Checking OCL Expressions Using Colored Petri Nets

Marta E. Calderón
Universidad de Costa Rica, Escuela de Ciencias de la Computación e Informática
San Pedro, Costa Rica
mcaldero@ecci.ucr.ac.cr

Abstract

This paper describes an approach to checking OCL expressions of a UML-based system model using CPN state space tools. The OCL is the part of the UML standard used to specify invariant conditions that must always hold for a system model. An approach to transforming a UML-based system model into a CPN model is taken as basis. Some CPN state space functions traverse all nodes of a state space and can be used to demonstrate that a condition holds. In particular, when a UML-based system model is transformed into a CPN model, CPN traversing functions can be used to demonstrate that an OCL expression holds. OCL expressions are transformed into CPN state space functions. These functions list all nodes in which the OCL expression does not hold. Using this information, software engineers can verify the UML-based system model and detect the presence of defects causing the OCL expression violation. Function results depend on the CPN model initial marking. Two OCL expression examples are presented to show how transformation and checking are done.

Keywords: Software Engineering, OCL, UML, Colored Petri Nets

1. INTRODUCTION

The Unified Modeling Language (UML) is a visual modeling language that can be used to capture functional requirements and decisions to be taken in software systems that need to be constructed [3]. UML has become an industry wide standardized notation for object–oriented development [3, 8]. Many systems models have been specified using UML notation. The Object Constraint Language (OCL) is the part of the UML standard used to specify invariant conditions through expressions. Invariant conditions must always hold for a system model [13]. OCL is a language used to provide more detail of UML specifications using an unambiguous language.

A UML-based system model is not an executable specification. However, several approaches have been suggested in terms of transformation of an UML-based system model to its corresponding Colored Petri Nets (CPN) model for analyzing dynamic properties of a UML-based system model. In [4], a transformation which uses the UML use case and sequence models to generate a hierarchical CPN is described. In [1, 9], statecharts and the collaboration model are used to generate a CPN model. These transformation approaches do not consider the relationships among use cases. The relationship between use cases is fundamental to provide a basis for the execution sequence for a large-scale UML model when it is mapped into a hierarchical CPN.

In [10, 11], the description of a transformation approach using the use case diagram, the collaboration model and the class model is presented. In [2, 12], a prototype tool supporting automation of this transformation approach is described. Tool support allows software engineers to apply model transformation to large-scale distributed systems and check dynamic behavior properties such as deadlock using the state space tools provided by a CPN tool [5, 6].

Transforming an UML-based system model into a CPN model also opens the possibility of checking whether an OCL expression associated with the UML-base system model holds. In this paper, an approach to checking OCL expressions using CPN state space tools is described. This study is limited to OCL expressions which express invariants and, therefore, return a Boolean value. The transformation approach from a UML-based system model to a CPN model described in [11] is taken as basis for this study, because objects instantiated from UML classes with attributes are transformed into CPN places. The CPN Tools developed at the University of Aarhus are used to show results.

This paper is organized as follows. Section 2 describes the transformation approach from an UML-based system model to a CPN model described in [11]. Section 3 describes a transformation approach for checking OCL expressions using CPN state space tools. Section 4 describes two examples of OCL expression checking. Finally, Section 5 concludes this paper.

2. TRANSFORMATION OF A UML MODEL

Details of the process of transforming a UML–based system model consisting of the use case model, the class model and the collaboration model into a hierarchical CPN model are taken from [11]. The hierarchical structure of the CPN model is useful for reducing the complexity of a plain CPN model for large–scale systems. However, the UML-based system model does not have a hierarchical structure. The use case model shows the relationships between use cases using use case dependencies through "include" and "extend," but these relationships are planar rather than hierarchical. Therefore, it is necessary to transform a UML-based system model into a hierarchical model. To achieve this, two relationships are defined:

Relationship 1: Each use case in the use case model is realized by the collaboration model.
Relationship 2: An object in the collaboration model is instantiated from a class in the class model.

