steps:

- ❑ (Step 1) *Behavior Composition Procedure*: Compose behaviors of super-object and sub-object.
- ❑ (Step 2) *Filtering Procedure*: Eliminate the composed nets which do not meet the contractual obligations.
- ❑ (Step 3) *Snet Construction Procedure*: Construct synchronization nets from the synchronization

contraints (firing sequence of gates).

- ❑ (Step 4) *IE Net Construction Procedure*: Construct interface equivalent nets for the filtered nets of step 2.
- ❑ (Step 5) *Synthesis Procedure*: Synthesize the synchronized nets from the synchronization of step 3 and the interface equivalent nets of step 4.

The first step intends to merge the all possible combinations of the behavior of super class and sub class while maintaing the integrity of each object. The second stpe eliminates the composed nets which do not meet the contractual obligations. In the third step, the synchronization constraints are defined by a form of temporal logic. The synchronization constraints defined in a form of temproal logic are transformed into a form of Petri nets as follows:

- ❑ (Step 3.1) Decomposition Procedure: The formulas are decomposed into current formulas and future formulas. Current formulas do not include temporal operators. Future formulas are also decomposed into current and future formulas from the next time point view. After every type of future formulas has been repeatedly decomposed, a graph is derived, where each set of the elements in current formulas form a transition. This graph is an incomplete model satisfying all specifications other than the eventuality formulas that are formulas of the form ◇f, ┐ □ f or ┐ (┐ f1 $f2).
- ❑ (Step 3.2) Elimination Procedure: Edges with an unsatisfiable eventuality formulas are deleted from the graph. The graph remaining after the elimination procedure is a complete model of the initial specification.

The forth step derives the interface equivalent nets from the filtered nets in the same way as the local analysis of the behavioral analysis [9]. In the final step, the synchronization nets are then synthesized

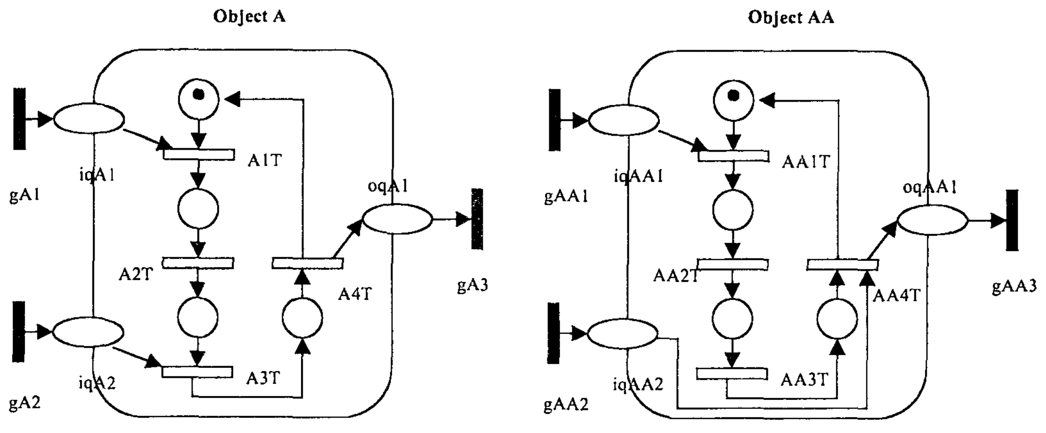

Figure 2. An Object Hierarchy.

from the synchronization constraint net and the interface euqivalent net. After these procedures has

