Evaluating Behavioral Correctness of a Set of UML Models

Yoshiyuki Shinkawa
Department of Media Informatics, Faculties of Science and Technology, Ryukoku University
1-5 Seta Oe-cho Yokotani, Otsu, Shiga, Japan

Keywords: UML, Model Correctness, Colored Petri Nets, Model Driven Development.

Abstract: In model driven software development, the correctness of models is one of the most important issues to construct high quality software in high productivity. Numerous research has been done to verify the correctness of those models. Conventional research mainly focuses on individual models, or at most the relationships between two individual models. However, the models must be correct as a whole set. This paper presents a Color Petri Net (CPN) based formal approach to verifying the behavioral correctness of UML models depicted by three different kinds of diagrams, namely state machine, activity, and sequence diagrams. This approach defines the correctness of a set of models from three different perspectives. The first perspective is the completeness that assures the syntactical correctness of the set. The second is the consistency that claims no conflicts between heterogeneous UML models. And the last is the soundness that represents the internal correctness of each model in the set.

1 INTRODUCTION

In model driven software development using UML, we need a set of sufficiently refined models expressed in the form of UML diagrams before starting implementation phase of a development project. One of the difficulties in preparing the above set is that we have to arrange the models represented by the different forms of diagrams, since UML provides us with thirteen kinds of diagrams to create the models from various viewpoints. Even though this variety of notations increases the expression capability of UML, it also increases the vagueness of the models, and these models might conflict with each other, which would make the UML models incorrect.

The incorrectness of the UML models often causes various problems in the implementation of the models, such as malfunctions, inconsistent data, or infinite loops. Therefore, it seems to be one of the most critical challenges to detect and resolve the conflicts in UML models in order to prepare the correct set of UML models which can be implemented without problems. However there are no definite criteria to evaluate whether the above set is correct for implementation.

Numerous research has been done to evaluate the correctness of UML models using various formalization tools, e.g. logic (Amalio and Polack, 2003) (Lausdahl et al., 2009), Petri nets (Garrido and Gea, 2002)(Shinkawa, 2006), algebraic specification (Favre and Clerici, 1999), process algebra (Fischer et al., 2001), and model checking (Knapp and Wuttke, 2006) (Shinkawa, 2008). The above research mainly focuses on individual models, or at most the relationships between two individual models. However, for practical use, the correctness of the whole set of UML models that will be implemented must be proved to be correct.

This paper presents a Colored Petri Net (CPN) based formal approach to evaluating the correctness of a set of UML models, each of which is expressed in the form of state machine, activity, or sequence diagrams. The reason we choose these three kinds of UML diagrams is that they are most frequently used to express the behavior of software, and moreover they are tightly interrelated.

The paper is organized as follows. In section 2, we discuss the relationships between the above three kinds of diagrams in order to define the correctness of a set of UML models. Section 3 introduces a CPN based formalization of these models for rigorous evaluation of the correctness. Section 4 presents how the formalized UML models are evaluated from several viewpoints.

Shinkawa Y..
Evaluating Behavioral Correctness of a Set of UML Models.
DOI: 10.5220/0004082202470254
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 247-254
ISBN: 978-989-8565-19-8

2 CORRECTNESS OF UML DIAGRAMS

When we develop software through Model Driven Architecture (MDA) (Warmer et al., 2003), many kinds of "models" are created in each phase of development, e.g., "requirement analysis", "external design", "internal design", and "implementation" phases. These models are usually expressed in the form of UML diagrams. Since UML 2.x provides us with thirteen different kinds of diagrams, we have to deal with a variety of model forms in MDA-based development.

Before discussing the correctness of a set of UML models, we first need to reveal the relationships between these diagrams.

2.1 The Relationships between UML Diagrams

Among the above thirteen diagrams, "state machine", "activity", and "sequence" diagrams compose one of the most important set of UML diagrams to represent the behavioral aspect of a system, since these diagrams respectively depict the behavior of objects in different ways, that is, as the behavior of individual objects, as the flow of process performed by these objects, and as the interactions between these objects. In order to construct a system appropriately that reflects a given set of UML models expressed in the form of these three kinds of diagrams, there must be no conflicts not only within each model, but also between the models.

