An Operational Component Specification Method*

Hyung Ho Kim*, Dong Kwan Kim**, Hyo Taeg Jung**, Yun Dae Chung**, and Doo Hwan Bae*

* Software Engineering Laboratory
Korea Advanced Institute of Science and Technology
Taejon, Korea
{hhkim,bae}@salmosa.kaist.ac.kr

**Computer Software Technology Laboratory
Electronics and Telecommunications Research Institute
Taejon, Korea
{dgkim,htjung}@etri.re.kr, ydchung@computer.etri.re.kr

Abstract

We propose an operational component specification method that provides execution models for analyzing behaviors as well as adopts checking rules for consistency. Our proposed operational component specification method will help the users to capture the meaning of components in a precise and abstract manner. Our method is mainly based on two ideas, object-orientation and the action system. Object-orientation makes specifications easy to understand and the action system leads to a simple and clear view of behaviors. Our method utilize Petri net formalism, especially the notion of tokens, in order to handle dynamic configuration in each component. We develop an extended version of Petri nets, called Object Petri Nets, to build execution models and extend analysis techniques of Petri nets.

1. Introduction

Component-based development (CBD) [4, 2, 8] is an emerging discipline for promoting practical reuse of software. In CBD, by building new softwares with independently developed components we can gain the benefits promised by the software reuse such as quality improvement and rapid development. In order to facilitate the reuse of such components, there must be an effective way by which component developers communicate what services each component provides and how it works to its users. Therefore rigorous specifications for components are required for capturing the intention of component developers in a precise and abstract manner.

In rigorous specifications for components, the notion of consistency among various aspects of specifications must be carefully defined. In addition, we strongly believe that specification methods for components must be operational in the sense that specifications permit observations of their possible behaviors. In this paper, we propose a method for component specifications and develop a technique for building an execution model for each component specification. The main goal of our efforts is to provide a theoretical means for consistently describing structural and behavioral aspects of components and analyzing them.

Our approach has been inspired by two key ideas. The first is object-orientation. Object-oriented methods have been popular since they support useful modeling principles, such as "inheritance" and "association". In addition, it is natural to describe components with objects because most of component technologies such as COM[2] and Javabeans[8] are based on object-oriented concepts. In general, object-oriented methods provide various techniques for modeling diverse aspects of systems. Following this convention, we use mainly two object-oriented modeling techniques: one for the structural aspect and the other for the behavioral one. Each model makes it easy to focus on a particular aspect by suppressing details of the other. We adopt the classification principle, called the existence dependency [13] for integrity checking in both models: classifying

*This research is supported in part by ETRI(Electronics and Telecommunications Research Institute)
classes according to the existence dependency allows formulating a number of rules for semantics integrity checking. In fact, the two model we use are equivalent to those of [13] except they have operational semantics. Throughout this paper, we assume that a component consists of a set of objects and their relationships.

The second is the action system[1] that was proposed as a method for specifying distributed systems. Even though the message passing style dominates in object-oriented modeling, we prefer the action system style because it leads to a simple and clear view of behaviors as well as makes it easy to design the overall behavior among objects. Furthermore, the action system provides the framework for an execution model that can be seamlessly integrated with the existence dependency and its notion of consistency. In the action system, the overall behavior of objects is described in terms of the possible interactions among objects, called actions. In this paper, a sequence of actions with participating objects is called a computation and an execution model will be characterized by a set of possible computations. Each component statically exposes its functionality through a set of actions and dynamically shows its behavior by computations.

It should be noted that there are some efforts that have integrated object-orientation with the action system such as DisCo[7], and more recently, Catalysis[4]. What distinguishes our specification method from others is mainly the capability of handling dynamic configuration among objects and their relationships precisely.

To handle dynamic configuration, we have taken Petri nets[11] as a starting point, although finite state machine (FSM) based formalisms, such as Statecharts[5], are dominant as models for specifying behavior of objects. This is because the notion of dynamic configuration looks awkward in FSM based formalisms. In FSM, we face a dilemma such that the number of newly created objects can be infinite but it is impossible to encode an infinite number of objects with finite states. On the contrary, with Petri nets, we can get a significant benefit from the notion of tokens. It should be emphasized that our intention is not to give a rigorous evaluation of the two behavioral formalisms. Instead, we illustrate how the notion of tokens can be extended to handle dynamic configuration.

