Software modeling and analysis using a hierarchical object-oriented Petri net

Jang-Eui Hong *, Doo-Hwan Bae

Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Kusong-dong, Yusong-gu, Taejon 305-701, South Korea

Received 9 February 1999; received in revised form 6 June 2000; accepted 8 October 2000

Abstract

Petri net is used widely to analyze and model various systems formally. Recently, many Petri nets mania devote their efforts to enhancing and extending the expressive power of Petri nets. One such effort is to extend Petri nets with object-oriented concepts. An object-oriented paradigm provides excellent concepts to model real-world problems. Object-oriented concepts allow us to build software systems easily, intuitively, and naturally. Although several high-level Petri nets with the concept of objects are suggested, these nets do not fully support the object-oriented concepts. In this paper, we propose a hierarchical object-oriented Petri net (HOONet). The formal syntax and semantics of HOONet are explained in detail. HOONet supports a wide range of object-oriented features including abstract, encapsulated and modularized objects, object interaction by message passing, inheritance, and polymorphism. HOONet also supports a variety of modeling and analysis mechanisms such as incremental modeling of evolving systems, unfolding the HOONet to lower level Petri net, and incremental reachability analysis for HOONet models. We demonstrate the usefulness of HOONet by applying it to modeling and analysis with an example. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Hierarchical object-oriented Petri net; Requirement analysis; Software modeling; Incremental analysis

* Corresponding author.
E-mail addresses: jehong@salmosa.kaist.ac.kr (J.-E. Hong), bae@salmosa.kaist.ac.kr (D.-H. Bae).

0020-0255/00/$ - see front matter © 2000 Elsevier Science Inc. All rights reserved.
PII: S 0 0 2 0 - 0 2 5 5 ( 0 0 ) 0 0 0 9 0 - 6
1. Introduction

The development of a software system begins with two main activities: requirements analysis and system modeling. Requirements analysis serves two purposes: to thoroughly understand the problem and to reduce potential errors caused by ambiguous requirements. The purpose of system modeling is to depict the overall structure of the system by decomposing it into its logical components. Many researchers have focused their interests on the development of techniques to support these two activities [28].

There are two ways to achieve the goals of those two activities. One is to formally specify and analyze systems, and the other is to naturally describe and model systems. When specifying, modeling, and analyzing the behavior of a critical and complex system, we choose a language which can formally depict the properties of that system. Formal languages support us in describing system properties clearly, exactly and in detail. In particular, Petri net is the most widely used in various application domains because of its simplicity and flexibility in depicting dynamic system behaviors, and its strong expressive and analytic power of system behaviors. Despite the powerful capability of formal methods, designers are still likely to recognize the shortage of formal methods when analyzing and modeling systems of various domains. Thus, many Petri net researchers have devoted their efforts to enhancing and extending the theory and techniques of Petri nets [6,7,10] including high-level Petri nets such as colored Petri nets (CPN) [16,18].

The concepts of object-oriented paradigm such as encapsulation, inheritance, etc., have been widely used in the system modeling because they allow us to describe systems easily, intuitively, and naturally [9,17,25]. Designers who are familiar with formal methods have come to understand the usefulness of the object-oriented concepts. Along with this trend, object-oriented formal methods have also become of particular interest to researchers in recent years, and many experts have suggested object-oriented formal methods such as object Petri nets (OPN) [2], VDM++ [12], Object-Z [26], etc. Among these studies, research on OPN formalism has been actively studied to extend Petri nets formalism to OPN such as OBJSA [3], COOPN/2 [4], and LOOPN++ [19]. Although the results of such studies have shown promise, these nets do not support fully all the major concepts of object-oriented paradigm. Detail comparisons are described in Section 6.

In this paper, we formally define hierarchical object-oriented Petri net (HOONet), a high-level Petri net that supports object-oriented concepts. Modeling features in HOONet support information hiding through optimized interfaces; abstraction through abstract places, transitions and tokens; polymorphism through dynamic binding of abstract tokens; inheritance through properties shared among classes; and interaction through message passing. We also propose two analysis methods for HOONet models-unfolding of HOONet
to lower-level Petri nets for structural analysis and incremental reachability analysis for behavioral analysis. We apply HOONet modeling and analysis methods to an example system in order to demonstrate its effectiveness.

This paper is organized as follows: In Section 2, we justify the need for defining HOONet by explaining the integration of Petri net formalism and object-oriented concepts which are required to accomplish our goal. We also provide an informal and intuitive introduction to HOONet as well as formal definitions and behavioral semantics. We explain the representation of object-oriented concepts using HOONet in Section 3. Section 4 describes an overall procedure for modeling a system with HOONet, and then explains each step in detail by modeling an example car rental service system. In Section 5, analysis techniques for ensuring the accuracy of structural and behavioral properties in HOONet models are discussed. We also provide the algorithms to generate a reachability graph of HOONet model. In Section 6, we briefly review some of the earlier proposals for OPN and identify their shortcomings by comparing them with respect to expression power of object-oriented concepts. Section 7 concludes our paper and suggests further research issues.

2. Hierarchical object-oriented Petri net

The purpose of designing HOONet is to aid in the modeling and analysis of object-oriented software systems and to bridge the gap between formal treatment of Petri nets and object-oriented approach for the modeling, analysis, and prototyping of complex software systems.

2.1. Definition of the HOONet

A HOONet model is a variant Petri-net representation that corresponds to a class in object-oriented paradigm. The HOONet is composed of several parts: object identification place (OIP) is a unique identifier of a class; internal object net (ION) is a net to depict the behaviors (methods) of a class; and data dictionary (DD) declares the attributes of a class. The formal definition of HOONet is given as follows:

**Definition 2.1.** HOONet is a 3-tuple, \( \text{HOONet} = (\text{OIP}, \text{ION}, \text{DD}) \) satisfying the requirements below:

1. OIP is a special place which is defined as a tuple \( \text{OIP} = (\text{oip}, \text{pid}, M_0, \text{status}) \), where
   - oip is a variable for the unique name of a HOONet,
   - pid is a unique process identifier to distinguish multiple instances of a class, which contains return address,
   - \( M_0 \) is a function giving tokens with specific values to OIP,
   - status is a flag variable to specify the state of OIP.
2. ION is a variant CPN representing the changes in the values of attributes and the behaviors of methods. It will be formally defined in Definition 2.2.

