High-level Petri net for incremental analysis of object-oriented system requirements

J.-E. Hong and D.-H. Bae

Abstract: To complement the weakness of Petri nets in terms of naturalness, modularity, and reusability, high-level Petri nets with object concepts have been suggested. It is difficult to apply these nets to the requirements specification of object-oriented software systems because of insufficient support for the object-oriented concepts. A hierarchical object-oriented Petri net (HOONet) is developed to complement the weakness of the existing formalisms and formally define its syntax and semantics. A reachability analysis method is provided to check such behavioural properties as boundedness, liveness and persistence of the HOONet models. The HOONet provides incremental modelling and analysis of the requirements with the support of object-oriented concepts.

1 Introduction

When specifying, modelling, and analysing the requirements of a complex system, we often choose a modelling language which can formally depict the properties of the system. One popular modelling language is the Petri net which represents system behaviours based on state-based transitions. Petri nets are widely used in various application domains because of simplicity, flexibility to depict the dynamic behaviours and strong expressive and analytic power of the system behaviours [1]. Despite the powerful modelling mechanisms of Petri nets, software developers are still likely to recognise some weakness in Petri net formalisms. Thus, many Petri net fanatics have been devoted to enhancing and extending the theories and techniques of Petri net. In addition, as object-oriented technology is widely accepted in industry for developing applications of various domains, research has attempted to merge the concepts of object and the Petri net formalism. A hierarchical object-oriented Petri net (HOONet) is developed to complement the weakness of the existing formalisms and formally define its syntax and semantics. A reachability analysis method is provided to check such behavioural properties as boundedness, liveness and persistence of the HOONet models. The HOONet provides incremental modelling and analysis of the requirements with the support of object-oriented concepts.

2 Hierarchical object-oriented Petri net

HOONet is a high-level Petri net supporting the representation scheme of object-oriented concepts. A HOONet model is represented as Petri-net form for an object, and has components to represent a unique name, attributes, and its behaviours (methods) of an object. The formal definition of HOONet is given as follows:

Definition 1: HOONet is defined with a tuple HOONet = (OIP, ION, DD), where

(i) OIP (object identification place) is a special place which is defined as a tuple, OIP = (oip, pid, MO, status), where

oip is a unique name of a HOONet model,

pid is a unique process identifier that distinguishes the multiple instances of an object,

MO is an initial marking function,

status is a flag variable (either "pre" value or "post" value) to represent the specific states of OIP

(ii) ION (internal object net) is a variant of CPN (coloured Petri nets) that represents the internal behaviours of an object, which is defined as a tuple ION = (\( \mathcal{P}, T, A, C, N, G, E, F, M_0 \)), where

\( \mathcal{P}, T, \) and \( A \) are finite sets of places, transitions, and arcs, respectively

\( C, N, G, \) and \( E \) mean the functions of a colour set, a node, a guard, and an arc expression, respectively. They are the same as defined in [12]

\( F \) is a special arc from transitions to OIP, and depicted as a rim of ION, and

\( M_0 \) is a function giving initial marking to specific places.
(iii) DD (data dictionary) contains the declarations of variables, token types, and functions per a HOONet model using Standard ML [13].

When modelling and analysing object-oriented system requirements, the first issue to be considered is abstraction. Abstraction focuses on the essential, inherent aspects of an entity and ignores its accidental properties. In requirements modelling, this means focusing on what an object is and does, before deciding how it could be refined into details. The use of abstraction gives a freedom to make decisions as late as possible by avoiding premature commitments to detail [14, 11]. In HOONet models, abstraction can be represented as abstract places, abstract transitions, and abstract tokens, and these components can be refined into subnets in further modelling.

The second issue is static reaction. Although an abstraction is not yet made concrete, the intended behaviour must occur without suspension to simulate the models. To do this, it is necessary to get an artificial effect, as if the actual execution of the abstraction had occurred. This can be accomplished by the prior definition of postconditions for such behaviours, or by the interactions of analysts. This is termed static reaction. Also of concern is depicting the interaction between the subsystems. Such interaction can be explained with method calls and global variable accesses [11]. In HOONet models, this can be represented with communicative transition and message (token) passing. Interaction through global variables can be achieved with the declarations of global type variables in the data dictionary.