To address scalability, a large scale UML based system model is transformed into a hierarchical CPN model organized in three layers: a use case layer, an object layer, and an operation layer. Figure 1 shows an overview of the hierarchical structure of a CPN model derived from a UML model. The use case layer ((b) in Figure 1) is the highest level of the CPN model and is derived from the use case model. A use case is transformed into a transition, for example, the UseCase1 transition in Figure 1. A transition is created for each actor participating in the UML model, for example, the transition Actor in Figure 1.
Message communication occurring between an actor transition and a use case transition is derived from the collaboration model that describes message communication between the actor and objects (P1 in Figure 1). Messages occurring between two use case transitions (P2 in Figure 1) are defined by relationships between use cases.

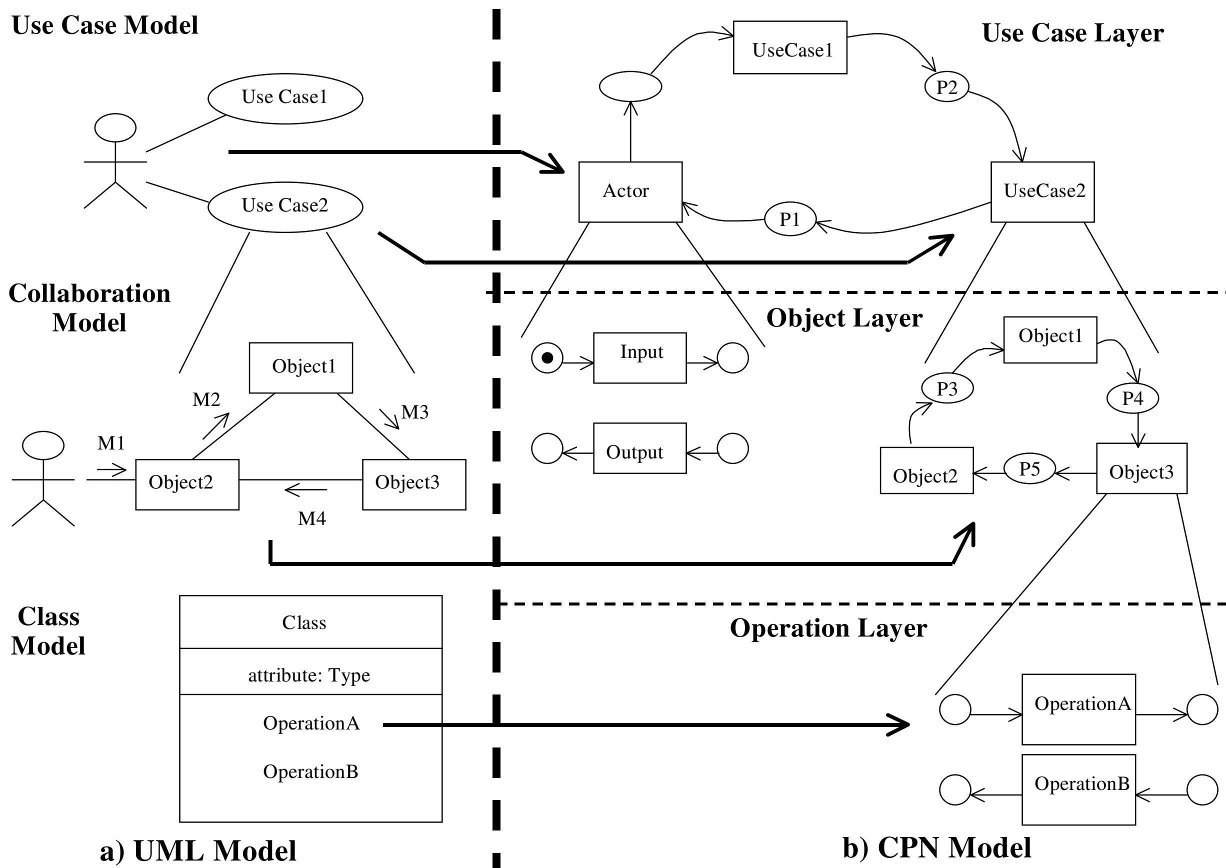

Figure 1. Overview of CPN Model Mapped from UML–based System Model (taken from [11], p. 41)