3. DD is a declaring part for variables, token types, and functions.

The general structure of HOONet is shown in Fig. 1.

**Definition 2.2.** An internal structure of HOONet, ION = (P, T, A, X, N, G, E, F, M0), where
1. P and T are finite sets of places and transitions, respectively,
2. A is a finite set of arcs such that \( P \cap T = P \cap A = T \cap A = \emptyset \),
3. X is a function mapping from P to a set of token types declared in DD,
4. N, G, and E mean the functions of nodes, guards, and arc expressions, respectively, which are the same as defined in [18],
5. F is a special arc from any transitions to OIP, and notated as a body frame of ION, and
6. M0 is a function giving an initial marking to any place.

When modeling a system using several HOONet features, defined in the above, we also need some means to represent abstract information (i.e., abstract state and abstract action) and interactions between subsystems. The following definitions are supporting such means.

**Definition 2.3.** A set of places in HOONet is defined as \( \mathcal{P} = \{ \text{PIP}, \text{ABP} \} \), where
1. Primitive place (PIP) is a basic place to represent local states of a system, the same as in basic Petri nets [23].
2. An abstract place, ABP = (pn, refine_state, action) represents an abstract state, where
• pn is the name of an abstract place,
• refine_state is a flag variable denoting the refinement of an ABP,
• action is the static reaction imitating the internal behaviors of ABP.

The ABP is to represent abstract information (state), and can be refined in further modeling step. In order to indicate whether the refined information is modeled using HOONet features, the flag variable “refine_state” has a value such as true or false. The abstract place is depicted with bold-lined circle in HOONet model.

Definition 2.4. A set of transitions in HOONet, \( \mathcal{T} = \{\text{PIT, ABT, COT}\} \), where

1. Primitive transition (PIT) is a basic transition, the same as in basic Petri nets [23].
2. An abstract transition \( \text{ABT} = (\text{tn}, \text{refine_state}, \text{action}) \), where
   • \( \text{tn} \) is the name of an abstract transition,
   • \( \text{refine_state} \) has the same meaning as in the definition of ABP, and
   • \( \text{action} \) is the static reaction imitating the internal behaviors of ABT.
3. A communicative transition \( \text{COT} = (\text{tn}, \text{target}, \text{comm_type}, \text{action}) \) is a transition representing a method calling, where
   • \( \text{tn} \) is the name of a communicative transition,
   • \( \text{target} \) is a flag variable denoting whether the called method by COT was modeled (a “yes” value) or not (a “no” value),
   • \( \text{comm_type} \) is also a flag variable representing whether the interaction of COT is synchronous (a “SYNC” value) or asynchronous (a “ASYN” value), and
   • \( \text{action} \) is the static reaction reflecting the execution results of the called method.

In general, when the model includes abstract information (behavior and states), it is difficult to simulate the system behavior represented in this model. However, it is possible to simulate the model with abstract information in HOONet. The variable “action” is represented with algebraic expression to define the post-condition of abstract behavior. This variable is used to obtain an artificial effect, as if the actual execution of the abstract place or abstract transition had occurred. In HOONet model, ABT and COT are represented with bold-lined rectangle and double-lined rectangle, respectively.

Token types in HOONet are described in DD, and classified into two categories: primitive type and abstract type. The primitive type is the same as that of CPN, while the abstract type is a compound of primitive types. The abstract type is required to express the states of the abstract place. The abstract type is declared with “complex” type or “record” type. The token type of a HOONet model which is represented with abstract information, should be also represented with the abstract token type, since the token type expressed with detailed information does not adequately represent the states of abstract
behaviors in the model. 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 models. The decomposition of the abstract token is explained in object-oriented representation (Section 3).

The definition of DD is very straightforward, is written in a textual grammar such as CPN ML [29]. The detailed grammar is explained with an example in Section 4.

### 2.2. Behavioral semantics of HOONet

Behavioral semantics of HOONet does not violate the semantics of CPN formalism such as binding, firing, and so on. However, it should be defined for behavioral semantics of some features of HOONet which are not supported in CPN formalism. Before describing in detail the formal semantics of HOONet, we explain the overall behavior of HOONet models.

As shown in Fig. 2, the HOONet model “Client” at the left side contains the abstract place \( A_p \), and the model at the right side represents the refinement of the abstract place. When a token is given to OIP “Client” of the left-side model in Fig. 2, a transition among \( T_1 \), \( T_2 \), and \( T_3 \) is enabled and fired. At the entry point of the abstract place \( A_p \), the token is transferred to the OIP of the refinement model when the value of the variable “refine_state”, defined in Definition 2.3, is true (Step 1 in Fig. 2). After the completion of internal behavior of the refined model (Step 2), the token flows to the OIP \( A_p \) through the body

---

**Fig. 2. Logical concept of HOONet behavior.**

---
frame of the refined model (Step 3), and then returned to the abstract place $A_p$, carrying with the execution results (Step 4).

2.2.1. Behavioral semantics of OIP

As explained above, the binding and firing actions of OIP are constrained by the variables which define the state of OIP. The behavioral semantics of OIP is defined as follows:

$$\forall t \in \text{OIP} \land \forall v \in \text{Var}(t) : B(t).$$

if status = “pre”

$$M(X) \lor M(X\bullet) \text{ for } X \text{ such that OIP.pid.return = X}$$

if status = “post”

where Var(t) means a set of variables appearing in the guard function and the arc expression of $t$, and $B(t)$ means the binding of $t$.

2.2.2. Behavioral semantics of abstract place

The abstract place shows different behaviors depending on that the abstraction is refined or not. When the abstraction was refined, the binding and firing occur from the abstraction to its refined model. Otherwise, a transition among ABP$\bullet$ will be fired. The behavioral semantics of ABP is defined as follows:

$$M(\text{OIP}) \text{ such that ABP.pn = OIP.oip}$$

if refine_state = true

$$R(\text{action}); \forall t \in \text{ABP} \land B(t)$$

if refine_state = false

where $R(\text{action})$ means an evaluation of the expression which is defined in variable “action” of ABP.

2.2.3. Behavioral semantics of abstract transition

The abstract transition shows different behaviors depending on that the abstraction is refined or not. When the abstraction was refined, the binding and firing occur from the abstraction to its refined model. Otherwise, a place among ABT$\bullet$ will be marked. The behavioral semantics of ABT is defined as follows:

$$M(\text{OIP}) \text{ such that ABT.tn = OIP.oip}$$

if refine_state = true

$$R(\text{action}); +M(\text{ABT}\bullet) \land -M(\bullet\text{ABT})$$

if refine_state = false

where $+M(\text{ABT}\bullet) \land -M(\bullet\text{ABT})$ means the addition of tokens to output places and deletion of tokens from input places by the firing of the ABT.