Definition 2: A set of place types in HOONet, $\mathcal{P} = \{P, P_a\}$, where

(i) primitive place $P_i$ is a basic type of places that represent the local states of a system, the same as in general CPN [12]
(ii) abstract place $P_a = (pn, \text{refine\_state}, \text{action})$ is a place type which represents abstract states, where

$pn$ is the name of an abstract place

$\text{refine\_state}$ is a flag variable denoting the refinement of an abstract place

$\text{action}$ is a static reaction that imitates the internal behaviours of an abstract place.

If the value of $\text{refine\_state}$ is “true”, the abstract place is refined into subnets, and “false” value means the abstract place is not yet refined.

Definition 3: A set of transition types in HOONet, $\mathcal{T} = \{T_p, T_a, T_e\}$, where

(i) primitive transition $T_p$ is a basic transition type in general CPN
(ii) abstract transition $T_a = (tn, \text{refine\_state}, \text{action})$, where

$tn$ is the name of an abstract transition

$\text{refine\_state}$ and $\text{action}$ have the same meanings as in the definition of the abstract place

(iii) communicative transition $T_e = (tn, \text{target}, \text{ctype}, \text{action})$ is a transition type that represents calling a method, where

$tn$ is the name of a communicative transition

$\text{target}$ is a flag variable denoting whether the method called from $T_e$ was modelled (a “yes” value) or not (a “no” value)

$\text{ctype}$ is also a flag variable denoting whether the interaction of $T_e$ is synchronous (a “SYNC” value) or asynchronous (a “ASYN” value)

$\text{action}$ is the static reaction that reflects the execution results of the called method.

The variable $\text{ctype}$ with its “SYNC” value denotes that the caller waits for the result from the called method. With “ASYN” value the token is duplicated. Each of the duplicated tokens is transferred to the called object and the next place in its net, respectively.

In HOONet, the token types are described in the data dictionary, and classified into two categories: primitive type and abstract type. The primitive type is the same as that in CPN [12], while the abstract type is a compound set of primitive types. The abstract type is required to express the representative states of detailed substates or the unrefined states of abstract places. When a net is modelled with high-level information, the token types representing the states in the net should also be represented with high-level information, since the token type expressed with detailed information does not adequately represent the states of abstract behaviours in the net. When the designer declares the token type in detail at the abstract level, it is still acceptable. However, such representation for abstract states is not concise, and can cause changes in further refinements. The type of the abstract token is declared with the prefix “complex”. This type is decomposed into several subtypes (primitive types) of tokens in refined nets. The definition of the data dictionary is very straightforward, and is written in a textual grammar such as CPN ML [13]. The graphical structure of the HOONet is shown in Fig. 1. With a marking at OIP and $\text{OIPstatus} =$ “pre”, a firing transition from the OIP (generally, $x$ is the input to $x$, $\text{output}$ is the output of $x$; formally, $\text{output} = \{y \in \mathcal{P} \cup \mathcal{T} | (y, x) \in A\}$ and $\text{output} = \{y \in \mathcal{P} \cup \mathcal{T} | (x, y) \in A\}$ where $A$ is a set of arcs) is selected depending on the token values of the marking. As the ending process of each method, the token arrives at the OIP through the body frame of ION, and the value of $\text{OIPstatus}$ becomes “post”. If the value of $\text{pid}$ of the OIP is “self”, the action is terminated, otherwise the token is returned to the calling point.

