Behavioral Modeling Technique for Multiple Objects of Software Using Extended Place/Transition Nets with Attributed Tokens

Tomohiko Takagi
Faculty of Engineering and Design, Kagawa University
2217-20 Hayashi-cho, Takamatsu-shi, Kagawa 761-0396, Japan

Ryo Kurozumi
Graduate School of Engineering, Kagawa University
2217-20 Hayashi-cho, Takamatsu-shi, Kagawa 761-0396, Japan
E-mail: takagi@eng.kagawa-u.ac.jp, s19g457@stu.kagawa-u.ac.jp

Abstract
This paper shows EPNAT (Extended Place/transition Net with Attributed Tokens) and a modeling technique using it in order to address the problem of EPN. In software modeling using EPN, objects of which the software consists need to be defined individually as sub-EPN models, even if they have the same behavior. However, in software modeling using EPNAT, objects that have the same behavior can be integrated into one sub-EPNAT model, and therefore EPNAT models would be smaller than EPN models.

Keywords: software modeling, behavioral model, place/transition net, VDM

1. Introduction
EPN (Extended Place/transition Net)\(^1,2\) is a formal modeling language to represent the behavior of software that consists of multiple objects. The objects mean modules, components, or subsystems in this study, and they interact each other to provide expected functionality of the software. EPN enables engineers to construct unambiguous and executable software specifications that can be used for systematic skeleton code and test case generation. However, in software modeling using EPN, objects need to be defined individually even if they have the same behavior, which causes an increase in model size. A large model that includes redundant definitions often leads to additional cost for design, implementation, test, and maintenance, due to its poor readability.

This paper shows a novel language called EPNAT (EPN with Attributed Tokens) and a modeling technique using it in order to address this problem. In EPNAT, objects are expressed as attributed tokens that are classified into types, and also the states and events of objects of the same type are expressed as places and transitions, respectively. Attributed tokens can pass through places and transitions that belong to the same types as theirs. EPNAT models are converted to VDM++ specifications, and allow engineers to refine software specifications, create programs, test cases, and so on.

The rest of this paper is organized as follows. Section 2 shows software modeling using EPN and its problem. In section 3, we propose EPNAT and a modeling technique using it. In section 4, we discuss its effectiveness and future work.

2. Related Work

2.1. Software Modeling Using EPN
Petri net including PN has been used in traditional software modeling and testing.\(^3,4\) In previous study, we...
extended the PN by introducing VDM++ (a formal modeling language in VDM)\(^2\) in order to enhance its representation power, and generate test cases systematically. The extended PN, that is, EPN\(^{1,2}\) consists of the following four kinds of structural elements.

- Places to express states of each object
- Transitions to express events of each object
- Tokens to express a current state of each object
- Arcs to specify the flow of tokens between places and transitions

Details that cannot be represented in PN, such as the actions and pre-conditions (guards) of transitions, are formally written in VDM++. When a transition is fired, the values of variables that are defined for each object can be changed by its actions.

An EPN model consists of multiple sub-EPN models that represent the behavior of objects, and the sub-EPN models are connected by glue transitions that play an important role to specify the interaction among the objects. In an EPN model, the execution traces of software, that is, test cases are described as the sequences of successive markings (distribution of tokens on places), transitions, values of variables, and so on. An EPN model that gives engineers the overview of the behavior of software can be converted to a VDM++ specification that is used to refine the specification of the software.

### 2.2. Problem

In software modeling using EPN, objects need to be defined individually as sub-EPN models even if they have the same behavior, which causes an increase in model size. In general, a large model that includes redundant definitions often leads to additional cost for design, implementation, test, and maintenance, due to its poor readability.

We introduce an example given in Fig. 1 in order to discuss this problem. This simple load balancer (hereinafter, referred to as SLB) contains multiple servers and requests as objects. All server objects have the same behavior, and also all request objects have the same behavior. However, when the behavior of SLB is defined as an EPN model, each object is defined as a sub-EPN model that represents the behavior of each object. That is, if \(s\) and \(r\) are the maximum numbers of servers and requests that can be handled at the same time by SLB, an EPN model of SLB will contain \(s\) and \(r\) sub-EPN models for server and request objects, respectively.