However, few criteria are defined for evaluating the conflicts between UML models. Therefore, we first have to define the "relationships" between UML models. Supposing we are given a set of UML models depicted by the above three kinds of diagrams, which represents the behavioral aspect of a system, it must include all the objects composing the system. These objects show the different forms according to in which diagram they appear. Therefore, the relationships between arbitrary two models expressed in the form of different diagrams cannot be simply identified by comparing them. In order to identify the relationships between the above three kinds of models, we first reveal how an object acts in each diagram.

In a state machine diagram, a diagram itself represents an object, and the behavior of an single object is depicted as state transitions. On the other hand, an activity and sequence diagrams usually include multiple objects which behave independently. In a sequence diagram, an object occurs as a "lifeline", while no explicit object occurs in an activity diagram. An object occurs in two possible forms in an activity diagram, that is, as an object flow between activities, or as an actor of an activity.

As shown above, even though an object is a common element between these three diagrams, its behavior is differently expressed within them. Therefore, we need more common measures that exist within all of them. One of the convenient measures is a method invocation, since it occurs explicitly within these diagrams. In a state machine diagram, it is associated with a state transition, which is referred to as an "action" of the transition. It also occurs in an activity diagram as an "activity" or "action", or as a "message" exchanged between lifelines in a sequence diagram. By using the method invocation as a common measure, the behavior of an object in the three different diagrams is represented commonly as an order of method invocations.

In order to define the relationships between these diagram types using the method invocations, several assumptions are taken into account in this paper, which seem acceptable in software development. Firstly, the state of each object is determined by a set of values assigned to the instance variables of the object. Secondly, the above instance variables can be updated only by the methods of the object to which the variables belong. This means all the instance variables are fully encapsulated. Thirdly, each lifeline, activity, and action mentioned above is definitely associated with an object.

These assumptions suggest that each time a message is received by a lifeline, or an activity/action is performed in a sequence or activity diagram respectively, the associated object makes a state transition in the way as specified in the state machine model for the object. These state transitions divide each lifeline of a sequence diagram into the zones associated with the states. On the other hand, there seems to be no way to divide an activity diagram by the states, since unlike a sequence diagram, an activity diagram is not partitioned by each object unless we use "swim lanes". Therefore, in order to divide an activity diagram into the zones like a sequence diagram, it is a good practice to reorganize an activity diagram by aligning activities and actions along the objects that perform them. In doing so, we can divide an activity diagram into the zones like a sequence diagram. Since the state for each zone is taken from a state machine diagram, we refer to the states in an activity or sequence diagram as the "injected state", and the process to identify the state as the "state injection".

Assuming all the activity and sequence diagrams included in the currently focused set are divided into such zones, the above three kinds of UML models, that is, the models depicted by UML state machine,

activity, or sequence diagrams, are to have another common measure "state" to evaluate the correctness.

2.2 Three Perspectives on Correctness

In order to evaluate the correctness of a set of UML models using the above two measures, namely "method invocation" and "zone of the state", some restrictions are imposed on the UML models composing the set to be evaluated.

First of all, any method name that occurs in an activity or sequence model[1] must occur in the corresponding state machine model with the same signature. Here the corresponding state machine model means the state machine model that represents the object including the method. In addition, each activity/action in an activity model, and each lifeline in a sequence model, must be uniquely associated with an object.

On the contrary, some method names that occur in a state machine model may not occur in any activity or sequence models in the set of UML models, since we assume each state machine model represents all the possible state transitions by method invocations, while the object within an activity or sequence model only show a part of these transitions.

When a set of UML models satisfies the above conditions, all the activities/actions in the activity diagrams, and all the message exchanges in the sequence models, can be fully related to the state transition in the corresponding state machine model through method invocations. This means we can rigorously evaluate the correctness of the set. We refer to such a model that satisfies the above conditions as a "**complete**" model, and such the property of a set of UML models as "**completeness**".

Assuming a set of UML models is complete, the behavior of each model can be interrelated through a series of method invocations. As mentioned above, a state machine model represents the whole behavior of the object, while an object in an activity or sequence model may show a part of the behavior. Therefore, the behavior of an object in an activity or sequence model must be a part of the behavior of the corresponding state machine model. The behavior of an object is defined as a series of method invocations in this paper, therefore there is another constraint that must be satisfied by a set of UML models.