3 Specifying requirements using HOONet

This Section explains the object-oriented representation for specifying system requirements using HOONet and formally defines the behavioural semantics of HOONet models. For an explanation of our approach, an example system, shutdown trip system, which is the preferred system for quickly terminating reactor operation when certain parameters exceed specified limits, is introduced with the following brief requirements [15]:

The shutdown trip system (STS) of the reactor safety system controls the operations of the turbine, emergency core cooling (ECC) system, and heat transport pump
(HTP) depending on the given input values, temperature and pressure. Within the normal range values of temperature and pressure, the STS has no action. When out of range, the operations of Turbine, ECC, and HTP change to turn-off, turn-on and high-frequency, respectively.

3.1 Modularisation and encapsulation of real-world entities

One goal is to treat objects as black-boxes, whose external interface is public but whose internal details are hidden. Hiding internal details allows the design (also including implementation) of an object to be changed without requiring modification to its clients [11].

In our modelling approach, system requirements are decomposed into objects based on actor-based behaviour. These objects are represented with a set of HOONet models. A HOONet model modularises the behaviour of an actor. OIP of HOONet plays the role of the public interface, and encapsulates the internal behaviours of an object.

Fig. 2 shows the HOONet model for the example system STS. This Trip model represents a tripping system according to the input values of pressure and temperature. The abstract transition down represents the abstraction of the behaviours of ECC, HTP, and Turbine when out of the normal range of temperature and pressure. As shown in Fig. 2, the initial state of the Trip system can be changed to one of three states, normal, recovered, or abnormal, depending on their guard conditions. InRange and OutRange are boolean variables representing whether the degree of temperature and pressure are within the specified range or not, and the variable flag represents the occurrence of down behaviour.

3.2 Abstraction and refinement

The abstraction mechanism allows for specifying incrementally system requirements even if the requirements are partially omitted. The algebraic expression "action" defined in an abstract component makes it possible to simulate the HOONet models containing such omissions. The high-level requirements of a system are usually described with some abstract information. Along with further specification of requirements, abstractions are refined into detailed information, which is represented with refined HOONet models. Fig. 3 shows the refinement of the abstract transition down in the Trip object. When the range of temperature or pressure goes out of the upper limit, the operations of the cooling system, heat transport pump, and turbine are either turned on or off according to the conditions. These operations are performed by interaction with each other objects as ECC, HTP and TURB.

The properties, which should be preserved between the abstract model and its refined model, can be defined by the morphism between each model [14]. In our HOONet models, the sine qua non condition of morphism is sufficient to show that the observable behaviours are equivalent between the abstract component and its refined model. The observable behaviours are decided on within such constraints as token type, arc expression, and guard expression. Thus the requirements of an observable morphism in HOONet models are given as follows:

Definition 4: The observable morphism between abstractions and their refinements should satisfy the following requirements:

(i) \( \forall p \in P \omega \ C(p) \subseteq T_{RN} \), and \( \forall t \in T_o : (C(t) \cup C(\circ)) \subseteq T_{RN} \).

The multiset \((C(p))\) of token types of every abstract place should be a subset of the set \((T_{RN})\) of all token types defined in the DD of the refined net (RN). And the multiset of token types for incoming places and outgoing places of an abstract transition should also be a subset of \(T_{RN}\).

(ii) \( \bigcup_{a,a'} Var(Var) \subseteq Var(OIP_{RN}) \subseteq V_{RN} \).

The set \((Var(Var))\) of all variables appearing in the incoming and outgoing arc expressions of an abstract transition should be a subset of the set \((V_{RN})\) of all variables defined in DD of RN.

(iii) \( \forall t \in T_d, G(t) \equiv \bigcup_{x \in OIP} G(t) \).

The evaluation of guard expression of an abstract transition should be equivalent to the evaluation of all guard expressions of the outgoing transitions of OIP_{RN}.

The behavioural semantics of an abstract component are defined with the flag variable refine.state. If the value of the refine.state is "true," the OIP of the refined net is marked, otherwise (a "false" value) the static reaction of the abstract component is executed and the binding of a transition in \(\forall t \in P_o\) occurs for an abstract place or \(+M(T_o) \land -M(T_o)\) is performed for an abstract transi-
3.3 Inheritance: sharing properties

Inheritance is the sharing of attributes and operations among classes based on hierarchical relationships. A class can be defined generally and then can be refined into successively finer subclasses. Each subclass incorporates, or inherits, all of the properties of its superclass and adds its own unique properties. The properties of the superclass need not be repeatedly defined in each subclass [11].