<table>
<thead>
<tr>
<th>Tokens to express a current state of each object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arcs to specify the flow of tokens between places and transitions</td>
</tr>
</tbody>
</table>

This simple load balancer (hereinafter, referred to as SLB) can handle multiple servers and requests. Each server and request have uniquely identified in SLB. Also, each server and request have specific capacity and data size, respectively. It is assumed that the capacity is larger than the data size. A server uses the capacity of data size just during processing a request. A new server can be added to SLB at any time, and it can start working after maintenance. If there are requests on a queue that is described below, SLB allocates a server to the request that waits on the head of the queue. The server to be allocated must be in a state of working, and must have enough remaining capacity to process the request. If SLB finds multiple servers that can be allocated, it selects an arbitrary one. SLB can allocate a server to multiple requests if the server has enough remaining capacity. After a server has finished processing a request, SLB deallocates the server. A server that is not allocated can stop working and go to a state of maintenance. A server in a state of maintenance can be removed from SLB at any time.

SLB can receive new requests at any time. When SLB receives a request, the request is added to the tail of the queue. If SLB finds a server for the request on the head of the queue, the request is removed from the queue.

![Fig. 1. Software requirements of a simple load balancer.](image)

The objects that have the same behavior cannot be easily integrated into one sub-EPN model, since the variables for each object are defined as attributes of an EPN model or each sub-EPN model.

### 3. EPNAT and a Modeling Technique

In this section, we propose EPNAT and a modeling technique using it in order to address the above-mentioned problem.

#### 3.1. EPNAT

EPNAT is the extension of EPN, and it includes all the structural elements of EPN. The most obvious difference is that tokens in EPNAT correspond to objects that are classified into types, and they have variables for the objects. In this paper, tokens that have variables for objects are called attributed tokens. For example, in SLB of Fig. 1, the server objects (that is, objects of server type) need to have variables such as ID and capacity, and thus attributed tokens of server type (that is, attributed tokens that correspond to the server objects) have those variables. Attributed tokens of the same type have the same variables but different values. For example, the attributed tokens \(a_{k_e}\) and \(a_{k_b}\) that correspond to the server objects \(a\) and \(b\) respectively.
and \( b \) in SLB have different values, such as \( a_{\text{in}}.\text{ID}=1 \) and \( a_{\text{out}}.\text{ID}=2 \), respectively.

Places and transitions that express the states and events of objects respectively are also classified into types, and they are shared by attributed tokens of the same type in EPNAT. Attributed tokens can pass through places and transitions that belong to the same types as theirs. For example, the attributed tokens of server type can pass through the places and transitions of server type. Note that glue transitions belong to multiple types.

Transitions can have actions for attributed tokens. When a transition is fired, the variables of attributed tokens that pass through it can be changed by its actions. Transitions can also have pre-conditions and post-conditions that need to be satisfied just before and after the fire of the transitions, respectively. Places can have invariants, that is, a condition that needs to be always satisfied. The variables of attributed tokens can be referred in pre-conditions, post-conditions, and invariants.

All structural elements of each sub-EPNAT model belong to the same type, and each sub-EPNAT model represents the behavior of all the objects of the type. For example, one sub-EPNAT model of server type covers all the objects (an arbitrary number of the objects) of server type. All sub-EPNAT models are connected by glue transitions in an EPNAT model that represents the behavior of software.

### 3.2. Example

Fig. 2 (i) shows the overview of an EPNAT model of SLB that is constructed based on Fig. 1. It consists of two sub-EPNAT models that represent the behavior of objects of server type and request type. There are two places and six transitions in server type, and two places and three transitions in request type. The transitions "allocate" and "deallocate" are glue transitions, and the two sub-EPNAT models are connected by them.

The labels "seq" and "set" indicate the collection type of places, that is, whether the order of arrival of attributed tokens on places should be kept or not. Either "seq" or "set" should be specified for each place by engineers. In VDM++ specifications that is discussed later, places with the labels "seq" and "set" are implemented as instance variables of sequence type and set type, respectively. The label "inv" means that a place has an invariant. Details of "inv" are written in VDM++ by engineers, but they are omitted in the overview of an EPNAT model for ease of readability.