This constraint requires that any series of method invocation in an activity or sequence model must occur in the corresponding state machine model when execute the models. We refer to a set of UML models that satisfies this constraint as a "**consistent**" set, and the property of a set of UML models as "**consistency**".

The above two constraints for the model correctness mainly focus on the mutual integrity between the models, however few concerns are taken about the internal correctness within each model. As for this type of correctness, UML provides us with a few model components to give the constraints. One is the *state invariant* in a sequence diagram, and the other is a set of the *pre-* and *post-conditions* in an activity diagram[2].

Both constraints require a model to have the designated state at the specified point. If the set of UML model is complete and consistent, each activity or sequence model can be divided into the zones associated with the injected states, and therefore we can evaluate the validity of the given state invariants or pre- and post-conditions. For the validation of state invariants or pre- and post-conditions against the injected states, these states must be expressed in the common formal way. One of the appropriate ways is to express them in the form of predicate logic formulae which include the instance variables.

Assuming the predicate logic formula for a pre- or post-condition, or state invariant is $P(\vec{x})$, and the injected state of the zone where the $P(\vec{x})$ is located be $Q(\vec{x'})$, $P(\vec{x})$ must hold under the condition $Q(\vec{x'})$, that is, $Q(\vec{x'}) \rightarrow P(\vec{x})$ must hold. Here, $\vec{x}$ and $\vec{x'}$ represent the arguments of the predicates or functions included in the formulae P and Q, which consist of the tuples of instant variables. We refer to a set of UML models as "**sound**" if all the state invariants, pre-conditions, and post-conditions in the set satisfy the above constraint, and such a property of UML models as "**soundness**".

The above three perspectives, namely completeness, consistency and soundness assure the correctness of a set of UML models from individual model level to a whole system level. However, the UML models within the set are specified by three kinds of diagrams, and difficult to examine the correctness. In addition, UML itself is comparably vague for rigorous correctness evaluation. Therefore we need more formal and common notation for the UML models in the set. In the next section, we discuss Colored Petri Nets (CPN) as a common notation and evaluation tool.

[1] A model represented in the form of an state machine diagram is referred to as a state machine model. An activity and sequence models are interpreted as well.

[2] Local pre- and local post-conditions are also provided for an action.

3 CPN MODELING FOR UML

Colored Petri Net (CPN) is an extension of a regular Petri net, which introduces functionality and data types for more flexible control of the regular Petri net (Jensen and Kristensen, 2009). CPN is formally defined as a nine-tuple *CPN=(P, T, A, Σ, V, C, G, E, I)* , where

P : a finite set of places.
T : a finite set of transitions.
(a transition represents an event)
A : a finite set of arcs $P \cap T = P \cap A = T \cap A = \emptyset$.
Σ : a finite set of non-empty color sets.
(a color represents a data type)
V : a finite set of typed variables.
C : a color function $P \rightarrow \Sigma$.
G : a guard function $T \rightarrow$ expression.
(a guard controls the execution of a transition)
E : an arc expression function $A \rightarrow$ expression.
I : an initialization function : $P \rightarrow$ closed expression.

In this section, we briefly introduce how the above three kinds of UML models are expressed in the form of CPN.

3.1 Transforming State Machine Models into CPN

Both the state machine models and CPN models have the similar structures to finite automata with variable (Skoldstam et al., 2007). Therefore, state machine models are transformed into CPN by relatively simple rules. The structural relationships between these two models are as follows.

1. Each state in a state machine model corresponds to a place in a CPN model. The color of the place is composed of the data types that determine the state.
2. A state transition corresponds to a transition located between two places that reflect the source and destination states of the state machine

Other complicated control structures, such as a *composite* state, or *pseudo* states like *fork/join*, *choice/junction*, and *history*, can be transformed into CPN (Shinkawa, 2011). For example, junction pseudo state is expressed in the form of CPN as shown in Figure 1. In addition to the structural transformation, model semantics must also be transformed. The semantics of a state machine model can be defined by the two contrastive viewpoints, namely "static" and "behavioral". The former semantics represents "what each state means", and expressed in the form of predicate logic formulae as discussed above. On the other hand, the latter semantics represents "how the model

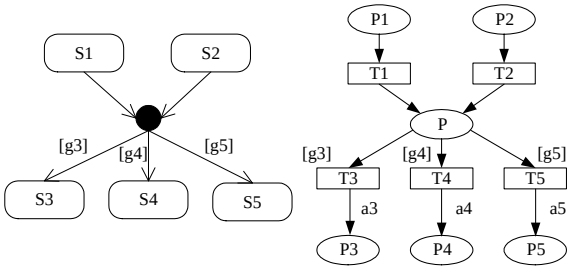

Figure 1: Junction pseudo state.

makes state transitions", which is controlled by the *guards* on the transition arcs and pseudo states. These guards can also be expressed in the form of the predicate logic formulae. In CPN, these logic formulae are expressed as CPN/ML functions with boolean result values.