2.2.4. Behavioral semantics of communicative transition

The interactions between HOONet models (i.e., classes/objects) are occurred differently depending on the type of the communication and whether the interacting model is drawn or not. This behavior of COT is defined as follows:
2.2.5. Behavior of inheriting component

HOONet supports the inheritance mechanism of object-oriented paradigm more concretely than other OPN formalisms. When a HOONet model is defined with general common properties, its child models can also be defined that inherit the common properties. This concept allows to reduce the complexity of drawn models, and allows to reuse the parent model. The inheritance concept and its semantics in HOONet are explained in detail in Section 3.3 with an example.

3. Object-oriented representation in HOONet

HOONet provides more expressive power than existing OPN formalisms with respect to its support to object-oriented concepts. In order to model object-oriented system requirements, HOONet supports the representation of class/object structure, information hiding, abstraction hierarchy, and inheritance. Using the HOONet features which were defined in the previous section, we explain the representation of object-oriented concepts in HOONet in this section.

3.1. Information hiding

A class in object-oriented paradigm is represented with a HOONet model, and it consists of an OIP, a DD (corresponding to data), and an ION (corresponding to behaviors). The OIP in a HOONet model plays the role of a public interface by having a unique name for external reference, and also the OIP is a place which initiates and terminates the internal behavior of the model. The DD is a dictionary to refer detail definitions of the variables, functions and token types used to describe internal behaviors in the ION. These structural components of HOONet can fully support the information hiding concept that the internal details of HOONet model are treated as black boxes and are hidden from view.

3.2. Abstraction

In system design, abstraction means focusing on what an object is and does, before deciding how it can be refined into detail aspects. The use of abstraction
preserves the freedom to make decisions for as long as possible by avoiding premature commitments to details [6, 21, 25]. Roughly, abstraction can be classified in two categories: abstraction that aids in specifying control, and abstraction that allows us to hide the actual representation of the data [1, 9]. In HOONet models, the concept of abstraction is reflected in places, transitions, and tokens. The abstraction of these components is needed, as noted in [21], to maintain the symmetric structure of nets and analyze net models containing abstract states and behaviors. Conceptually, ABP, ABT, and abstract token (ATN) are to represent the abstractions of object states, method behaviors, and token types, respectively.

The property of abstraction and its refinement can be defined with the morphism between the abstraction and its refinement. Morphism shows that the constraints of abstraction are preserved in its refined model (RM) [21]. In the HOONet formalism, the sine qua non condition of morphism is showing that the observable behaviors between the abstract component and its refined components are equivalent, and that the token types in abstraction are fully contained in the DD of the refinement.

When an ABP is refined, the constraints that should be preserved in the refined model are the token types and arc expressions of incoming and outgoing edges of the ABP. In case of ABT, the token types of input and output places and guard expressions of the ABT are the constraints. The definition of observable morphism in the HOONet model is given as follows:

**Definition 3.1.** The observable morphism, $\forall C \in \{\text{ABT, ABP, ATN}\}, \; C \Rightarrow \text{RM}$, should satisfy the requirements below (where, $\Rightarrow$ means “is refined to”):

1. For token types of ABP and ABT,
   \[ \forall p \in \text{ABP}, \; \text{TYPE}(p) \subseteq \text{TYPE}(\text{TT}_{\text{RM}}), \; \text{and} \]
   \[ \forall t \in \text{ABT}, \; \{\text{TYPE}(\bullet t) \cup \text{TYPE}(t\bullet)\} \subseteq \text{TYPE}(\text{TT}_{\text{RM}}), \]
   where $\text{TT}_{\text{RM}}$ is a set of all token types defined in DD of RM. And $\text{TYPE}(p)$ means a mapping of each place $p$ to a set of token types.

2. For all incoming and outgoing arc expressions $a_i, \; a_o$ of ABP, and for all $a_i$ of ABT and all $a_o$ of $\text{ABT} \bullet$,
   \[ \bigcup_{a_i, a_o} \text{Var(expr)} \subseteq \text{Var(OIP}_{\text{RM}}). \]

3. For all guard expressions of ABT, $t \in \text{ABT}$, $G(t) \equiv \bigcup_{\forall l \in \text{OIP}_{\text{RN}}} G(t_l)$, where $G(t)$ is a set of all guard functions of $t$.

Abstraction and its refinement make it possible to model the system by an incremental and hierarchical approach; this is a useful mechanism to reduce complexity in the process of analysis and design.
3.3. Specialization by inheritance

Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship. A class can be defined broadly, and then 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 repeated in each subclass [9,25].

The notation of inheritance in the HOONet model is shown in Fig. 3. Fig. 3 represents the subclass Customer which inherits all properties from superclass Person. The symbol expressing inheritance is “::<”. This inherited class Customer includes an added behavior which is led by “aspect” transition, and includes a redefined behavior which is led by “license” transition.

When the inherited class is instantiated, the DD of the subclass is extended to include that of the superclass, and a set of transitions which can be enabled with $E(OIP, t)(b)$ is also extended. An issue to be considered in the inheritance mechanism is overriding the properties of a superclass by modification. The overriding features (e.g., “license” in Customer model) of the subclass refine and replace the overridden features (e.g., “license” in Person model) of the superclass. The reasons for overriding the features (methods) of the superclass are to specify any behavior that depends on the subclass, and to tighten the specification of a method. It should be determined which method is supposed to be bound between the overridden method and the overriding method, because overriding redefines a method with the same name and does not permit overriding the signature or the form of a feature. This decision is reached by

Fig. 3. Inheritance representation of HOONet model.
the scoping rule of the (dynamic) binding. The rule searches up the inheritance hierarchy to bind the method that implements a specific operation in the inherited object.

**Definition 3.2.** *Inheritance* of a class from a superclass satisfies the requirements as follows: Let I, S, and N be an inherited subclass, a superclass, and the newly defined and overriding methods, respectively.