According to relationship 1 above, a use case in the use case model is realized by a collaboration diagram in the collaboration model. The collaboration diagram is supported by objects. Therefore, a transition of the use case layer in the CPN model (e.g., UseCase2 in (b) of Figure 1) is decomposed into a set of object transitions on the object layer

(e.g., Object1, Object2, and Object3 in (b) of Figure 1). Similarly, as actors both provide input to and receive output from the system, an actor transition in the use case layer (e.g., Actor in (b) of Figure 1) is decomposed into input and/or output transitions in the object layer (e.g., Input and Output transitions in (b) of Figure 1).

If an object in a collaboration model is an instance of a class with attributes defined in the UML class model, then the object is decomposed into both a transition and a place in the object layer. The attributes of the object will be stored in the place. Additionally, if the object with attributes participates in several collaboration diagrams, it is important to maintain consistent information in the CPN model. Therefore, a place to represent the attributes should also be present in the use case layer, so that all the use cases supported by the same object refer to the same place and have access to the same information values.

In the operation layer of the CPN model, an object transition on the object layer (e.g, Object3 in Figure 1) is decomposed into operation transitions (e.g., operationA and operationB). This can be done due to two facts. First, an object in the collaboration model is instantiated from a class in the class model, as stated above in relationship 2. Second, an object provides operations to other objects.

Colorsets in the CPN model are derived from the UML class model. A colorset is created for each message class and its attributes. The message classes are captured from message communication occurring between objects in the UML collaboration model. Each message class is a type of message that is transformed into a colorset. Colorsets are also created for classes that have attributes defined in the UML class model. Variables are declared for all colorsets. All the colorsets and their variables are specified in the declaration area of the CPN model.

This approach to model transformation from a UML-based model to a CPN model addresses the scalability of model transformation. A large scale UML-based system model is usually a very complex, flat structure. The transformation into a hierarchical CPN model reduces the complexity of plain UML-based system models in the case of a large-scale system. Additionally, the dynamic behavior of the transformed UML-based system model can be validated using analysis tools provided by CPN tools such as the CPN Tools[5]. The individual scenarios of each use case can be checked using CPN Tools simulation options, and dynamic behavior properties can be checked using the space state (also called occurrence graph) and strongly connected component graphs generated by CPN Tools [5].

3. OCL Expression Checking

OCL expressions can be associated with a classifier such as a class in a class diagram [7]. The classifier becomes the context within which the invariant must hold. If a class with attributes is the context of an OCL expression, the expression may refer to constraints about the attribute values of an object instantiated from the class. This is the kind of OCL invariants analyzed in this study.

According to the transformation model described on Section 2, if an object in a collaboration model is instantiated from a class with attributes defined in the UML class model, then the object is decomposed into both a transition and a place in the object layer. If the object supports more than one use case, a place representing its data is created in the use case layer in order to keep data consistency [11]. This information is very important in order to establish an approach for checking OCL expressions using CPN, because it makes possible to relate the context of an OCL expression to a place in a CPN model.

The state space tool provided by CPN Tools allows a user to define non-standard queries to investigate properties of a CPN [5]. Some functions provided by the state space tool, such as *PredAllNodes*, traverses the nodes of the state space generated. This powerful feature is useful to prove that constraints always hold in a system model. State space queries can reference contents of one or more places and are independent of the rest of the CPN model. In particular, checking that an OCL expression holds is done transforming it into a CPN function or set of functions referencing the place created to represent the object (and its data) derived from the context class of the OCL expression. CPN function arguments are specific to a place, depending on the colorset assigned to it. Therefore, if the same constraint needs to be checked for two different places, it is necessary to change query arguments, but the function logic is reusable.