3.2 Transforming Activity Models into CPN

Both an activity model and a CPN model express the progress of a process, and they have similar structures (Eshuis and Wieringa, 2003). While a CPN model is structurally composed of only four graphical elements, an activity model consists of many graphical elements to express complicated control flows. In spite of the different appearances, the structure of an activity model can be transformed into CPN as follows.

1. Replace each activity and action by a transition with an output place from it.
2. Replace each initial and final node by a place.
3. Replace a *fork node*, which split a single flow of control into parallel flows, by a transition with multiple output places to initiate the parallel flows. And replace a *join node*, which merge parallel flows of control into a single flow, by a transition with multiple input places.
4. Replace a *decision node* by a place with multiple output transitions, each of which has the guard for conditional branch, and replace *merge node* by a place with multiple input transitions.
5. Replace each *accept event action*, which deals with an asynchronous event, by a transition with an input place, and replace each *send signal action* , which generates a message or signal, by a transition with an output place[3].
6. Replace an *event handler* in the same way as the

[3]The occurrence of an asynchronous event is controlled by a token that is marked in the input place.

accept event action. While an *accept event action* deals with an asynchronous event that occurs outside the model, an *event handler* deals with an exceptional event that occurs in the model. However, they can be expressed in the same way in a CPN model.

7. Replace an *iteration expansion region* by a transition with an input place to which the transition feedbacks the token, replace a *parallel expansion region* by the same CPN structure as the *fork* and *join node*, and replace a *stream expansion region* by transition with output place.

In addition to the above structural transformation rules, semantic transformation rules must be defined as we did for state machine models. The semantics of an activity model is described as a method associated with each activity/action along with a *guard* on each flow or decision node. These semantic elements can be transformed into CPN/ML functions in the same way as state machine models.

3.3 Transforming Sequence Models into CPN

Unlike a state machine or activity model, the appearance of a sequence model is quite different from that of a CPN model. However, since the behavior of a sequence model is defined as a series of message passing, which can be interpreted as a series of method invocations, the model shows the similar behavioral characteristics to a state machine or activity model. Therefore, a sequence model can be transformed into the behaviorally equivalent CPN model in the similar way to the transformation of a state machine or an activity model as mentioned above. The transformation rules are summarized as follows (Shinkawa, 2011).

1. Replace each *receiving event occurrence* by a transition with an output place.
2. Replace an *alt*, *opt*, *break*, and *loop* combined fragment by two conflicting transition with appropriate *guards* to control the branch operation.
3. For a *critical* combined fragment, add a special place that holds a “lock” token, and draw bidirectional arcs to the transitions that conflict with the fragment.
4. Replace a *par* fragment by a transition with multiple output places for concurrent operations.
5. Replace a *seq* fragment in the same way as the *par* fragment, then add the arcs to control the required sequences[4]from the output place of a preceding transition to the succeeding transition.

In addition to the above structural transformation rules, the semantic transformation rules are needed. These rules are based on method invocations and *guards* in such combined fragments as *alt*, *opt*, and *loop*, and can be transformed in the similar way to activity models.

3.4 Zoning Activity and Sequence Models

In order to discuss the soundness of activity and sequence models, we need to divide the CPN models transformed from them into zones based on states so that we can identify the state at any point of the models. Since the state of each object is updated by a method invocation of the object, and each method corresponds to a transition with an output place, the state is regarded to be expressed as the token value marked in the output place. Therefore, each output place of the transition that corresponds to a method can be classified based on the states. On the other hand, the state of the transitions can be defined to be the same as that of the input place, since the state is updated after the firing of the transitions.