1. \( TT_I = TT_S \cup TT_N, \)
2. a set of enabled transitions, \( ET_I = \{ t_i | \forall t_N, E(OIP_I, t) \} \cup \{ t_j | \forall t_S, E(OIP_S, t) \} \), and
3. for \( t_i \in T_N \land t_j \in T_S, \exists t_i, t_j \in ET_I \mapsto ((t_i, b) \in Y \land (t_j, b) \notin Y). \)

### 3.4. Parametric polymorphism

Supporting polymorphism means that it is possible to dynamically bind a token type depending on the value of an incoming token. Operationally, dynamic binding may be regarded as a dispatching mechanism that acts as a case statement to select (dynamically) the appropriate behaviors in response to the value of a token. In the HOONet method, polymorphism can occur in cases where the variable refine_state of ABP or ABT is true, and a COT calls the OIP of an external class. Dynamic binding for polymorphism in abstract components progresses in two steps: first, the complex type of a token is decomposed into several detail types of tokens of OIP; the values of tokens in OIP are then defined with specific values depending on the value of the complex type. For example, in Fig. 3, an abstract place “ap1” is a place representing whether mental or physical status of a customer is normal or abnormal (i.e., \( TT_C = \text{complex with normal} \mid \text{abnormal}; \)). When the token value of C is normal, and the normal status means the mental status of a customer, the token type C is decomposed into

\[
TT_C = \text{record with }
\]

\[
\begin{align*}
\text{IntQ} &= [0..110] \mid [111..130] \mid [131..150]; \\
\text{EmoQ} &= [0..3] \mid [4..7] \mid [8..10]; \\
\text{Spek} &= \text{trouble} \mid \text{common} \mid \text{fluent};
\end{align*}
\]

Then, the token value of each type is assigned to \( \{(\text{IntQ}, [111..130]), (\text{EmoQ}, [4..7]), (\text{Spek}, \text{common})\} \) in the refined model of the “ap1”. However, in case that physical status is normal, the token type C is decomposed into

\[
TT_C = \text{record with }
\]

\[
\begin{align*}
\text{Height} &= \text{short} \mid \text{average} \mid \text{tall}; \\
\text{Weight} &= \text{light} \mid \text{middle} \mid \text{heavy}; \\
\text{Heart} &= \text{low} \mid \text{center} \mid \text{high};
\end{align*}
\]

and, the token value is assigned to \( \{(\text{Height, average}), (\text{Weight, middle}), (\text{Heart, center})\} \) in the refined model. In this manner, a token
type can be bound with different subtypes according to the value (here, mental or physical) of a parametric variable.

Another type of polymorphism can also be considered about the attributes and methods of the class created by inheritance. This type of polymorphism was explained with a scoping rule and its dynamic binding in Section 3.3, along with inheritance.

3.5. Message passing between objects

The passing of message between objects occurs when a method calls another object for interaction. A method call transfers a token from the COT to the OIP of the called object (HOONet). Depending on the values of the incoming token of the OIP, it is determined which transition is enabled and fired among the transitions in the OIP. When the execution of a specific method ends, the results of the execution are transferred to the COT through its own OIP. There are two issues to be considered:

1. Synchronous or asynchronous method calls: the COT must receive the results of the execution of called methods in the synchronous call. So, the places in the COT cannot be marked until the result is returned. At this time, the value of target is “yes”. In the asynchronous call, even though the token is transferred from COT to OIP, the places in the COT should have markings. In the COT defined with the asynchronous type, the token is passed to both the OIP and COT. The result of execution of a called method is stored in the global variables for later use.

2. Nesting of method calls: during a method execution, it is possible to call the methods of other objects or of itself. We provide the variable pid to represent the nesting of method calls. When a COT calls a method, the variable pid is set with the value of a transferring token and the process identifier indicating the instance of the class created by the method call. The size of the variable pid is proportionally increased whenever a subsequent call occurs.

3.6. Object instantiation

An instance can be created when a class initiates the system behavior (This corresponds to \texttt{main()} function in procedural languages), or a class is called from other class (or object). At the creation of an instance, the instance has a unique identifier which is distinguished with the value of variable pid. The pid value of the instance corresponding to \texttt{main()} function, is “self”, and the pid value of called instance is the returning pointer to the calling class.

For a more detailed explanation, the procedures to modeling an example system are presented in the next section.
4. Modeling a system using HOONet

4.1. Modeling and analysis process

The overall modeling and analysis process of system requirements using HOONet is shown in Fig. 4. In order to design a system using HOONet, we first briefly decompose the requirements of the system, and specify the requirements by formal methods. In the early process of system modeling, we do not know exact detailed information of the system. Thus, the first specification of requirements will be described with HOONet including only high-level and available information about the system. In the next step, the specification expands to models of the system, and these models are refined with information gained from previous steps and further analysis of requirements.

There exists some feedback and adoptions of design information between steps. After completion of the system modeling, we analyze the correctness of the models and the behaviors of the system. Additionally, we note the fact that the correctness of the HOONet models can be analyzed despite that the modeling is not completed. These steps are based on the concepts of abstraction and decomposition (or refinements), and can be performed in an iterative and incremental way [15]. Steps 1 and 2 will be explained in Sections 4.2.2 and 4.2.3, respectively, and Step 3 will be explained in Section 5.

![Diagram of the design process of a system using HOONet.](image-url)
4.2. Modeling with an example system

4.2.1. Simple requirements of a CRS system

The initial requirements for a car rental system (CRS) are as follows:

- The CRS system supports basic rental services, such as rental of cars, return of the rented cars.
- To rent a car, a customer must take out an insurance policy, and a rental agent checks the customer’s credit standing.
- The information on a customer consists of a name (identifier), the driver’s license number, the credit card number, etc; information on the car includes an identifier, the type of car, the state of the car, insurance cost, etc.
- When an accident occurs after the rental, the customer contacts the rental agent, rather than the insurance agent.
- When a car is returned after an accident, the rental agent sends the car to the insurance agent. When the car is returned from the insurance agent, it is assumed that the state of the car is fully recovered.
- Even if the rental period of the car has not expired, the customer involved in an accident should return the car, and should pay the appropriate fee by agreement.
- Other requirements comply with explanatory remarks.

4.2.2. Decomposing user requirements

Before formally writing the requirement specifications, it is necessary to rearrange user requirements by functional decomposition. The CRS system has such representative entities (actors) as “customer (CUST)”, “rental agent (REAG)”, and “insurance agent (INAG)”. There are some interactions among them, such as “rent a car”, “take out an insurance”, and so on. Table 1 shows the abstract data and functions for each entity.