The label "act" means that actions are given to a transition. In VDM++ specifications, transitions including their actions, incoming and outgoing arcs are implemented as operations. The labels "pre" and "post" mean that a transition has a pre-condition and post-condition respectively, but the latter does not appear in Fig. 2 (i). Details of the labels "act", "pre" and "post" are written in VDM++ by engineers, but they are omitted in the overview of an EPNAT model for ease of readability.

Attributed tokens of server type and request type have two and three variables, respectively. The term "nat" means natural number type in VDM++. In Fig. 2 (i), there are six attributed tokens. Note that it is a snapshot and there should be no attributed tokens in the initial state of the model.
SLB. In VDM++ specifications, attributed tokens are implemented as record type.

Fig. 2 (ii) shows the details of the glue transition "allocate" written in VDM++. The statement (a) specifies the way of selection of attributed tokens to be moved. The statements (b) and (d) specify the way of deletion and addition of attributed tokens on the related places, respectively. The statements (c) specify the actions to be executed on the transition "allocate", and they correspond to the label "act" that is pointed by (α) in Fig. 2 (i). The statement (e) specifies the pre-condition that needs to be satisfied just before the fire of the transition "allocate", and it corresponds to the label "pre" that is pointed by (β) in Fig. 2 (i). The codes that are pointed by (γ) in Fig. 2 (ii) need to be considered and written based on software requirements by engineers.

In this paper, the details of the other structural elements in Fig. 2 (i) are omitted because of limitations of space. Many VDM++ codes for them can be semi-automatically generated by our coding pattern that is discussed in section 3.3.

3.3. Modeling Technique

The modeling technique using EPNAT consists of the following four steps. Note that these steps are not always separated clearly. If engineers find any problems in a current step, they can return to a previous step. Also, engineers can incrementally and iteratively proceed with these steps. The product that has been constructed in each step is executable, and thus should be tested in the end of each step.

**Step 1: Construction of Sub-PN Models**

Objects are identified and classified into types, and then the abstracted behavior of objects of the same type is defined as sub-PN models, as shown in Fig. 3.

**Step 2: Integration of Sub-PN Models**

All the sub-PN models are connected by glue transitions in order to complete a PN model that represents the abstracted behavior of software.

**Step 3: Addition of Details to a PN Model**

The following definitions, which are written in VDM++, are added to the PN model in order to complete an EPNAT model.

- Variables of attributed tokens
- Collection type of places ("seq" or "set")
- Invariants for places
- Actions to be executed on transitions
- Pre-conditions and post-conditions for transitions
- Conditions to select attributed tokens to be moved on transitions

**Step 4: Conversion to a VDM++ Specification**

The EPNAT model is converted to a VDM++ specification. Fig. 4 shows our coding pattern to perform this conversion systematically. The codes shown in the form of \(<\text{terms}>\) are extracted from the EPNAT model, or are written by engineers.

The VDM++ specification is used to refine software specifications, create programs, test cases, and so on.

4. Discussion and Future Work

In the previous section, we proposed EPNAT and a modeling technique using it in order to address the problem of EPN. In software modeling using EPN, objects need to be defined individually as sub-EPN models even if they are the same type, which causes an increase in model size. However, in software modeling using EPNAT, the objects of the same type can be integrated into one sub-EPNAT model, and therefore EPNAT models would be smaller than EPN models. We discussed the problem and effectiveness by using an example of SLB.

The coding pattern proposed in this paper will not be suitable for extremely large and complex software requirements, since they are defined as one class in a VDM++ specification. For example, each sub-EPNAT model may be defined as a class in order to address this problem, which will be discussed in our future study.
Our technique should be supported by a tool, since engineers will need to spend a certain amount of time and effort to construct EPNAT models and VDM++ specifications. In our previous study, a prototype tool was developed to edit EPN models and VDM++ specifications, and automatically convert between them. The new functions to edit EPNAT models and convert them to VDM++ specifications are being added to the prototype tool. We will apply it to non-trivial software requirements to evaluate the effectiveness of our technique further.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Number JP17K00103.

References