In order to build an execution model of a component specification, we design an extended version of Petri nets, called Object Petri Nets. Object Petri Nets, in computations, coherently handle the lifetime of objects and their relationships with respect to existence dependency. We also provide analysis techniques for exploiting behavioral properties of component specifications. For brevity we call Object Petri Nets just OPeN and a net of Object Petri Nets an OPeN-net. Our approach consists of the following two steps.

1. Specify a component with the proposed modeling techniques and check its consistency.

2. Analyze the behavior of the component specification using its automatically generated execution model

Note that OPeN is invisible to users in our approach. That is, users need not to deal with OPeN directly, rather they can specify and analyze components only using familiar notions such as object and action.

This paper is organized as follows. In Section 2 we explain the structural and behavioral models for component specification. In Section 3 we present the definition of OPeN and illustrate how to translate the models to OPeN nets. In Section 4, we show extended analysis techniques for OPeN. We briefly summarize related works and conclude this paper in Section 5 and Section 6, respectively.

2. Component Specification with Objects

In this section, we present modeling techniques for component specifications. Our approach is based on the notion of objects: a component is composed of a set of objects and the overall behavior of a component is determined by the behavior of individual objects and their relationships. To capture structural and behavioral aspects of each component, we build the two models called component structure model and component behavior model. In 2.1 and 2.2 we introduce component structure model and component behavior model, respectively. Then we explain how the two models characterize an execution model in 2.3.

2.1 Component Structure Model

A component structure model consists of a set of classes which are organized according to the existence dependency. The existence dependency plays a role of the classification principle "aggregation" in current object-oriented methods but provides simple and clear semantics. The existence dependency brings a number of rules for semantic integrity checking to us. For details, the reader is referred to [13].

Before introducing the component structure model, we summarize the definition of the existence dependency. The concept of the existence dependency is defined in terms of the lifetime of objects. The lifetime of an object begins at the point of its creation and ends
at the point of its deletion. The definition of the existence dependency is as follows: an object \( p \) is existence dependent of an object \( q \), called the parent, when 1) the lifetime of the object \( p \) is embedded in the lifetime of the object \( q \) and 2) the object \( p \) always refers to the object \( q \) within its lifetime. In this case we write \( p \prec q \) and call the object \( p \) the existence dependent object of the object \( q \). Similarly, this definition is extended at the class level such that a class \( P \) is existence dependent of a class \( Q \) if all objects of the class \( P \) is existence dependent of objects of the class \( Q \). Through this paper, the function \( PT \) associates each class with a set of its parent classes. If there is no parent of a class \( c \), the valuation of \( PT(c) \) is an empty set.

For example, let us consider a simple library component where a class \( COPY \), which captures individual instance of books, might be existence dependent of a class \( BOOK \), i.e., \( COPY \prec BOOK \). It means that the objects of the class \( COPY \) cannot exist without its parent, an object of the class \( BOOK \), and always refer it according to the definition of the existence dependency.

It should be noted that there are constraints deduced from the definition of the existence dependency. First, an object is never existence dependent of itself. Second, cyclic dependencies among existence dependencies are not allowed since they lead to circular prerequisites, which are impossible. Finally, an existence dependent object always refers just one object of each parent class as its parent, while a parent object does not have this limitation. Thus the cardinality of the existence dependency must be annotated to denote how many objects can be dependent on one parent object at one point in time. A natural number \( n \) specifies a range of cardinality from 0 to \( n \) and "*" stands for an arbitrary positive integer. The function card is used to map a pair of a parent class and its dependent class to their cardinality.

For graphical representation of the component structure model, we adopt the class diagram in UML notation, a de facto standard object-oriented modeling language[12]. Since the role of the existence dependency corresponds to that of aggregation, we select the notation of aggregation, a line with a hollow diamond, for the existence dependency. For example, Fig. 1 shows a component structural model for a simple library component.

In addition to the existence dependency, in Fig. 1, we use the "association" relationship. This relationship is just one kind of syntactic sugar: any association will be unfolded into an existence dependent class of its ends as parents. For more discussions the reader is referred to [13].