Understanding the CPN state space instruction *PredAllNodes* is fundamental for the transformation of an OCL expression to a non-standard CPN query. *PredAllNodes* lists all state space nodes satisfying a specified Boolean predicate function [5]. The predicate function is evaluated for each node in the state space. In order to know in which state space nodes the OCL expression does not hold, the checking approach followed is the comparison of the results of two functions (see Figure 2). The first function returns a multiset (a set allowing repeated elements) representing the ideal situation in which the OCL expression holds for every state space node. The second function returns a multiset reflecting what actually happens, detecting state space nodes in which the OCL expression does not hold. The second function logic depends on the OCL expression. Both multisets have the same number of elements, which is guaranteed using the same input argument for both functions. The input argument depends on the OCL expression.

For each state space node, the two multisets are compared, resulting in a Boolean value returned. If the two multisets differ from each other, the OCL expression does not hold (see Figure 2). After analyzing all state space nodes, *PredAllNodes* displays the list of nodes in which the OCL expression does not hold. Using this information, a software

engineer can verify the UML-based system model and detect the presence of defects causing the OCL expression violation. Checking results depend on the CPN model initial marking.

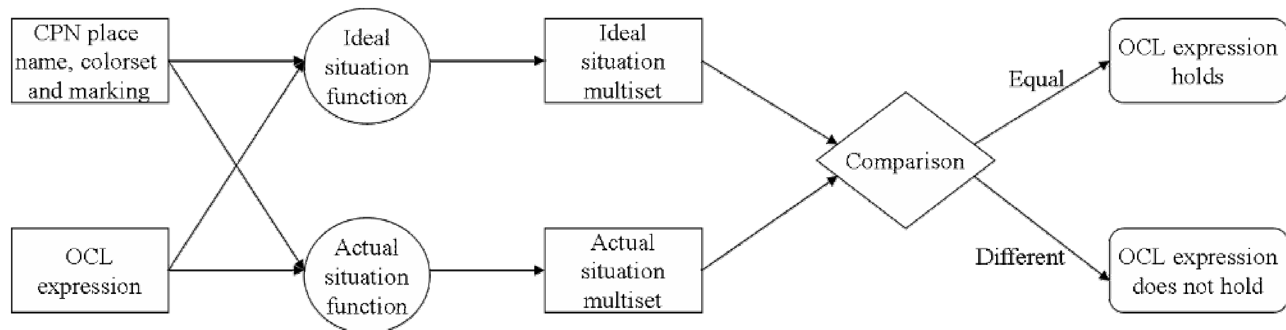

Figure 2. Comparison Approach to Checking OCL Expressions

The construction of the function detecting whether the OCL expression holds is a programming exercise depending on the OCL expression itself and the place name and colorset. OCL expressions may define maximum or minimum values allowed, relationships between values of two or more attributes, uniqueness constraints, or existence of at least one attribute value, among others. It is not possible to generalize a function or set of functions useful for all types of OCL expressions, but it is possible to generalize them for one type of expression. Section 4 describes two examples in which an OCL expression is transformed to a set of CPN functions and checked to show whether the OCL expression holds or not.

4. Examples

The OCL expression checking approach is described using two examples. The first one shows how to check a minimum value constraint and the second one a uniqueness constraint.

4.1 Minimum Value Constraints

Suppose that in the class model of a UML-based system there is a class called *Company* with two attributes: *name*, and *numberOfEmployees*. An OCL expression indicating that the number of employees in every object instantiated from class *Company* must always be greater than 50 is as follows:

context company **inv**:
    self.numberOfEmployees > 50

When the UML-based system model is transformed into a CPN model, a place called *company* is created in the object layer to store attribute data of an object *company* instantiated from class *Company*. In this case, there is a colorset called *company* assigned to the place called *company*. This is a compound colorset consisting of the product of two simple colorsets: *companyName* and *numberOfEmployees*. CPN colorset and variable declarations are shown in Figure 3. Colorsets and variables are created when the UML-based system model is transformed into the CPN model. Additional variables may be required when creating CPN code required to check an OCL expression.

```
▶ colset companyName
▶ colset numberOfEmployees
▼ colset company  = product
  companyName *numberOfEmployees;
▼ var n : companyName;
▼ var e : numberOfEmployees;
```