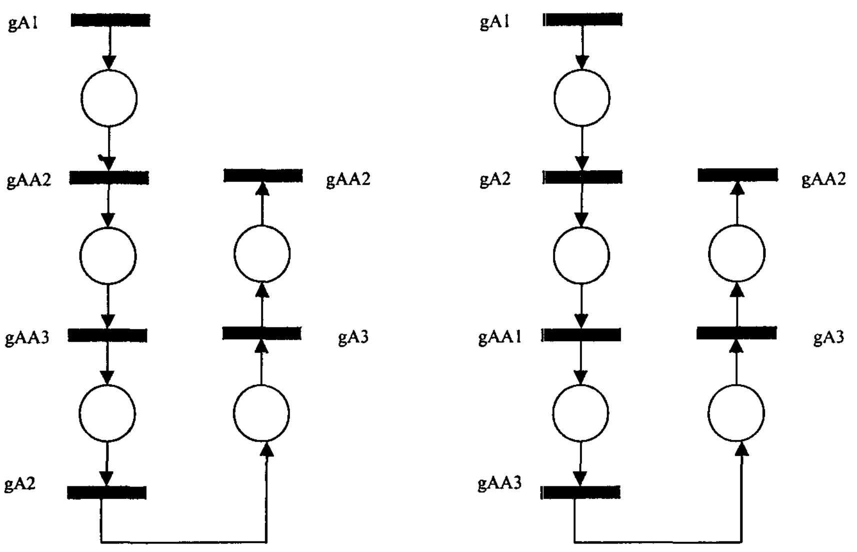

Figure 3. IE Net of (a) First and (b) Second Combination

finished, synthesized nets meets the synchronization constraints, the integrity of objects and contractual obligations.

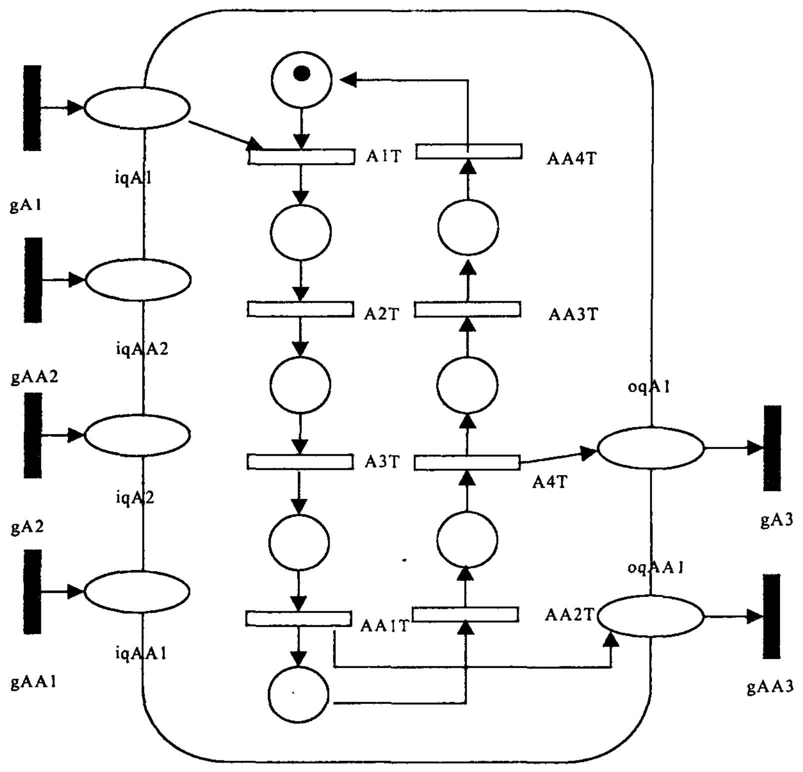

Figure 4. A Synthesized Net After Inheritance

5. An Example to Illustrate Inheritance Procedure

An object A is a super class of object AA and their behaviors are shown in Figure 2. The synchronization constraints among object A and object AA are defined as follows:

$\Box\ ((gA2 \Rightarrow @(gAA1)) \wedge (gAA2\$gA3))$.

The formula states that the firing of gA2 forces gAA1 to be fired in the next state and gAA2 can be fired after firing gA3.

The contractual obligation can be defined as follows.

$\Box\ ((AA1T\$A2T) \wedge (AA2T\$A3T) \wedge (A4T \Rightarrow @(AA3T)))$.

The formula states that an action transition AA1T can be fired after A2T has fired, the firing of A3T enables AA2T to be fired in the later states, and the firing of A4T forces AA3T to be fired in the next state.