The inheritance notation in the HOONet is shown in Fig. 5. This Figure represents the subclass Down2 inherited from the class Down in Fig. 3. The symbol expressing inheritance is "<" and is declared within the name of OIP. The inherited subclass Down2 includes the additional behaviour of alarm signal issuing, denoted with the place alarm_on.

When the inherited subclass is instantiated the DD and OIP of the subclass should be extended to include those of the superclass. To override the properties of the superclass with modification, which method is supposed to be bound between the overridden method and the overriding method, should be determined because overriding redefines the binding. This rule searches for an inheritance hierarchy to bind a method that implements a specific operation in the object of the subclass. These notions of inheritance mechanism in HOONet models are formally defined as follows:

Definition 5 Inheritance of properties from the superclass satisfies the following when the subclass is instantiated (let I, S, and N be an inherited subclass, a superclass and the newly defined and overriding methods, respectively):

(i) \( DD_I = DD_C \cup DD_N \)

(ii) a set of enabled transitions of the subclass \( ET_I = \{ (t, E(OIP, t)) \mid (t, E(OIP, t)) \} \cup \{ (t, E(OIP, t)) \} \)

(iii) for \( (t, E(OIP, t)) \) and \( (t, E(OIP, t)) \) where \( (t, E(OIP, t)) \) is the enabled transition from \( p \) to \( i \) with the binding \( b \).

3.4 Supporting dynamic binding

Dynamic binding can be regarded as a dispatching mechanism that acts like a case statement to dynamically select the appropriate behaviours in response to the value of a token. In HOONet models, dynamic binding for parametric polymorphism can occur when the variable refine_state of \( P_a \) or \( P_o \) is true. Dynamic binding is processed with two steps. First, the complex token type of the abstract component is mapped onto the detailed token type of the refined model, and then the token values of the OIP in the refined model are defined with a specific value depending on the value of the complex typed token. For instance, in Fig. 2, the token type TC represents the trip conditions of the shutdown system (i.e. \( TT TC = complex with weak | strong \)). When the token value of TC is weak, the token type TC is decomposed into

\[
TT TC = record \{ Temp = OutRange \midInRange; \nonumber \\
Pres = OutRange \midInRange; \nonumber 
\}
\]

and the values of each token are assigned with \((\{Temp, OutRange\}, \{Pres, InRange\})\) in the refined net. However, in the case that the weak value indicates the condition of pressure, the token values are assigned with \((\{Temp, InRange\}, \{Pres, OutRange\})\) in the refined net. An abstract token value can be bound with the different values in the refined net according to the value of the parametric variable.

3.5 Interaction between objects

Message passing is a mechanism allowing interaction between objects. The message passing in HOONet occurs by calling the method of an object, and passing a token from \( T_c \) of an object to OIP of another object. There are two types of method calls to be considered.

3.5.1 Synchronous method call: \( T_c \) must receive the result from the execution of a called method in the synchronous method call. So the places of \( T_c \) are not marked and wait until the result is returned. This waiting occurs when the value of target is "yes". If the value of target is "no," the static reaction of \( T_c \) is executed and the output places of \( T_c \) are marked. This is formally defined as

\[
M(OIP) \text{ for } T_c.in = OIP.oip; -M(sT_c) \text{ if target = "yes"} \nonumber \\
E(action); +M(t_c) \land -M(sT_c) \text{ if target = "no"} \nonumber 
\]

\( IEEE \ Proc.-Softw., \ Vol. 148, \ No. 1, \ February 2001 \)
where $E(\text{action})$ means the evaluation of the static reaction of a communicative transition.