The discussion so far suggests that a transition with an input place and the connecting arc compose a smallest unit of a state in a transformed CPN model. However, there could exist the transitions that do not correspond to methods, but represent some control structures such as *if - then -else*, *loop*, *break*, and so on. These transitions do not affect the state of a system, and therefore the states of these transitions, associated places, and arcs are taken over from the states of the preceding units of states.

The transformed CPN models are divided into zones based on the above concept. The concrete procedure to define the states is discussed in the next section.

4 EVALUATING THE MODEL CORRECTNESS

The correctness of a set of UML models, which includes state machine, activity, and sequence models, has three different perspectives as mentioned in section 2.

The first perspective is the *completeness* of the set. This perspective means that each method name within the activity models or sequence models occurs within

[4]The required sequences by the *seq* fragment are slightly confusing and tricky. The detailed description is found in (OMG, 2011).

the corresponding state machine model. In the activity models, the method invocations are associated with the activities or actions, while in the sequence models, they are associated with message passing.

The second perspective is the *consistency* of the set, which means the activity and sequence models behave consistently with the corresponding state machine models. The behavior of the models is defined based on the order of method invocations.

The last perspective is the *soundness* of the set, which means all the activity and sequence models in the set satisfy the state based constraints designated by pre- and post-conditions, or by state invariants. The states in these models are injected from the corresponding state machines.

In this section, we discuss how the above three perspectives of the correctness are verified in the transformed CPN models.

4.1 Verification of Model Completeness

When transforming state machine models, activity models, and sequence models into CPN models, all the methods are implemented in the form of CPN/ML functions. Since all the methods that occur in the activity or sequence models must occur in the corresponding state machines, one of the simplest ways to preserve the completeness is to transform and compose the CPN models from all the state machine models before transforming other model types. And in subsequent transformation for activity and sequence models, method implementation by CPN/ML is restricted to only referring the functions already defined in the state machine models. By avoiding to create new CPN/ML functions in the transformation of activity and sequence models, all the method names and their arguments are controlled by the state machine models. As a result, the completeness is assured by CPN/ML compiler.

4.2 Verification of Model Consistency

In order to evaluate the consistency of a set of UML models, we first need to identify the order or sequence of method invocations for each object in the models. For this purpose, we append trace facility to each transformed CPN model. This facility is composed of a place to hold the history of method invocations, arcs to/from all the transitions, and a transition to examine the above history. In addition, several data types are introduced as color sets to express the history, namely "colset Object = int" to distinguish each object, "colset Method = int" to distinguish each method in a object, "colset MethodList = list Method", and "colset Trace = product Object * MethodList".

The history of method invocations is expressed as a token with the color "Trace", and is created for each object. The place to hold the history is labeled "TraceHolder-*m*", where $m(=1,2,\cdots)$ represents the CPN model id, and the place is associated with the color "Trace". The arc function of the arc to the "TraceHolder-*m*" is defined as follows.

1. For a transition on a single process path that represents a method, add the method id to "MethodList" part of the "Trace" token of the object.
2. For a transition to split a single process path into *n* parallel process paths, terminate the current "Trace" token by adding a unique negative integer at the end of the "MethodList" part. Subsequently, create *n* new "Trace" tokens for the *n* parallel paths, of which "MethodList" part begin with the above unique negative integer followed by a path id $p\ (=1,2,\cdots,n)$.
3. For a transition on the *i*-th path of the *n* parallel paths, use the "Trace" token of which "ModuleList" part has "*i*" as the second element.
4. For a transition to merge *n* parallel paths into a single path, terminate *n* "Trace" tokens by adding the same negative integer as the first element at the end of "ModuleList" part. The succeeding transitions use the same "Trace" token before the most recent transition to split.

Each CPN model is modified as shown in Figure 2. By the above modification of CPN models, each CPN model yields a set of "Trace" tokens in its own "TraceHolder-*m*". The consistency evaluation is to be made between a state machine model and activity or sequence model, focusing on a particular object. Since we assume a state machine model is built for each object, and several activity and sequence models are built including these objects, the consistency evaluation should be made object by object. This evaluation can be made by the guard function of a transition with two input places, one is the "TraceHolder-*m*" of a state machine, and the other is that of an activity or sequence model. The guard function examines a pair of "Trace" tokens from the state machine model and an activity or sequence model. Since each "Trace" token includes an object id, we can extract the necessary tokens from the "TraceHolder-*m*" of the activity or sequence model to evaluate whether they satisfy the consistency criteria. The detailed evaluation steps are as follow.