The abstract data and functions of each entity can be translated into the states and their changes of each entity. This represents a system with state-based Petri nets. For example, the function “request car” is one of the customer’s behaviors, and the previous and subsequent states of the behavior are determined by a set of the values of variables which comprise the abstract data “customer info”.

Table 1
The high-level information of CRS system

<table>
<thead>
<tr>
<th>Entity</th>
<th>Abstract data</th>
<th>Abstract functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUST</td>
<td>customer info.</td>
<td>request car, operate car, notice accident, return car</td>
</tr>
<tr>
<td>REAG</td>
<td>car info., rental info.</td>
<td>rent car, take insurance, manage car, receive car, request repair</td>
</tr>
<tr>
<td>INAG</td>
<td>insurance info.</td>
<td>support insurance</td>
</tr>
</tbody>
</table>
1. Customer (CUST)
   - The state of a customer is divided into (1) Before_rental, (2) During_rental, and (3) After_return, and the state “During_rental” is further divided into (2.1) Normal_operation and (2.2) Under_accident, more precisely.
   - The change of states can be defined with a tuple, \((\text{previous state}, \text{actions}, \text{sequent state})\). That is, the subsequent state is determined by the occurrence of an action in a specific state.

<table>
<thead>
<tr>
<th>No.</th>
<th>Previous state</th>
<th>Actions</th>
<th>Sequent state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Before_rental</td>
<td>request car</td>
<td>During_rental</td>
</tr>
<tr>
<td>2</td>
<td>During_rental</td>
<td>operate car</td>
<td>Normal_operation</td>
</tr>
<tr>
<td>3</td>
<td>Normal_operation</td>
<td>occurs accidents</td>
<td>Under_accident</td>
</tr>
<tr>
<td>4</td>
<td>Normal_operation</td>
<td>normally return car</td>
<td>After_return</td>
</tr>
<tr>
<td>5</td>
<td>Under_accident</td>
<td>return car with accident</td>
<td>After_return</td>
</tr>
</tbody>
</table>

2. Rental agent (REAG)
   - The high-level states of REAG are (1) Car renting and (2) Car receiving. The state (1) is divided into three states such as (1.1) Ready_rent, (1.2) Rented_car, and (1.3) Not_rentable (impossible to rent). Recursively, the state Rented_car is divided into (1.2.1) Deliver_car and (1.2.2) Take_insurance. The state (2) divided into (2.1) Normal_receiving and (2.2) Accident_receiving, and for the state (2.2), it did into (2.2.1) Accept_accident, (2.2.2) Before_repair, and (2.2.3) After_repair.
   - The actions causing the state changes of the rental agent are “process the rental”, “take out insurance”, “process the return”, and “process the accident”.

3. Insurance agent (INAG)
   - The INAG has the three states, such as (1) Ready_insurance, (2) Accept_insurance, and (3) Expire_insurance, and its actions are “grant insurance”, “process insurance”, and “perform repair”.

4.2.3. Specify and model the system
(1) High-level specification and modeling. The HOONet model for “CUST”, depicted with the decomposed requirements in the above, is shown in Fig. 5. The “CUST” model has an abstract place \(\text{opeCar}\) and two communicative transitions \(\text{REAG}_1\) and \(\text{REAG}_2\). These components represent that a customer interacts with a rental agent to rent or return a car. Among them, the place \(\text{opeCar}\) includes such behaviors as operating a car and returning a car. Also, returning action can include “return by normal” and “return by accident”.

(2) Data dictionary. The DD declares the token types, variables, and functions. It can also contain some expressions which are post-conditions of ABPs,
ABTs and COTs. DD is a variant of CPN ML (or SML) in terms of its syntax and semantics [29]. We have some types to declare as components of the model; that is a set of types = \{primitive, complicated, user_defined\}. The primitive types include boolean, string, alphanumerics, integer, real, and array; the complicated types are record and complex type. The record type means that several primitive types can be declared as a composite type, and can have record types to define the nested token structure. The complex type means those types can be comprised with several undefined types. Therefore, those types are refined into several types through dynamic binding of parameters. The symbol “+” in a type means a globally referenced type. The DD of the HOONet model “CUST” is as follows:

**Level 1: Data Dictionary of “CUST”**

Var +CX = Boolean; /* rent or not */
TT C = complex;
COT(REAG) = {
    Fun(CX = = F): CX = T ∧ M(OpeCar,C);
    Fun(CX = = T): CX = F ∧ M(AftRet,C);
};

In this DD, the token type C in CUST was defined with the complex type. This token type will be decomposed into refined token types in the Data Dictionary.

Fig. 5. The HOONet model of “CUST”.

![Diagram](image-url)
at the next level. The global variable “CX” indicates whether or not the customer rents a car. The “action” term COT(REAG), corresponding to the post-condition of COT, changes the value of CX to true (T) and marks at the place OpeCar when the value of CX is false (F). This “action” will be discarded and does not affect the behavior of the COT if HOONet model REAG were drawn.

(3) Detail specification and modeling. At level 1, the model for REAG can be also drawn to represent the rental agent’s behaviors. The main activities of REAG are renting a car and receiving a car. The HOONet model for REAG is shown in Fig. 6.

The DD of REAG is similar to the Data Dictionary of CUST. The global variable “CX” is also used in this model. In REAG model of Fig. 6, the actions starting from the transition RL_t10 represents the behavior of renting a car to a customer, and the actions led by the transition RL_t20 is for receiving a car from the customer’s return. The behavior for receiving a car can be also decomposed into two major behaviors: normal receiving and receiving after

![Fig. 6. The “REAG” model at level 1.](image-url)
accident. The refinement of this part is omitted. Here, the place “NotRen” represents a status in which the customer cannot rent a car for any reason.

The abstract places and transitions are refined into more detailed models at level 2. We consider two refined HOONet models: the abstract place OpeCar in CUST and the abstract transition RENTING in REAG. Fig. 7 shows the HOONet model “OpeCar”. This model represents the customer’s behaviors for return by normal and return by accident of the rent car.