2.2. Component Behavior Model

A component behavior model consists of the action class table (ACT) \(^1\), which maps classes to actions and summarizes relevant information, a set of regular expressions that specify the behavioral constraints of classes, and a set of descriptions for each action. An ACT has a row for each action and a column for each class. A symbol in a cell indicates how the class participates in the action: a blank means that the class is not involved in the action. The symbols a ‘C’, an ‘M’, and an ‘E’ stand for the creation, modification, and end of objects, respectively. Each symbol is called the involvement type of an object in that action. To denote this relation, the function IT associates a pair of a class and a transition with each symbol. In a similar way, the function IC associates each action with a set of classes that involve in it. Fig. 2 illustrates an possible ACT for the simple library component shown in Fig. 1. Note that the ACT includes the association \( LOAN \) as a class. As said before, this is because associations are unfolded into an existence dependent object.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{act_example.png}
\caption{The ACT for the simple library component}
\end{figure}

\(^1\)This table is originally called Object Event Table (OET)[19]. Since, in this paper, the notion of action corresponds to that of event, we change its name to Action Class Table.
In general, each object is only allowed to involve in actions in a specific order during its lifetime. For example, before an object is deleted, it must be created. We annotate a regular expression to each class to describe its sequence constraints and assume that all objects follow the sequence constraints of their class. Regular expression fits well for describing the sequence constraints but any other techniques that are equivalent to regular expressions would do as well. In Fig. 3 the regular expressions describe the allowed sequence of actions for each class in the simple library component.

\[
\begin{align*}
\text{MEMBER} &= \text{enter}(\text{borrow + renew + return + lose})*.\text{leave} \\
\text{BOOK} &= \text{catalogue}(\text{acquire + borrow + renew + return + lose})*.\text{decatalogue} \\
\text{COPY} &= \text{acquire}(\text{borrow + renew + return})*.\text{lose} \\
\text{LOAN} &= \text{borrow.renew}*.\text{(return + lose)}
\end{align*}
\]

Figure 3. Regular expressions for classes in the simple library example

To capture the effects of actions, each action \(a\) is described in the form when \(g\) do \(S\) where the guard \(g\) is a boolean expression on the states of participating objects and the body \(S\) is a statement on their states. Informally, the guard of each action describes its enabling conditions and the body describes how the states of participating objects change. We denote the guard of an action \(a\) by \(\text{guard}(a)\) and the body by \(\text{body}(a)\). Fig. 4 shows a possible description of the action \(\text{borrow}\).

\[
\text{action \text{borrow}(m:MEMBER, b:BOOK, c:COPY, l:LOAN)}
\]
\[
\begin{align*}
\text{when } m.\#\text{ofLoan} <4 \\
\text{do } \\
&\text{m.}\#\text{ofLoan} += 1; \\
&c.\#\text{ofLoan} += 1;
\end{align*}
\]

Figure 4. Description for the action \(\text{borrow}\)

2.3. Informal Descriptions on Execution Model

When finishing a component specification, we check consistency with respect to the integrity checking rules[13]. Then, we generate an execution model for analysis.

In an execution model, each object decides whether it involves in common actions or not. When required objects agree to synchronize on an occurrence of an action and satisfy the guard of the action, it may occur and change the states of participating objects. In this way, an \(\text{ACT}\) and a set of regular expressions characterize a set of computation. The decision of involvement is dependent on the regular expression of an object and its relationships among other objects. Informally, conditions that allow an object \(o\) to involve in an action \(a\) are as follows.

- The regular expression of the object \(o\) can accept the action \(a\).
- All parent objects of the object \(o\) can participate in the action \(a\).
- When the object \(o\) is created in the action \(a\), objects of which the object \(o\) is existence dependent must involve in the action \(a\). Objects that will be parents must obey cardinality constraints.
- If the object \(o\) is deleted in the action \(a\), then there must be no existence dependent object of it.

It is obvious that these conditions prevent the occurrences of actions from the violation of the existence dependency. Formal treatments of execution models are the main issue in Section 3.

3. OPeN: Definitions and Translation

3.1. Why Not Finite State Machine, But Petri Nets

To build execution models we develop an extended version of Petri nets called OPeN. In this development, one of the important issues is to deal with dynamic configuration in each component. In order to make the problem clear, consider how a component consisting of objects works. Usually a component reacts continuously to incoming events. When an event arrives, what really happens is that, in the component, the objects responsible for this event collaborate to deal with it. In this collaboration, the action of each object depends on its local state and other objects with which it will collaborate. For instance, in the simple library component, the action \(\text{borrow}\) may occur if an appropriate object \(\text{copy}\) exists or, otherwise, this action may not. For this reason a technique that deals with dynamic configuration is necessary.