1. Let *P* and *Q* be CPN models which are transformed from an activity or sequence model, and from a state machine model respectively.

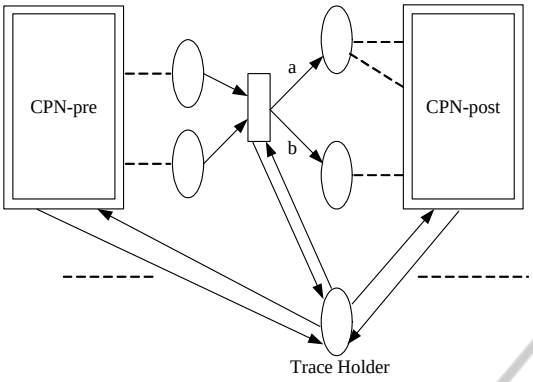

Figure 2: Trace holder place.

2. Let trace(P) and trace(Q) be a pair of "Trace" tokens for the same object, both "ModuleList" part of which have the same integer as the first element.
3. The model P and Q are consistent if the "ModuleList" part of trace(P) includes that of trace(Q) as a substring.

The verification of the consistency between two CPN models, one is from an activity or sequence model, and the other is from a state machine model, is performed as shown in Figure 3. In this figure, the gurd g of the transition e examines the consistency between P and Q according to the above algorism. The guard g returns true if P and Q are NOT consistent. If the inconsistency is detected, the transition e removes the "Trace" token from the "TraceHolder-m", and consequently the succeeding transitions in P and Q halt. The transition e provides the error message token to the place "Inconsistent".

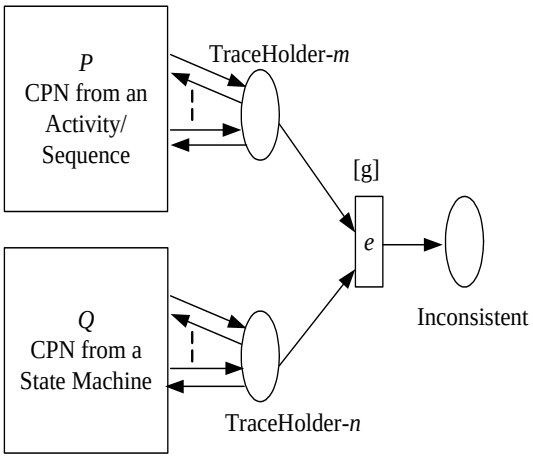

Figure 3: Consistency verification.

4.3 Verification of Model Soundness

For model soundness verification, we need the injected states at the locations where *pre-* and *post-conditions* or *state invariants* are specified. The injected state at an output place of a transition t is determined as the state in the corresponding state machine model, to which there is a state transition associated with the same method invocation as the above transition t. However, the same method invocation may occur at several states in the state machine model. Therefore we need to find the most adequate correspondence of the method invocation between the state machine and the activity or sequence models. This correspondence is identified by examining the same series of method invocations in both of them.

This examination can be done in the similar way to the consistency verification. The detailed process is shown as follows.

1. Let the place in the activity or sequence model be p, to which the state is to be injected, and the transition and the associated method be t and m respectively.
2. Let the places in the corresponding state machine model be $q_1, q_2, \cdots, q_n$, to which the transitions$_1, s_2, \cdots, s_n$ that represents the method m supply tokens.
3. Prepare two places c_1 and c_2 for controlling the state injection, and draw an outgoing arc from t and s_1 respectively.
4. Prepare a transition e_1 which has a pair of incoming/outgoing arcs from/to the "TraceHolder-m" of each model, and draw incoming arcs from c_1 and c_2.
5. Set the guard g of the transition e_1 in the same way as the consistency verification.
6. Prepare a transition e_2 which has the same structure as e_1, and set the guard functioning as the negation of g, namely $\neg g$.
7. Repeat the steps 3 to 6 for s_2 tp s_n.

The above process could find multiple identical series of method invocations in the state machine model. In such case, we cannot identify the state of the place currently focused on, but can obtain a set of possible states. Therefore, the verification of the model soundness should be examined for all the possible states. The verification for each state is made as follows.

