RC23211 (C0405-001) May 12, 2004
Mathematics

IBM Research Report

Towards Formal Verification of UML Diagrams Based on Graph Transformation

Yu Zhao[1], Yushun Fan[1], Xinxin Bai[1], Yuan Wang[1], Hong Cai[2], Wei Ding[2]
[1]CIM Research Center
Department of Automation
Tsinghua University
Beijing, China 100084

[2]IBM Research Division
China Research Laboratory
HaoHai Building, No. 7, 5th Street
ShangDi, Beijing 100085
China

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Towards Formal Verification of UML Diagrams Based on Graph Transformation

Yu Zhao[1], Yushun Fan[1], Xinxin Bai[1], Yuan Wang[1], Hong Cai[2], Wei Ding[2]
[1] CIM Research Center
Department of Automatioin
Tsinghua University, Beijing, China 100084
zhaoyu00@mails.tsinghua.edu.cn, fan@cims.tsinghua.edu.cn, {bxx02, yuan-wang02}@mails.tsinghua.edu.cn
[2] IBM China Research Lab
4F, Haohai Building, 5th Shangdi Street
Haidian District, Beijing China 100085
{caihong, dingw}@cn.ibm.com

Abstract

In this paper, a meta-level and highly automated technique that could formally transform UML diagrams for verification is presented. Firstly, the meta-model hierarchical structure of UML is reviewed and the relationships among different UML diagrams are analyzed from different views. An approach for transforming and verifying UML statechart diagrams is developed based on graph transformation. This approach can be used to transform UML diagrams to Petri-nets while preserving dynamic consistency properties. Finally, the approach is validated through a case study.
Keywords: Graph Transformation, UML, Formal Verification, Petri-nets

1. Introduction

Models make (software) platforms more accessible by raising the level of abstraction and enabling reuse of assets across platform changes. The Unified Modeling Language (UML), is a well known dominant object-oriented modeling language for the design process of IT systems.

However, despite its success as being a unified and visual notation in industrial, UML still lacks analysis and verification capabilities. Therefore, many models cannot really be analyzed in detail, in particular when they are used to describe complex, distributed systems. Being lack of analysis and verification methods limits the use of UML and reduces the quality of the UML models. Thus developing technologies of analysis and verification of UML model is significant to modelers who use UML to model their systems.

UML is lack of a precise formal semantics, which hinders the formal verification and validation of system design. On the other hand, models established in many mathematical domains (like Petri-nets, transition systems, process algebras, etc.) are precise and could be analyzed and verified by using various tools in these domains. So transformations between UML and these models are significant for analysis and verification of UML model. UML models are projected into mathematical models for transformation, and the results of the formal analysis are back-annotated to the UML models to hide the mathematics from designers.

Because of not only the abundant analysis techniques but also the understandability of Petri-nets, and some existing successful research on the verification of the UML dynamic view diagrams by Petri-nets, we also choose the Petri-nets as the target model of transformation for verification of UML models. In order to get a more general transformation approach between UML and Petri-net, we

research the transformation at the meta-model level. And for reaching a automatic and correct process, we use graph transformation grammars and systems to define and implement the transformation, and verify the transformation itself.

The rest of the paper is structured as follows. In Section 2, the related works are reviewed first. Existing researches on the verification and transformation of UML models mostly focus on the various model elements, but the relationships among different diagrams are rarely discussed, so In Section 3, the relationships among different UML diagrams and the potential target Petri-net of the transformation for these relationship and the verification contents and approaches are analyzed Then In Section 4, a brief introduction to Petri-net and its analysis ability are given. In Section 5, the graph transformation technique that could transform UML diagrams into Petri-net is discussed. Because of the importance of the verification of the transformation itself, some properties of a correct transformation are analyzed. In section 6, we give a case study on transforming concrete UML statecharts into Petri-nets using graph transformation techniques and present our debugging approach for guarantee the correctness of transformation, too. Finally, the paper is concluded in Section 7 with some concluding remarks and future directions.

2. Related work

Several projects based on graph transformation have been carried out to support model transformation (not limit for UML):

AToM^3 [1][2] is a tool for Multi-Paradigm modeling under development at the Modeling, Simulation and Design Lab (MSDL) at McGill University. The two main tasks of AToM^3 are meta-modeling and model-transformation. UML diagrams are transformed by AToM^3 into Petri-nets for verification by means of simulation. And it transforms the various diagrams (just from dynamic view) into one combined Petri-net, and also accepts the UML models with incomplete or redundant information. But it doesn't consider the diagrams of static view, and the transformation environment is not responsible for the correctness of transformation.

GROOVE[3] is a project centered around the use of simple graphs for modeling the design-time, compile-time, and run-time structure of object-oriented systems. Graph transformation is used as a basis for model transformation and to define the operational semantics of language. Its shortcoming is the same as that of AToM^3 that it lacks the verification of transformation itself

The VIATRA[4] (VIsual Automated model TRAnsformations) model transformation system is a prototype tool that provides a general and automated framework for specifying transformations between arbitrary models conforming to their meta-model. The outstanding contribution of VIATRA approach is the consideration of the verification of transformation. Fundamental characteristics of a correct transformation have been summarized including the semantic correctness. But its model checking based approach for verification of semantic correctness focuses on the model level but not the meta-model level. So the verification is not general enough and hard to be implemented automatically for any arbitrary source model.

3. UML meta-model and relationship between various diagrams

UML is based on a type of simple, general and unambiguous meta-modeling theory. UML meta-model language is defined in itself recursively, i.e. it's specified by a subset of UML annotations and semantics. The structure of UML meta-model[5][6] is conformable with the MOF (Meta-Object Facility) framework, and has four layers: meta-meta-model, meta-model, model and

object.