Unfortunately, FSM based formalisms lack the notion of dynamic configuration although they are usually adopted in current object-oriented methods. Recall the dilemma introduced in Section 1. This dilemma stems from the discrepancy between the finiteness of FSM and the infinitness in the possible number of objects. To resolve this discrepancy, there are mainly two approaches. The first is to assume the number of objects to be finite. For example, Objectcharts in [3] allows only static configuration of objects. We believe that
the limitation of this approach is rather severe since dynamic configuration among objects is inherent. In contrast, the second is to extend FSM to handle the infiniteness. For instance, object finite state machine (OFSM) changes its structure to reflect the creation of a new object over time [14]. However, in spite of its theoretical importance, OFSM was judged to be impractical from a performance standpoint. We believe that the change of structures causes the inefficiency.

When establishing an execution model based on Petri-net formalism, we gain a significant benefit from the notion of tokens: it is possible to encode dynamic configuration by the modification of a net’s tokens called marking, but not of its structure. To realize these ideas we define the structure and behavior of OPeN in the next subsection.

3.2. OPeN: Definitions

We develop the definitions of OPeN to handle dynamic configuration consistently with respect to the existence dependency, formally. We begin the definitions of OPeN with that of tokens. A token denotes an object and required information. To realize this idea, we define functions for tokens as follows.

**Def. 1** Let $C$ be a set of classes. The following functions are defined for a set of tokens $N$.

- **parents**: $N \rightarrow \text{set of } (C \times N)$ is a function that associates a set of parents and their classes pair with each token.

- **children**: $N \rightarrow \text{set of } (C \times N)$ is a function that associates a set of existence dependent objects and their classes pair with each token.

- **type**: $N \rightarrow C$ is a function that associates a class taken from $C$ with each token.

In the above function definitions, for convenience, each token is denoted by a natural number. Thus, a set of tokens $N$ corresponds to a set of natural numbers. These numbers called *object identity* are all distinct. Throughout the paper objects and tokens are used interchangeably. The functions parents and children capture the existence dependency of tokens. For instance, if a token $n$ is existence dependent of a token $m$ and their classes are $N$ and $M$, respectively, then the valuation of parents($n$) and children($m$) include ($Mm$) and ($Nn$), respectively. The functions parents and children play a key role in maintaining the constraints of the existence dependency. Informally, the function parents records the parent objects of each object, and the function children counts the existence dependent objects of each object and prevents an object from its deletion when there exists any existence dependent object of it. Finally, the function type denotes the class of each object.

The structure of an OPeN-net, similar to that of a P/T net, can be formally defined as follows:

**Def. 2** Let $C$, $A$, and $N$ be a set of classes, a set of actions, and a set of tokens, respectively. An OPeN-net is a 6-tuple $O_n = (P, T, F, \text{Cls, Act}, M_0)$, where

- $P$, $T$, and $F$ are a set of places, transitions and arcs, respectively, where definitions are the same as the ones for standard P/T nets.

- **Cls**: $P \rightarrow C$ is a class function that associates a class taken from $C$ with each place.

- **Act**: $T \rightarrow A$ is an action function that associates an action taken from $A$ with each transition.

- **M_0**: $P \rightarrow \text{set of } N$ is a function that associates a set of tokens with each place. This function is called the *initial marking*.

The functions Cls and Act map places and transitions to classes and actions, respectively. In this way, firing sequences of an OPeN-net denote the computations of a corresponding component. Similar to $M_0$, $M$ is used to denote a marking function, i.e., $M$: $P \rightarrow \text{list of } N$.

The enabling rule of OPeN captures the decision on the occurrence of actions. The enabling rule is defined as follows:

**Def. 3** A transition $t$ in $O_n$ is enabled with respect to $TKs$ in a marking $M$ if there exists $TKs$ such that for $\forall p_1, \ldots, p_n \in^* t$,

1. $\exists tk \in TKs \text{ s.t. } tk \in M(p_i)$ for $i = 1 \ldots n$ and $\forall cls \in IC(\text{Act}(t))$

2. guard$(\text{Act}(t))(TKs) = true$

3. if $\text{IT}(cls, t) = 'C'$ then $\forall c \in PT(cls) \exists tk \in TKs \text{ s.t. } type(tk) = c \land \text{CardinalityCondition}(tk, cls)$

4. if $\text{IT}(cls, t) = 'M'$ then $\exists tk \in TKs \text{ s.t. } type(tk) = cls \land \forall (c, tk') \in \text{parents}(tk) \exists tk'' \in TKs \text{ s.t. } tk' = tk''$

5. if $\text{IT}(cls, t) = 'E'$ then $\exists tk \in TKs \text{ s.t. } type(tk) = cls \land \text{children}(tk) = \emptyset \land \forall (c, tk') \in \text{parents}(tk)$

where $\text{CardinalityCondition}(tk, cls) = \text{card}(type(tk), cls) > \#(\{(c, n) \in \text{children}(tk) \land c = cls\})$
In the above definition, \( \tau \) represents a set of pre-places related to the transition \( t \). In a similar way, \( t^0 \) represents a set of post-places related to the transition \( t \). This definition formally declares the conditions that allow the occurrence of actions. Our definition assumes that transitions are related to appropriate places so that it is possible for tokens of certain classes specified in an ACT to involve in associated actions. In particular, a transition requires post-places for the involvement type ‘C’ and ‘M’ and pre-places for ‘M’ and ‘E’. An OPeN is said to be complete if it meets this assumption.