1. For the place p associated with a state invariant S, which is expressed in the form of a predicate logic formula, put a new transition e that refers to the token of p. The guard g of the transition e

implements the formula "$\neg(T \rightarrow S)$", where T is the injected state of p.

2. For the place q_1 and q_2 associated with a pre- and post-conditions respectively, put new transition e_1 ande_2 that refer to the token e_1 and e_2. The guards g_1 and g_2 of e_1 and e_2 implement "$\neg(T_1 \rightarrow S_1)$" and "$\neg(T_2 \rightarrow S_2)$, where T_1 and T_2 are the injected states of q_1 and q_2.

The above CPN structure halts the CPN model execution if a soundness violation is detected in the same way as the consistency verification, which is shown in Figure 3.

5 CONCLUSIONS

A Colored Petri Net (CPN) based evaluation process for the correctness of a set of UML models is presented in this paper. While the previous research mainly focused on the correctness of individual UML models, or the relationships between two individual models, this process deals with a set of UML models described by three different kinds of diagrams, namely a state machine, activity, and sequence diagrams. The correctness of the set is discussed from three perspectives, that is, completeness, consistency, and soundness of the set.

The completeness of the set assures no syntactic conflicts exist within the set, while consistency and soundness guarantee the semantic correctness of the set. In order to evaluate the correctness rigorously using those criteria, all the UML models are transformed into CPN models.

The completeness of the set can be verified by transforming all the state machine models into CPN with preparing associated CPM/ML functions before transforming other types of models. On the other hand, the consistency and soundness verifications need additional CPN components to the transformed CPN models, which examine the order of method invocations and injected states adequately.

The presented process can assure not only the correctness of each model in the set, but also the correctness of complicated interrelationships between the models, so that we can provide more accurate software specifications.

REFERENCES

Amalio, N. and Polack, F. (2003). Comparison of formalisation approaches of uml class constructs in z and object-z. In *3rd international conference on Formal specification and development in Z and B*, pages 339–358. Springer-Verlag.

Eshuis, R. and Wieringa, R. (2003). Comparing petri net and activity diagram variants for workflow modelling: A quest for reactive petri nets. In *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*, pages 321–351. Springer.

Favre, L. and Clerici, S. (1999). Integrating uml and algebraic specification techniques. In *32nd International Conference on Technology of Object-Oriented Languages and Systems*, pages 151–162.

Fischer, C., Olderog, E., and Wehrheim, H. (2001). A csp view on uml-rt structure diagrams. In *4th International Conference on Fundamental Approaches to Software Engineering*, pages 91–1–8. Springer-Verla.

Garrido, J. and Gea, M. (2002). A coloured petri net formalisation for a uml-based notation applied to cooperative system modelling. In *the 9th International Workshop on Interactive Systems. Design, Specification, and Verification*, pages 16–28. Springer-Verlag.

Jensen, K. and Kristensen, L. (2009). *Coloured Petri Nets: Modeling and Validation of Concurrent Systems*. Springer-Verlag.

Knapp, A. and Wuttke, J. (2006). Model checking of uml 2.0 interactions. In *Workshops and Symposia at MoDELS 2006*, pages 45–51.

Lausdahl, K., Lintrup, H., and Larsen, P. G. (2009). Connecting uml and vdm++ with open tool support. In *the 2nd World Congress on Formal Methods*, pages 563–578. Springer-Verlag.

OMG (2011). *Unified Modeling Language Superstructure*. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

Shinkawa, Y. (2006). Inter-model consistency in uml based on cpn formalism. In *13th Asia Pacific Software Engineering Conference*, pages 411–418. IEEE.

Shinkawa, Y. (2008). Model checking for uml use cases. In *Software Engineering Research, Management and Applications 2008*, pages 233–246. Springer.

Shinkawa, Y. (2011). Inter-model consistency between uml state machine and sequence models. In *6th International Conference on Software and Data Technologies, Vol 2*, pages 135–142.

Skoldstam, M., Akesson, K., and Fabian, M. (2007). Modeling of discrete event systems using finite automata with variables. In *46th IEEE Conference on Decision and Control*, pages 3387–3392. IEEE.

Warmer, A., Bast, J., and Kleppe, W. (2003). *MDA Explained: The Model Driven Architecture?: Practice and Promise*. Addison-Wesley Professional.