3.5.2 Asynchronous method call: Even though the token is transferred from $T_r$ to OIP, the places of $T_r$ should have markings in the asynchronous method call. In $T_r$, defined with the asynchronous type, the token is passed to OIP and $T_r$ both. The result of the execution of a called method is stored in the global variables for later use.

$$
| M(OIP) |_{T_r.m} = OIP.oip \cdot M(T_r) + M(T_r) \cdot M(OIP)_{T_r} |
$$

3.6 Instantiation control

The identification of multiple instances of a class is achieved by the variable $\text{pid}$ of OIP. This variable distinguishes each instance by the dynamic assignment of a value as a unique identifier.

The $\text{pid}$ value is proportionally changed to identify each instance along with the instantiation chain. If an instance $B$ is generated by an activity of an instance $A$, the $\text{pid}$ value of the instance $B$ is assigned with the result of concatenation of the $\text{pid}$ value of the instance $A$ and the return address of the instance $B$ (i.e. $\text{pid} = \text{pid}(\text{return})$). If the $\text{pid}$ value is self, this instance is a root of the instantiation chain and an initiator of such behaviour as $\text{main()}$ function in procedural languages. The $\text{pid}$ value is increased whenever the instantiation chain becomes deeper, and is decreased whenever each instance is destroyed by the ending of their behaviour.

4 Incremental reachability analysis

4.1 Overall approach

Conventional reachability analysis methods essentially involve the enumeration of all reachable markings to check such behavioural properties as boundedness, liveness, and nondeterminism of Petri-net models. This enumeration appears as a reachability graph (or tree). However, since conventional reachability analysis methods cause state-space explosion [12], some compositional reachability analysis methods have been suggested [16–18]. Our incremental approach has a major difference when compared with other compositional approaches. This difference owes to taking advantage of the abstraction and refinement mechanisms of the HOONet models. Thus, even if a system is not completely modelled, reachability analysis is possible [9]. Additionally, our approach satisfies the one-entry one-exit constraint [19] required to perform the compositional reachability analysis, since the interface of each HOONet is limited to only OIP. Thus, a global reachability graph can be generated by merging separately generated graphs [9]. Fig. 6 shows the logical concept of our incremental analysis approach.

The hierarchical structure of the HOONet models shown in the upper side of Fig. 6 represents the abstraction hierarchy of the models. As may be seen the modelling of shutdown trip system in Section 3, we first modelled the Trip object of hierarchy-level 1, then the Down and Down2 objects are sequentially modelled. Such sequential modelling allows for incrementally modelling objects of each level from high levels toward low levels, even though we do not know the details of Down2, ECC, and so on. The postconditions written in the “action” fields of the abstract and communicative components and the interface limited to OIP of HOONet make it possible to model a system with such an approach. For instance, when we defined the postconditions of the communicative transitions ECC, HTP, and TURB in Fig. 3 with EccOn := "$\text{true}$", HtpHigh := "$\text{true}$", and TurOff := "$\text{true}$", respectively, we could simulate only the behaviours of trip, Down, and Down2 models which are interesting parts of whole system models, and also could generate their reachability graphs without the refined models of ECC, HTP, and TURB objects. After the modelling of one or more among the remaining objects, these objects can be integrated into global models (or a global graph).

4.2 Reachability graph generation

Algorithm 1 generates the reachability graph (RG) of the HOONet models. In this algorithm, the routine GenNode ($M_1, M_2$) generates an arc between the source node $M_1$ and the destination node $M_2$; and then tags $M_1$ with a reachable node (R), and $M_2$ with a node (M) whose successors are not yet found. The routine AllGen ($M_1, i = 2..n$) generates arcs from the source node $M_i$ to all the successors ($M_j$) of $M_i$; and tags $M_i$ with "$R_i$", and all $M_j$ with "$M_j$". The pseudocodes of these routines are as follows, and their conceptual mechanisms are shown in Fig. 7.

GenNode ($M_1, M_2$);
begin
create a node labelled with $M_2$;
create a new arc from $M_1$ to $M_2$;
end.