When a transition \( t \) is enabled with respect to \( TKs \) and fires with it, each token in \( TKs \) is removed from every place in \( \tau \). A new token set \( TKs' \) is constructed, and each token in \( TKs' \) is added to every place in \( \tau \). The firing rule formally describes how these steps proceed as follows:

**Def. 4** Let \( O_n \) be an OPeN. An enabled transition \( t \) with respect to \( TKs \) in a marking \( M \) yields a new Marking \( M' \) as follows:

\[
\forall p \in P, \\
M'(p) = \begin{cases} 
M(p) - tk, & \text{where} \\
& tk \in TKs \text{ s.t. } tk \in M(p) \text{ if } p \in (\tau - t^0) \\
& M(p) - tk + tk', & \text{where} \\
& tk \in TKs \text{ s.t. } tk \in M(p) \\
& \text{and } tk' \in TKs' \text{ s.t. } \text{type}(tk') = \text{Cls}(p) \text{ if } p \in (\tau - t^0) \\
& M(p) + tk', & \text{where} \\
& tk' \in TKs' \text{ s.t. } \text{type}(tk) = \text{Cls}(p) \text{ if } p \in (t^0 - t) \\
& M(p), & \text{otherwise} 
\end{cases}
\]

where \( TKs' \) is a set of tokens such that

\[
\forall c, n \in I(C(\text{Act}(t))), \\
\text{if } I(T(cls, t) = 'C') \text{ then} \\
\text{there is a token } tk' \text{ in } TKs' \text{ s.t.} \\
\text{children}(tk') = \emptyset \land \\
\text{parents}(tk') = \\
\{(c, n) | \forall c \in \text{PT}(cls) \land \exists n \in TKs \cdot \text{type}(n) = c\} \\
\text{if } I(T(cls, t) = 'M') \text{ then} \\
\text{there is a token } tk' \text{ in } TKs' \text{ as same as } tk \in TKs \text{ s.t. } \text{type}(tk) = cls, \text{except} \\
\text{tk' = body}(\text{Act}(t))(tk) \land \\
\text{children}(tk') = \text{children}(tk) \\
+ \{(c, n) | \forall c \in \text{I(Act}(t)) \land I(T(c, t) = 'C', \exists n \in TKs' \cdot \text{type}(n) = c\} \\
- \{(c, n) | \forall c \in \text{I(Act}(t)) \land I(T(c, t) = 'E', \exists n \in TKs' \cdot \text{type}(n) = c\}
\]

### 3.3. OPeN: Translation

An OPeN-net can be obtained by translating from component structure model and component behavior model. In fact, the structure of an OPeN-net can be constructed only from a set of regular expressions since they have sufficient information when both models are consistent.

Translations are performed in two steps. In the first step, we build partial OPeN-nets for each class. These OPeN-nets are partial in the sense that they do not meet the consistency assumption, that is, they are not complete. Partial OPeN-nets are constructed from the regular expressions of each class. Key operations in this step are to transform a regular expression to an equivalent FSM and to transform a FSM to an equivalent Petri net. Through these operations, a regular expression can be transformed to a partial OPeN-net since the structure of OPeN is similar to P/T nets. Then, mappings of the functions Cls and Act are assigned for this net: the function Cls associates the corresponding classes with every place. And the function Act associates the corresponding actions with each transition. Fig. 5 illustrates the regular expression of the class **COPY** and its equivalent partial OPeN in the simple library component.