Figure 3. CPN Declarations for Constraint Checking

Once colorsets are known, a set of CPN state space functions for checking that an OCL expression holds is constructed. The CPN state space function *PredAllNodes* is used. The predicate function is built to extract the nodes in which the number of employees for at least one company is less than 51 (see Figure 4). The predicate function consists of comparing the results of two functions: *returnsFalse'* and *numberOfEmployeesLess51'*. Both functions are called

extended functions because they take advantage of the possibility of extending a function to each element in a multiset [5]. This is achieved using the predeclared function *ext_col*, as shown in Figure 4. The two functions in the predicate function are designed to return multisets with the same number of elements. *returnsFalse'* represents the situation in which the OCL expression holds, inserting a false value for each company in place *company*. *numberOfEmployeesLess51'* represents the actual situation, adding a true value to the multiset when a company has less than 51 employees and a false value when it has more. If at least one company has less than 51 employees, the two resulting multisets are different for the node being evaluated and the *PredAllNodes* function displays the node number.

```
fun returnsFalse((n,e): company) = false;
val returnsFalse' = ext_col returnsFalse;
fun numberOfEmployeesLess51 ((n,e): company)=
  (e <= 50);
val numberOfEmployeesLess51' = ext_col
  numberOfEmployeesLess51;
PredAllNodes (fn n =>
  returnsFalse' (Mark.Page'company 1 n) <><>
   numberOfEmployeesLess51'
     (Mark.Page'company 1 n));
```

Figure 4. OCL Mininum Value Constraint Transformation to CPN Code

Figure 5 shows a partial CPN model in which the OCL expression does not hold. This state was reached after simulating the CPN model. The CPN code shown in Figure 4 can be executed in CPN Tools after entering the state space tool, calculating the state space, and calculating the strongly connected component graph. Results of executing function *PredAllNodes* using the same CPN initial marking used when the CPN was simulated are displayed in Figure 6. The last line contains the list of state space node numbers in which the OCL expression does not hold. In this example, state space nodes 2 and 4 violate the OCL expression. Node details (marking of all places in the CPN model) can be displayed using function *NodeDescriptor* [5].

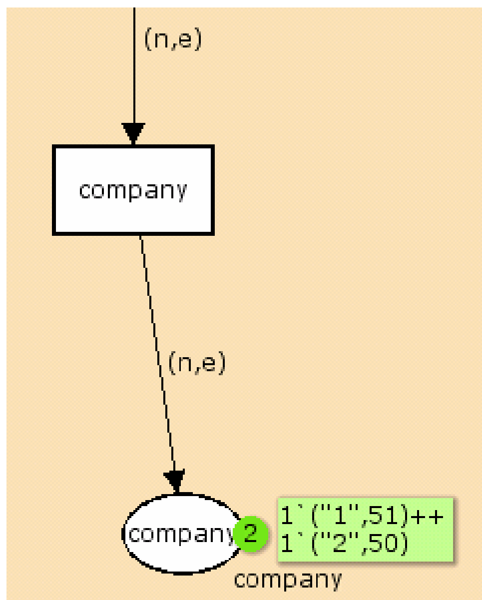

Figure 5. Partial CPN Model Showing OCL Expression Violation

```
val returnsFalse = fn : company -> bool
val returnsFalse' = fn : company ms -> bool ms
val numberOfEmployeesLess51 = fn : company -> bool
val numberOfEmployeesLess51' = fn : company ms -> bool ms
val it = [4,2] : Node list
```

Figure 6. Results of Executing the OCL Expression Checking Function

4.2 Uniqueness Constraints

More complex constraints such as uniqueness can also be checked. For example, class *Person* has attributes *socialSecurityNumber*, *firstName*, and *lastName*. Attribute *socialSecurityNumber* must be unique for each person. This constraint is expressed using the following OCL expression:

```
context Person inv:
        Self.allInstances->forAll(p1, p2 | p1<>p2 implies
                p1.socialSecurityNumber <> p2.socialSecurityNumber)
```