Fig. 6 Logical concept of incremental analysis

\begin{itemize}
  \item a Reachability graph for trip, down and down2
  \item b Reachability graph for the refinement of ECC
  \item c Integrated reachability graph
\end{itemize}

Fig. 7 Routines: GenNode and AllGen

\begin{itemize}
  \item a GenNode($M_1, M_2$)
  \item b AllGen($M_1, M_2$)
\end{itemize}
where \( S_R \) and \( S_T \) mean the sets of all reachable nodes and the nodes whose successors are not yet found, respectively. 

\[
\text{AllGen} \ (M_1, M_i);
\]

begin 
for \( (M_i \text{ in all possible next moves form } M_1) \) do

\[
\text{GenNode} \ (M_1, M_i);
\]
end.

Algorithm 1: GenRG (input: Nets, output: RG);

1. Let the node which has initial marking, be root of a RG; and tags this node with \( M_1 \).
2. While \( \{\text{all nodes tagged with } M_i \} \neq \emptyset \) do begin switch (a node tagged with \( M_i \))

\[
\text{case OIP: (status = "pre") then AllGen} \ (M_1, M_i);
\]
else if (\( \text{pid = "self"} \) ) then Exit();
else GenNode \( (M_1, M_i) \) for node to return;

\[
\text{case } P_i : \text{ AllGen} \ (M_1, M_i);
\]
\[
\text{case } P_o \text{ or } T_c : \text{ if (refine.state = "false") then AllGen} \ (M_1, M_i);
\]
else GenNode \( (M_1, M_i) \) for OIP of refined net;

\[
\text{case } T_c : \text{ if (target = "no") then AllGen} \ (M_1, M_i);
\]
else if (\( \text{ctype = "SYNC"} \) ) then GenNode \( (M_1, M_i) \) for OIP of called net;
else \( \text{AllGen} \ (M_1, M_i) \| \text{GenNode} \ (M_1, M_i) \) for OIP of called net;
end;
3. End-while;
4. End of Algorithm.

The steps to merge two reachability graphs are written in algorithm 2. This algorithm incrementally generates a global reachability graph by merging reachability graphs generated using algorithm 1. Let SetABC be a set of all \( P_o \)s, \( T_o \)s, and \( T_c \)s, and \( M_i \) be the preceding state of an element \( n_i \) in a given reachability graph.

Algorithm 2: MergeRG (input: RGs, output: GRG);

1. Select a component \( n_i \in \text{SetABC} \)
2. Using algorithm 1, generate the reachability graph corresponding to the refined model of the \( n_i \)
3. Inserts the reachability graph at the next location of the \( M_i \)

Our incremental analysis method is very useful in requirements specification of large scale object-oriented systems because it is possible to generate the reachability graph for only the parts of interest.

4.3 Analysis of system properties

When an abstract component delineating within a HOONet model is refined the RG of the refined net is generated by the algorithm. Then this RG can be merged into a global RG. Using the reachability graph we can analyse some behavioural properties such as deadlock freeness, liveness, and so on [1, 12].

Deadlock check: In the reachability graph, deadlock can be checked by dead marking which has no enabled transitions. The firing sequence from the initial marking to the dead marking represents the deadlock status of the system.

Liveness check: Since every transition should be fired at least once, every transition in the HOONet models should appear in their reachability graph. Transitions that do not appear in the graph cannot be fired. A system model should not have these dead transitions.

Fairness check: Two transitions are said to be in fair relation if the maximum number of times that either one can fire while the other is not firing, is bounded. This means that the system model does not have an infinite loop of behaviour. Thus all generated reachability graphs are fair.

Nondeterminism check: Nondeterminism between transitions can be checked using the reachability graph and the structural analysis. When transitions in conflict, where they have the same input place, are enabled at a reachable marking, a nondeterministic choice occurs. Nondeterminism is checked by searching two enabled transitions in conflict for all reachable markings in the reachability graph, and the final decision for such flaw can be made by the domain experts.