**Figure 5.** The regular expression of the class **COPY** and its partial OPeN-net

In the next step, we compose a complete OPeN-net from the partial OPeN-nets. We define a composition operation that combines two nets, a net \( A \) and a net \( B \), to a new net \( O \). Informally, composition rules of the operation are as follows: first, the net \( O \) has all places in two net \( A \) and \( B \). Next, transitions and arcs are considered. If only one net includes the transitions of an action, the net \( O \) has these transitions and their corresponding arcs. When both nets \( A \) and \( B \) include the transition of a same action, the net \( O \) has transitions corresponding to every pair of these transitions and their arcs. For example, when a net \( A \) has two
transitions of an action \( a \) and a net \( B \) has three transitions of the same action, a net \( O \) includes six transitions of the action \( a \). Since the composition operation is associative, the order in which all partial OPeN-nets are combined to a complete OPeN-net does not matter. Fig. 6 shows a complete OPeN-net of the simple library component.

![Diagram](image)

**Figure 6.** The complete OPeN-net for the simple library component

4. Behavioral Analysis on Component Specifications

Using OPeN we can observe the possible behaviors of specifications and reason about their properties formally. The first analysis technique we apply is simulation. That is, we execute the specification and observe its behavior. Behaviors of specifications are captured in terms of computations.

Since a manual simulation is tedious and error-prone, we have implemented a concept-proving prototype tool for OPeN\(^2\): this tool, in operation, computes a set of enabled transitions with participating objects and provides it to user. When a user selects a transition, the tool fires it and repeats its operation until the user forces the tool to quit. As an example, Fig. 7 shows a part of a simulation session of the simple library component.

The second analysis technique the tool supports is a generation of a kind of a reachability graph. OPeN can handle an infinite number of objects and, thus, have infinite states. For this reason, it is impossible to generate an ordinary reachability graph such as that of CPN[9]. To solve this problem, we restrict the definition of OPeN such that, at a time, the number of objects of each class cannot exceed a fixed number defined initially. Although the reachability graph with this restriction cannot fully exploit the possible behavior of the specification, our experiences show that investigation on restricted reachability graphs provides some insight of the behavior of specifications since, in general, restricted reachability graphs shows similar behavioral patterns to unrestricted ones. In fact, due to a limitation of memory, all component systems have a limitation on the number of objects at a time.

5. Related Works

DisCo[7] is similar to our specification method. DisCo is the first approach that integrated object-orientation with the action system and has strongly inspired our specification method. What distinguishes our specification method from DisCo is to deal with dynamic configuration among objects and their relationships. DisCo only deals with a static topology of objects, and provides the well-established notions for data computations and the reasoning framework for them as well as modularity and their refinements. These features are helpful guidance for further researches in our specification method. More recently, Catalystis[4] provides a methodology for object and component specifications, in which object-orientation and the notion of action are integrated. In particular, Catalystis provides the informal but rigorous notion of modularity and refinement on them. However, specifications in Catalystis are not executable and lack the notion of concurrency.

The theory of the action system[1] is originally based on the finite state model while we have established the new model of the action system on Petri nets. This change of the underlying model of the action system brings the capability for handling dynamic configuration among objects and their relationships to our specification method. Fortunately, the change of the underlying model does not lead to the radical change of the action system. In fact, as written in [1], the action system based on finite state model can be considered a
special kind of Petri net.

Until recently, there are various efforts to introduce object-oriented concepts in the Petri nets approach such as LOOPN[10] and HOOKnets[6]. In general, these approaches model the behavior of each object in a separate Petri net and provide additional constructs for describing the relationships among the objects. The main thing that distinguishes our approach from these is the role of Petri nets. In our approach, OPeN is an execution model for specifications and not modeling techniques. That is, OPeN is invisible to the users. The users only use familiar notations such as class diagrams and regular expressions for specification.

6. Concluding Remarks

Component-based development (CBD) is popular for promoting reuse in practical ways. In CBD, the rigorous specification of components plays a significant role since development of components is independent of component assembly. We strongly believe that a specification method must support a kind of facilities for consistency checking and formal behavioral analysis for quality control.

This paper has presented the component specification method to address these issues. Our approach is based on mainly two ideas. One is object-orientation that provides intuitive and expressive modeling techniques. Especially, the adoption of the existence dependency brings a number of integrity checking rules. The other is the action system. The action system leads to a clear and simple operation model as well as seamless integration with the existence dependency.

We have designed an extended version of Petri nets called OPeN as an execution model of our specification method. With OPeN, we can analyze the behavior of a component specification formally. The novel feature of OPeN is to make the handling of dynamic configuration among objects and their relationships consistent with respect to the existence dependency.

While our research offers improvement in formality of component specification, there are some issues that are worthy of further research. In particular, a notion of structured derivation of specifications is needed to support specification at the large and complex scale. For this purpose, modularity and a notion of refinement relation are to be established in our specification method.

References