Figure 7 shows CPN colorset and variable declarations and Figure 8 shows a partial CPN model showing constraint violation. Using the same approach of multiset comparison used in Section 4.1, function *PredAllNodes* detects all state space nodes in which the OCL expression is violated. The code is more complex than in the first example because probing this uniqueness constraint requires creating two nested loops in which the social security number of a person (outer loop) is compared with the social security number of all the other persons in place *person* (inner loop) for each state space node (see Figure 9). Extended function *equality'* is the responsible for determining the state space nodes in which the OCL constraint does not hold.

```
▼Declarations
  ▼colset socialSecNumber = string;
  ▼colset firstName = string;
  ▼colset lastName= string;
  ▼colset person = product
   socialSecNumber * firstName *
   lastName;
  ▼colset socSecNumberPair = product
   socialSecNumber * socialSecNumber;
  ▼colset node = int;
  ▼colset specialPair = product
   socialSecNumber * node;
  ▼var ssn1, ssn2 : socialSecNumber;
  ▼var fn1, fn2 : firstName;
  ▼var ln1, ln2 : lastName;
  ▼Standard declarations
```

Figure 7. CPN Declarations for Uniqueness Constraint Checking

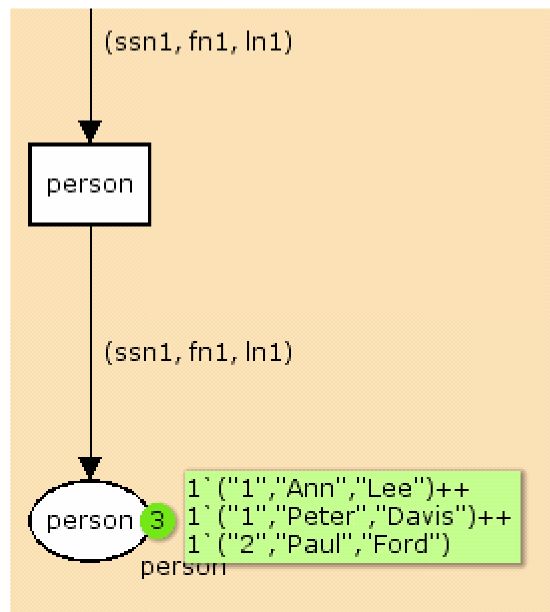

Figure 8. Partial CPN Model Showing Uniqueness Constraint Violation

```
fun getSocSecNumber ((ssn1, fn1, ln1): person) = ssn1 ;
val getSocSecNumber' = ext_col getSocSecNumber;

fun equality ((ssn1, ssn2) : socSecNumberPair) =
   (ssn1 = ssn2);
val equality' = ext_col (ext_col equality);

fun makeSocSecNumberPairs ((ssn1, n): specialPair) =
   socSecNumberPair.mult (1`ssn1,
         (getSocSecNumber' (Mark.Page'person 1 n)) -- 1`ssn1);
val makeSocSecNumberPairs' = ext_col makeSocSecNumberPairs;

fun returnsFalse((ssn1, ssn2) : socSecNumberPair) = false;
val returnsFalse' = ext_col (ext_col returnsFalse);

PredAllNodes (fn n =>

      ((equality' (makeSocSecNumberPairs' (specialPair.mult
                  (getSocSecNumber' (Mark.Page'person 1 n), 1`n))))
      <><>
      (returnsFalse' (makeSocSecNumberPairs' (specialPair.mult
                  (getSocSecNumber' (Mark.Page'person 1 n), 1`n)))))

);
```

Figure 9. OCL Uniqueness Constraint Transformation to CPN Code

4.3 Comparison of Examples

Extended function *returnFalse'* shown in Figure 4 and extended function *returnFalse'* shown in Figure 9 have the same purpose – representing the ideal situation in which the OCL expression holds –, but arguments are different due to the OCL expression type. In both examples, *returnFalse'* argument is the same as the argument of the other function in the Boolean predicate function (*numberOfEmployessLess51'* in Figure 4 and *equality'* in Figure 9). This is done to guarantee that the two multisets have the same number of elements.