The DD of this “OpeCar” model is as follows:

```
<table>
<thead>
<tr>
<th>Level 2: Data Dictionary of “OpeCar”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var +CA = Boolean; /* accident or not */</td>
</tr>
<tr>
<td>TT C = record with {</td>
</tr>
<tr>
<td>TT CI = record with {</td>
</tr>
<tr>
<td>c_name = string;</td>
</tr>
<tr>
<td>license = alphanumeric;</td>
</tr>
<tr>
<td>credit = string;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>TT CR = record with {</td>
</tr>
<tr>
<td>car_type = alphanumeric;</td>
</tr>
<tr>
<td>rent_period = date;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>COT(REAG) = {</td>
</tr>
<tr>
<td>Fun(CA = = F): CX = F \ M(EndSer, C);</td>
</tr>
<tr>
<td>Fun(CA = = T): CX = F \ M(EndSer, C);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
```

The DD shown above is the refined declaration of the DD of the “CUST”. The complex typed token C is decomposed into two record typed token CI and CR.

Fig. 8 shows the refinement of the abstract transition RENTING shown in Fig. 6. There are two reasons refusing to rent to a customer: low credit of customer \[GC = \text{“Bad”}\] and unavailable car\[(RI.car_type = = RV.types) \land (\text{return\_date} \neq \text{null})\]. These two needs should be satisfied in customer’s constraints to rent a car.

In Fig. 8, the place ReagDB is a place containing information on the customer’s credit [RL] and on the rental car[RV]. This place can also be represented as an abstract place because it can contain the actions such as retrieve, update, and delete data. The OIP RENTING and ReagDB places are initially marked when the RENTING is instantiated.

The DD of this model is as follows (where, the type RL is for the customer list of the rental agent, and RV is for the list of cars being held by the agent. The
variables RI and RC are declared with the types used in the DD of the model “CUST”):

**Level 2: Data Dictionary of “RENTING”**

```plaintext
TT R = record with {
  RI = CR;
  RL = complex;
  RV = complex};
Var n = integer with 1...1000; /* number of customers*/
Var m = integer with 1...100; /* number of cars*/
TT RL = record array [1...n] with {
  RC = CI;
  GC = with “Best”|“Good”|“Bad”| null}; /* grades of credits*/
TT RV = record array [1...m] with {
  types = alphanumeric;
  rent_fee = integer;
  return_date = date;
  car_states = SC};
```

The other models for the CRS system, are not presented here. The class INAG for the insurance agent at level 1 and a class for checking the current state of a car at level 3, can be modeled by refinement. However, we omit these models since we believe that the application of HOONet has been adequately explained.

When modeling a system with HOONet, we need not consider all the complex and extensive information about the system, because the modeling steps are based on refinement and incremental development. The reachability analysis of models can also be achieved incrementally. This approach can help us to handle large and complex software systems, and to easily modify the models.

5. Analysis of the HOONet models

We suggest two schemes to analyze the HOONet model. One is the unfolding of nets to analyze the structural and interactive behaviors of HOONet models. Our unfolding mechanism expands the components of HOONet such as OIP, ABP, ABT and COT into a lower-level Petri net: which is equivalent to CPN. Such expansion makes it possible to clearly represent the global
behaviors of the system, and to composite the models to represent the global system structure. The second scheme is the analysis of behavioral properties depicted in the HOONet models, which is achieved by performing reachability analysis for the dynamic properties of a system such as liveness, boundedness and conservation [18,23].

5.1. Unfolding of HOONet models

5.1.1. Unfolding OIP

OIP is a place with the role of a class identifier and the entry and exit point of a class. The unfolding of a HOONet model which is encapsulated with OIP and its body frame of the model, prepares a separation for the entry and exit points of the model. This can be easily achieved by duplicating the OIP as the exit after the removal of the body frame of the model, as shown in Fig. 9.
5.1.2. Unfolding of ABP and ABT

ABP and ABT are useful to represent the abstract information of system requirements. The unfolding of ABP or ABT substitutes the abstraction components with the refined model. In order to substitute the former with the latter, the unfolding for OIP of the refined model should precede. The representations for unfoldings of ABP and ABT are shown in Figs. 10 and 11, respectively. One issue in the unfolding of ABT is, as shown in Fig. 11, splitting the abstract transition into two internal sub-transitions. The transition \( i_{AT1} \) acts as the invocation of the refined model, and the transition \( o_{AT1} \) ends the invocation. Therefore, the unfolding of ABT is the insertion of the refined model between \( i_{AT1} \) and \( o_{AT1} \).

5.1.3. Unfolding of COT

COT is a feature that supports communication between objects. The communication mechanisms are classified into two categories in HOONet.
modeling: synchronous and asynchronous communication. The unfolding of
synchronous COT begins by splitting the COT into $o\_COT$ and $i\_COT$; an arc
is then inserted from $o\_COT$ to OIP of other interacting object and from dup-
plicated OIP to $i\_COT$, respectively. Fig. 12 shows the unfolding of COT; the
COT $O1$ is split to $o\_O1$ and $i\_O1$ in this figure.

The unfolding mechanism of asynchronous COT differs slightly from that of
synchronous COT. After bisecting the COT, an arc is inserted from $o\_COT$ into the OIP of other interacting object. This is shown in the lower right side of
Fig. 12.

Fig. 9. Unfolding OIP of HOONet model.

Fig. 10. Unfolding ABP of a HOONet model.
5.1.4. Unfolding of ATN

An ATN has abstract values to represent abstract states of a system. The unfolding of ATN is accomplished together with the refinement of ABP or ABT. We have explained how to decompose the token of a complex type into its several subtypes in Section 3.4. This decomposition is a concept similar to unfolding of ATNs.

For example, let “acc_type” be a type representing the extent of a car accident. At the high-level models, the declaration of the type “acc_type” will be defined as follows:

\[
\text{TT acc\_type} = \text{with waste} \mid \text{critical} \mid \text{smolder} \mid \text{trauma};
\]

If a car consists of six components such as engine, shaft, gear, radiator, wheel, and hood, the token types to represent damage to each component, in the low-level models, will be declared as follows:

\[
\begin{align*}
\text{TT Engine} &= \text{with disable} \mid \text{damage} \mid \text{unimpact}; \\
\text{TT Shaft} &= \text{with broken} \mid \text{curved} \mid \text{abrasion} \mid \text{upright}; \\
\text{TT Gear} &= \text{with collapse} \mid \text{defect} \mid \text{operate}; \\
\text{TT Radiator} &= \text{with disrupt} \mid \text{fumes} \mid \text{normal}; \\
\text{TT Wheel} &= \text{with crook} \mid \text{flat\_tire} \mid \text{optimal}; \\
\text{TT Hood} &= \text{with bend} \mid \text{scratch} \mid \text{slick};
\end{align*}
\]