Additionally, to show the efficiency and the performance improvement of our incremental analysis technique, we compare the number of states and state transitions with existing Petri net formalisms. We first calculate the number of states and state transitions of the HOONet models for STS as shown in Table 1.

To compare our results, as shown in Table 1, with existing formalisms, P/T nets, modular Petri nets, and coloured Petri nets are selected [12, 20]. And Trip and Down object models are selected as target models in the comparison. Table 2 shows the comparison results.

From Table 2 the number of states for HOONet models is reduced to about 5% for P/T net models, about 50% for modular Petri net models, and about 76% for CPN models. Such reduction comes from that the additional states
occurring in the subnet compositions of HOONet, are few compared with those of modular Petri nets and CPN. However, these statistics do not indicate any significant merits in our HOONet formalism. The focus of our proposal is that HOONet formalism supports the incremental modelling and analysis for missed or not-yet analysed requirements.

Fig. 8 shows a screen of our prototype system developed for supporting HOONet modeling and analysis. It was developed using Visual Cafe® and Java, and some of the model verification components are still under development.

5 Related work

Most OPN formalisms support the notion of modularisation and encapsulation but do not sufficiently provide the notions of abstraction and refinement. Abstraction mechanisms must be able to abstract both the state and state transition of a system to satisfy the duality property of Petri nets [14]. However, most OPNs do not provide such abstraction mechanisms. Some OPNs such as OBJSA [2], PN-TOX [5], and OPNets [7] support the notion of inheritance, but do not support the overriding caused by the modification of inherited properties.

In the aspect of interaction between system components, most OPNs support interaction with peculiar communication mechanisms. OPNets support interaction using the features of "INPORT" and "OUTPORT." COOPN/2 [3] has interface transitions and communicative operators to co-ordinate the interaction and G-Nets [8] support the interaction with "goal place" and "instantiated switching place". Another concern is the mechanism for controlling multiple instances of objects at the run-time environment of the object system. Only G-Nets suggest this control mechanism using the unique process identifier of each object instance. The notion of polymorphism was considered in Lakos's research [6], but the research did not consolidate this notion with the features of LOOPN++.

Although most high-level Petri nets try to model the object-oriented concepts in their representation scheme, they do not support the incremental analysis approach which is supported from HOONet. We believe that this incremental analysis approach can be achieved when the modelling technique can support the seamless abstraction and refinement mechanism which handles data and control together. However, although most OPNs such as COOPN/2 [3], LOOPN++ [6], OPNets [7] and G-Nets [8] can incrementally specify the system behaviour by modularised concept of object, they do not provide the capability for incremental approach which can analyse system properties for the only interested parts or the incompletely described system models.

6 Conclusion and future work

We have described the finding of deficiencies or extensible issues in existing OPNs formalisms. Although a number of high-level Petri nets with object concepts were suggested with clear idea of specific concerns, they did not support a sufficient number of features needed to design a system with full object-oriented concepts. To overcome this problem we suggested hierarchical object-oriented Petri net formalism with clear concepts and their semantics. Furthermore, we described the specifying and analysing methods for the system requirements and proposed methods to make it possible to develop object-oriented systems incrementally and iteratively. When we specify and analyse system requirements, HOONet supports the hierarchical decomposition and stepwise refinement of the requirements. This can handle the complexity of the requirements with the provision of abstraction and refinement mechanisms, and also can check the correctness of the requirements and the models by an incremental reachability analysis method.
Our ongoing studies focus on two issues: one is to apply HOONet formalism to other domains, the other is to extend HOONet formalism. As applicable domains of HOONet, we are trying to model scenario-based requirements using HOONet. One issue of scenario-based software modelling is the handling of the different abstraction levels between scenarios. The abstraction mechanism of HOONet will be able to solve this issue. Architecture description and component reuse in component-based software development are other applicable domains of HOONet. As an extension of HOONet formalism, we will consider the timing constraints which are needed to model real-time systems.

7 References