The two examples show that CPN code is dependent on the OCL expression type and the place name and colorset. Checking a similar OCL constraint for another place requires changing place names and function arguments but the logic is reusable. This opens the possibility of transformation automation because it is possible to define a set of functions to be executed for checking each OCL expression type.

5. CONCLUSIONS

This paper has described an approach to check OCL expressions of a UML-based system model using CPN state space tools. The CPN model used for checking is the result of the transformation of a UML-based system model represented by the use case model, the collaboration model, and the class model. The approach takes advantage of the fact that an object instantiating from a class having attributes is transformed into both a transition and a place. OCL expressions having such a class as context are checked using CPN state space functions. The CPN function *PredAllNodes* is used to determine the set of state space nodes in which the OCL expression does not hold. The Boolean predicate function executed by *PredAllNodes* compares the ideal situation in which the OCL expression holds with the actual situation. Two examples of OCL expression checking have been shown. The transformation approach from an OCL expression to a set of CPN functions described on this paper is manual. Transformation may be automated by using place colorset and OCL expression type information.

References

[1] Baresi, L. and Pezzè, M. “On Formalizing UML with High—Level Petri Nets.” In *Concurrent Object–Oriented Programming and Petri Nets: Advances in Petri Nets.* LNCS 2001, Gul A. Agha, Fiorella De Cindio, and Grzegorz Rozenberg eds, Berlin, Germany, 2001, pp. 276-304.

[2] Calderón, M.E. and Shin, M.E. “Tool Support for Model Transformation to Analyze Dynamic Behavior of Large-Scale Systems.” In *Proceedings of the 2005 Design, Analysis, and Simulation of Distributed Systems Symposium DADS’05* (San Diego, California, April 3-7), pp 107-114.

[3] Booch, G., Rumbaugh, J.; and Jacobson, I. *The Unified Modeling Language User Guide*. Reading, Massachusetts: Addison Wesley, 1999.

[4] Elkoutbi, M. and Keller, R.K.. "User Interface Prototyping Based on UML Scenarios and High-level Petri Nets." In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets* (Aarhus, Denmark, June 26-30) Springer-Verlag, Germany, 2000, pp 166-186.

[5] Jensen, K., Christensen, S. and Kristensen , L.M. *CPN Tools State Space Manual*. Aarhus, Denmark: University of Aarhus, 2002.

[6] Kristensen, L.M., Christensen, S., and Jensen, K. "The Practioner's Guide To Coloured Petri Nets." In *International Journal on Software Tools for Technology Transfer.* Vol 2, No. 2 (1998), pp 98-132.

[7] Object Management Group. *UML 2.0 OCL Specification*. June 2005, http://www.omg.org/docs/ptc/05-06-06.pdf.

[8] Rumbaugh, J., Booch, G., and Jacobson, I. *The Unified Modeling Language Reference Manual,* Reading, Massachusetts: Addison Wesley, 1999.

[9] Saldhana, J., Shatz, S.M., and Hu, Z.. "Formalization of Object Behavior and Interactions from UML Models." In *International Journal of Software Engineering and Knowledge Engineering (IJSEKE),* Vol 11, No. 6, (December, 2001), pp 643-673.

[10] Shin, M.E., Levis, A., and Wagenhals, L. "Mapping of UML-based System Model to Design/CPN Model for System Model Evaluation." Presented at the *Workshop on Compositional Verification of UML'03* (San Francisco, CA, October 22), 2003.

[11] Shin, M.E., Levis, A., Wagenhals, L., and Kim, D. "Analyzing Dynamic Behavior of Large–Scale Systems through Model Transformation." In *International Journal of Software Engineering and Knowledge Engineering.* Vol 15*,* No.1 (February, 2005), pp. 35-60.

[12] Shin, M.E and Calderón, M.E. "Meta-Modeling Approach to Tool Support for Model Transformation to Validate Dynamic Behavior of Systems." In *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP'05).* (Las Vegas, Nevada, June 27-29), pp 316-322.

[13] Warner, J. and Kleppe, A. *The Object Constraint Language: Precise Modelling with UML.* Reading, Massachusetts: Addison Wesley, 1999.