Here, we should consider the mechanism of unfolding the abstract token values in high-level models into the token values in low-level models.
Therefore, the unfolding function of ATN should be declared in the DD of the low-level model as follows:

\[
\text{TT waste} = \text{record with } \{ \text{engine} = \text{TT Engine with disable}; \\
\text{shaft} = \text{TT Shaft with broken | curved}; \\
\text{gear} = \text{TT Gear with collapse}; \\
\text{radiator} = \text{TT Radiator}; \\
\text{wheel} = \text{TT Wheel}; \\
\text{hood} = \text{TT Hood}; \\
\};
\]

\[
\text{TT trauma} = \text{record with } \{ \text{engine} = \text{TT Engine with unimpact}; \\
\text{shaft} = \text{TT Shaft with upright}; \\
\text{gear} = \text{TT Gear with operate}; \\
\text{radiator} = \text{TT Radiator with normal}; \\
\text{wheel} = \text{TT Wheel with flat_tire | optimal}; \\
\text{hood} = \text{TT Hood}; \\
\};
\]

Fig. 12. Unfolding COT in HOONet model.
As described in the above example, the ATNs are defined with all variables declared in the lower model. The token values which are assigned by decomposing of the abstract token can be partially overlapped among decomposed tokens in low-level model, but the value of each token is unique. It can also be represented as the abstract token type with the disjoint range of value – e.g., the degree of temperature, pressure, and so on.

5.2. Reachability analysis of HOONet models

In order to check the consistency or the completeness of system requirements in HOONet models, analysis technique based on the reachability graph is introduced. Our analysis scheme provides the following characteristics to reduce the state space explosion in the conventional reachability analysis.

- Reachability graphs can be generated and analyzed for each HOONet model.
- Although the whole system is not completely modeled, the reachability analysis for a part of the system is possible.
- The compositional reachability analysis can be performed for HOONet models since each HOONet model has one-entry and one-exit structure with respect to its behavior.

The reachability graph generation algorithm, supporting the characteristics mentioned above, for HOONet models is described in Algorithm “HOONet_Reachability_Graph”. Since Algorithm “HOONet_Reachability_Graph” is designed to analyze the HOONet model with abstract components, the analysis can be performed partially on the models of interest. To clarify the algorithm, some definitions follow:

- \( \text{Waiting} \) is a set of nodes, which contains those nodes for which we have not yet found successors.
- \( \text{Node}(M) \) is a procedure that creates a new node \( M \), and adds \( M \) to the set of \( \text{Waiting} \). If \( M \) is already a node, the procedure has no effect.
- \( \text{Next}(M_1) \) is to denote a set of all possible “next moves” from a marking \( M_1 \).
  i.e., \( \text{Next}(M_1) = \{(b,M_2) \in BE \times \mathcal{M}|M_1[b > M_2]\} \).
- \( \text{Arc}(M_1,b,M_2) \) creates a new arc with source \( M_1 \) and destination \( M_2 \).
- \( \text{GenNode}(M_1,M_2) \) is a procedure that creates a node and connects the node to the reachability graph.

begin
  Node(M_2);
  Arc(M_1,b,M_2);
  Waiting := Waiting - \{M_2\};
end.

- \( \text{AllGen}(M_1,M_2) \) is a procedure that performs the following:
begin
for all \((b, M_2) \in \text{Next}(M_1)\) do
GenNode\((M_1, M_2)\);
end.

As an additional note, if the source of a node is an element from among ABP, ABT and COT, the node should be considered as two nodes because they have the two states which are the initial and the post states of the primitive actions of the elements.

5.3. Algorithm \(\text{HOONet\_Reachability\_Graph}(\text{Nets, Graph})\)

Waiting := \(\emptyset\);
Node\((M_0)\);
repeat
select a node \(M_1 \in \text{Waiting}\);
Case of \(M_1\) on
OIP: if(status = "pre") \{AllGen\((M_1, M_2)\); break\};
else if(pid = "self") End_of_Algorithm;
else\{\(M_2 := \text{OIP.pid.return}\);
OIP.pid @ return;
Gen Node\((M_1, M_2)\);
break\};
PIP: \{AllGen\((M_1, M_2)\); break\};
ABP or ABT: if (not yet refined)\{AllGen\((M_1, M_2)\);
break\};
else\{OIP.pid @ return;
\(M_2 := \text{refined OIP}\);
GenNode\((M_1, M_2)\);
break\};
COT: if (not yet refined)\{do(static reactions) or
wait(user’s response);
AllGen\((M_1, M_2)\);
break\};
else if (synchronous call)\{OIP.pid @ return;
\(M_2 := \text{refined OIP}\);
GenNode\((M_1, M_2)\);
break\};
else\{AllGen\((M_1, M_2)\) || \(M_2 := \text{refined OIP}\);
GenNode\((M_1, M_2)\);
break\};
\}
until Waiting = \(\emptyset\)
End of Algorithm.
The reachability graph for HOONet model is incrementally generated depending on the types of places and transitions. As explanatory notes about Algorithm "HOONet_Reachability_Graph", if one of interacting classes was not modeled, the predefined post-conditions of the COT are executed, or the analysis progress is halted to accept user’s response. In case of asynchronous COT, the path in the graph branches off by performing parallel operations. The notations ⊕ and ⊖ mean concatenating and cutting the return address from oip.

Our suggested analysis technique can be very useful in the design and analysis of complex and large-scale systems because it is possible to generate reachability graphs for only the models of interest. For instance, in our example system “CRS”, it is possible to analyze the models without considering the insurance service.

Since the reachability graphs of HOONet models, generated by algorithm “HOONet_Reachability_Graph”, are equivalent with the reachability graphs of CPN model suggested by Jensen [18], we can check such properties as liveness, boundedness and conservation using CPN’s analysis techniques.

6. Related work

Recognition of the significance of requirements analysis and system modeling in software development process has resulted in a gradual increase in the use of formal methods such as Petri nets. Attempts to integrate object-oriented concepts into Petri nets, and vice versa, have been displayed in views such as LOOPN++ [19,20], COOPN/2 [4,5], OPNets [22] and G-Nets [24].