In the first step, all possible combinations of action firing sequence between object A and objcet AA are generated. The number of possible combination is 35. After eliminating unmeaningful sequences, we can obtain only two combinations as follows:

1: A1T→ A2T→ AA1TA3T→ AA2T→ A4T→ AA3T→ AA4T

2: A1T→ A2T→ A3T→ AA1T→ AA2T→ A4T→ AA3T→ AA4T

The interface equivalent net of the first combination is obtained as shown in Figure 3. Among the two combinations, the first one does not meet the synchronization constraint that the gA2 should be fired before enabling gAA1. Therefore the resulting net that meets all the constraints is the net derived from the second combination as shown in Figure 4.

IV. 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 [9] 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 [9])

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 [9].

V. 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.

REFERENCES

[1] Baldassari, M and Bruno, G, "PROTOB: Object-oriented Graphical Modeling and Prototyping of Real-Time Systems," *Second International Workshop on Computer-Aided Software Engineering*, July 1988, pp. 28/6-28/10.

[2] Bruno, G and Marchetto, G, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, Feb. 1986, pp. 346-357.

[3] Camurri, A and Franchi, P, "An Approach to the Design and Implementation of the Hierarchical Control System of FMS, Combining Structured Knowledge Representation Formalisms and High-Level Petri Nets," *Proc. of IEEE Int'l Conf. on Robotics and Automation*, 1990, pp. 520-525.

[4] Freeman, P and Malowany, A, "SAGE: A Decision Support System for the Sequencing of Operations within a Robotic Workcell," *Decision Support Systems* 4, 1988, pp.329-343.

[[5] Garnousset, H, Farines, J, Cury, J, Cantu, E and Kaestiner, C, "Simulation and Implementation Tools for Manufacturing Systems Modelled by Petri Nets with Objects," *International Conf. CIM 90*, June 1990, pp. 605-613.

[6] Genrich, H and Lautenbach, K, "System modeling with high level Petri nets," Theoretic Comput. Sci., vol. 13, 1981, pp. 109-136.

[7] Kodate, H, Fujii, K and Yamanoi, K, "Representation of FMS with Petri Net Graph and its Application to Simulation of System Operation," *Robotics and Computer Aided Manufacturing*, vol.

3, no. 3, 1987.

[8] Lee, K.H. and Favrel, J., "Hierarchical Reduction Method for Analysis and Decomposition of Petri Nets," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-15, No. 2, March 1985, pp.272-280.

[9] Lee, Y.K. and Park, S.J., "OPNets: An Object-Oriented High Level Petri Net Model for Real-Time System Modeling," *The Journal of Systems and Software*, vol. 20, no. 1, January 1993, pp.69-86.

[10] Murata, T.,"Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, April 1989, pp. 541-580.

[11] Paolo, B and Gini, M,"An Object-oriented approach to robot programming," *Computer-Integrated Manufacturing Systems*, vol. 2, no. 1, February 1989, pp. 29-34.

[12] Peterson, J., *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, 1981.

[13] Sibertin-Blanc, C., "High Level Petri Nets with Data Structure," *6th European Workshop on Petri Nets and Applications*, Espoo, Finland, July 1985.

[14] Sibertin-Blanc, C. and Bastide, R., "Object Oriented Structuration for High Level Petri Nets," *11th Conference of Application and Theory of Petri Nets*, 1990.

[15] Son, S.K., Kim, Y.H. and Lee, K.H., "Modeling on a Simple Automatic Storage/Retrieval System by Grafcet Model," *Proc. of Korea OR/MS Conference*, 1989, pp. 143-150.

[16] Tyszberowics, S. and Yehudai A., "OBSERV - A Prototyping Language and Environment combining Object Oriented Approach, State Machines and Logic Programming," *HICSS*, 1990, pp. 247-256.

[17] Wilson, R.G. and Krogh, B.H., "Petri Net Tools for the Specification and Analysis of Discrete Controllers," *IEEE Transactions on Software Engineering*, vol. 16, no. 1, January 1990, pp. 39-50.

[18] Yau, S.S. and Caglayan, M.U., "Distributed Software System Design Representation Using Modified Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 6, November 1983, pp. 733-745.