In this section, we survey the supportability of object-oriented concepts of existing OPN formalisms, and we compare those OPNs with HOONet within the following criterions:

- **Encapsulation**: Is the structural representation sufficient to depict a class, and do such representation support the concept of information hiding?
- **Abstraction**: In order to represent the abstract information of system requirements, is there any feature to symmetrically abstract system states and behaviors (i.e., data abstraction and control abstraction)?
- **Inheritance**: Is it possible to inherit and override all properties of superclass?
- **Dynamic binding**: Is the concept of parametric polymorphism supported for the interactions between objects?
- **Message passing**: In order to interact among modularized components or objects are synchronous and asynchronous communication mechanisms based on message passing supported?

With the criterions above, we summarized the comparison results between existing OPN formalisms and HOONet formalism in Table 2.

As shown in Table 2, even if most of existing OPN formalisms provide the structural representation for class/object modeling, supporting other criterions
is not enough to model the object-oriented software. COOPN/2 can define tokens as abstract data types (ADT) to support only data abstraction and LOOPN++ and OPNets use their abstraction concepts to reduce the complexity of drawn models. Fig. 13 shows the representation of its abstraction concept in OPNets formalism.

The boxed components “Object AA” and “Object AB” in Fig. 13 are defined with abstraction. However, these components are not places or transitions but simplified boxes. Thus, even though this feature is useful to increase the understanding of OPNets models, it does not support the concepts of data and control abstraction.

In case of provision of inheritance mechanism, existing OPNs are insufficient for redefining superclass’s properties. As an example, OPNets and G-Nets do not consider the notion of inheritance. Fig. 14 shows an example representation of inheritance mechanism in COOPN/2. In the class definition in textual description of COOPN/2, the expression of inheritance is clear, but it is not easy to distinguish overridden behavior from inherited behaviors. And also, it is difficult to know which classes are inherited classes with only graphical views of COOPN/2.

Table 2
Summary of comparison results for supportability of object-oriented concepts

<table>
<thead>
<tr>
<th>OPNs</th>
<th>Encapsulation</th>
<th>Abstraction</th>
<th>Inheritance</th>
<th>Dynamic bind.</th>
<th>Message pass.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COOPN/2</td>
<td>○</td>
<td>△</td>
<td>○</td>
<td>×</td>
<td>△</td>
</tr>
<tr>
<td>LOOPN++</td>
<td>○</td>
<td>△</td>
<td>○</td>
<td>×</td>
<td>△</td>
</tr>
<tr>
<td>OPNets</td>
<td>○</td>
<td>△</td>
<td>×</td>
<td>×</td>
<td>△</td>
</tr>
<tr>
<td>G-Nets</td>
<td>○</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>△</td>
</tr>
<tr>
<td>HOONet</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

* ○ – supported; × – not supported; △ – partially supported.

Fig. 13. Representation of abstraction concept in OPNets.
From Table 2, we know that existing OPNs do not support the notion of dynamic binding which occurs in the binding process of overriding methods or in the binding process of abstract tokens. This is due to the fact that existing OPN formalisms have concerned on the structural representation of object concepts, but not considered such aspects of processing the dynamic execution of object systems. In supporting the message passing between objects, most of OPNs can adequately represent the synchronous communication mechanism. However, they do not consider their behavioral semantics for the asynchronous communication mechanism.

Other than those mentioned above, OBJSA by Battiston[3], Petri net tools for object concurrency specification (PN-TOX) by Holvoet [13], and object coordination nets (OCoNs) by Giese [11] are proposed to enhance the modeling power of object-oriented systems. Although these proposals support the basic concepts of object such as encapsulation and modularization, none of them incorporates the concept of abstraction and inheritance thoroughly.

Unlike the existing OPN formalism, HOONet formalism can sufficiently support to represent the static and dynamic concerns of object-oriented software systems since our HOONet formalism provides the notion of abstraction hierarchy and inheritance hierarchy. Also HOONet is designed with more formal syntax and semantics than those of other OPN formalisms. Our proposed HOONet formalism can be used to model object-oriented system requirements in a natural and simple way.
7. Conclusion and further works

The maturity and popularity of object-oriented paradigms have steadily increased. One of the main requirements in modeling and analysis for complex and large software systems is that the design models should be unambiguous, precise, and verifiable. To fulfill these requirements, experts have suggested several methods which combine object-oriented method with formal methods. Although a number of high-level Petri nets with the concepts of objects were suggested with a clear idea in specific concerns, they did not fully support sufficient features that are needed in modeling of systems with object-oriented concepts.

To solve this problem, we suggest a high-level object-oriented Petri net HOONet, which supports most features of object-oriented concepts with clear semantics. Further, we describe the modeling and analysis methods for system models, and making it possible to develop a complex system incrementally and iteratively. This has been achieved from such bases as encapsulated and modularized objects, abstract information modeling, decomposition and refinement approach, and incremental reachability analysis.

HOONet can be used in various ways, but the following two are identified for its immediate application:

- A research issue in scenario-based software modeling is to solve the difference of abstraction levels for each scenario. When each scenario is analyzed and modeled with formal methods, and the abstraction level of each scenario differs, the integration of each formally represented scenario is problematic. In related studies on scenario-based modeling, this problem is solved with the assumption that each scenario is represented with the same abstraction level [8,27]. However, this assumption is unrealistic and is one of obstacles to be tackled for software modeling using scenarios. HOONet will be suitable for solving this problem because it is possible to represent such abstraction in a natural way.

- Reuse has been recognized as a key technology that can bring about significant productivity gains in software development. Reuse in software requirements is also another area where great benefits can be expected [14]. Nets modeled using HOONet formalism can be regarded as reusable components because each model is modularized and encapsulated with a formal interface. The key merit of reusing HOONet models is that additional mechanisms to support reuse are not needed.

In order to enhance the applicability of HOONet formalism to other domains, further studies should be performed to extend the HOONet. An extending issue to be cognized is the creation of a timed-HOONet formalism to represent the timing constraints in real-time system modeling. If we introduce the semantics of conventional time Petri nets to those of internal object net (ION) of HOONet, it is not difficult to model the timing behavior of a system.
However, when we consider the approach to modeling the timing constraints in object-oriented paradigm, this issue may not be solved by simple introduction of time Petri nets semantics. Another considerable issue is methodological consolidation between HOONet modeling method and UML-based object-oriented modeling paradigm. HOONet can represent the object structure and state transition of an object-oriented system. Thus, we consider that HOONet model can substitute the object diagram and state transition diagram in unified modeling language. However, the representation scheme for the relationship among objects is required in such substitution.

References