In UML models, the various diagrams are not isolated, but are rather correlated. Some of the relationships among the diagrams reflect the grammar rules and semantics of UML itself, e.g. an object in the object diagram must be an instance of an existing class. Other relationships reflect some essential characteristics of the system to be modeled or some rules a correct model must obey, e.g. in the same context, the sequence diagram and collaboration diagram are equivalence in semantics. Thus, when transforming a UML model into a Petri-net model, not only the static structure and dynamic semantic of every single diagram need to be transformed, but also the relationships among the various diagrams should be took into account and even be transformed when needed, such guarantee the correctness of mapping.

The relationships can be classified into three layers:

- The relationships among the same UML diagrams from different contexts

This layer includes the relationships among sequence diagrams, activity diagrams and statechart diagrams from their different contexts, respectively. They are mostly the aggregative relations. So we can use hierarchical Petri-nets as target models to reflect the aggregation and analyze the correctness of the relationships.

- The relationships among the various diagrams from one same view of a system

This layer includes the relationships among the various diagrams from the static structure view (class diagram and object diagram), dynamic behavior view (use-case diagram, sequence diagram, collaboration diagram, statechart diagram and activity diagram) and system architecture view (component diagram and deployment diagram), respectively. The relationships of this layer are mostly the encapsulation or equivalence relations. The mapping of these relations into Petri-nets can also use hierarchical Petri-nets. Otherwise we can transform the different diagrams to their target Petri-net model respectively and verify the equivalence of these Petri-nets.

- The relationship among the various diagrams from various views of a system

This layer describes the relationship between the diagrams of static structure view and the diagrams of dynamic behavior view. All the objects included in the sequence, collaboration, statechart and activity diagram and these objects' attributes and operations must be defined in the classes which these objects belong to. For this layer, we can use colored Petri-nets as target models to model the static properties of the elements in the dynamic behavior models. And the verification of this layer is by and large to evaluate the systems' performance, e.g. the rationality or efficiency of one deployment solution for components of a system. But according to the related work, this layer of relationships is rarely concerned in the research on the verification and transformation of UML models.

4. Petri-Net and its analysis ability

Petri-nets is both a formal and a graphical modeling language. The primary reason why we select the Petri-net as the target models of transformation is the abundant and mature analysis techniques of it. Despite new concepts, such as color, hierarchy, stochastic concept etc., are fused into Petri-nets in succession, the essence of syntax and dynamic behavior properties never changes. The basic properties of systems which can be studied with Petri-nets can be divided into two groups: the behavioral properties, including reachability, boundedenss, liveness, reversibility, coverability, persistent, synchronic distance and fariness, etc.; and the structural properties, such as structural liveness, controllability, structural boundedness,. conservativeness, repetitiveness, consistency and so on. The definition of these properties can be found in [7]. And we can also evaluate the performance

of the system modeled by advanced Petri-nets such as colored or stochastic Petri-net.

So within the transformation process, the content to be verified in the UML models should be translated into one property or one combination of several properties of the corresponding Petri-nets that can be analyzed by existing approaches. If with a transformation rule this translation could not proceed, it should not be called a correct rule.

5. Graph transformation[8] and verification of transformation

As both UML and Petri-net are based on the graphical notation, there comes a possibility of depicting them by the common graph concepts, and with it the possibility of transforming UML models into Petri-nets from the aspect of graph theory. Thereby, this is another reason for selecting Petri-net as the verification tool and considering the graph transformation as the foundation theory and highly automated mechanism for transformation.

Late in 1960's Rosenfeld in the USA and Schneider, Ehrig, Pfender, and Wadsworth in Europe introduced graph transformation for the generation manipulation recognition and evaluation of graphs. Since then graph transformation has been studied in a variety of approaches motivated by application domains such as pattern recognition, semantics of programming languages, compiler description implementation of functional programming languages, specification of database systems specification of distributed systems etc.

A graph consists of a set of labeled nodes and a set of labeled directed edges each of which connects a pair of nodes. Graph transformation consists of applying a rule to a graph and iterating this process. Each rule application transforms a graph by replacing a part of it by a graph. To this purpose each rule ***r*** contains a left hand side *L* and a right hand side *R*. The application of ***r*** to a graph G replaces an occurrence of the left hand side *L* in *G* by the right hand side *R*. The definition of rules and application of rules in graph transformation are[10]:

Rules: A graph transformation rule ***r*** = *(L, R, K, glue, emb, appl)* consists of two graphs *L* and *R* , called the left hand side and the right hand side of ***r*** , respectively, a subgraph *K* of *L* called the interface graph, an occurrence *glue* of *K* in *R*, relating the interface graph with the right hand side, an embedding relation *emb*, relating nodes of *L* to nodes of *R,* and a set *appl* specifying the *application conditions* for the rule

Application of rules: An application of a rule ***r*** = *(L, R, K, glue, emb, appl)* to a given graph *G* yields a resulting graph *H,* provided that *H* can be obtained from G in the following five steps:

1) Choose an occurrence of the left hand side *L* in *G.*
2) Check the application conditions according to *appl.*
3) Remove the occurrence of *L* up to the occurrence of *K* from *G* as well as all *dangling edges,* i.e. all edges incident to a removed node. This yields the *context graph D* of *L* which still contains an occurrence of *K.*
4) Glue the context graph *D* and the right hand side *R* according to the occurrences of *K* in *D* and *R.* That is, construct the disjoint union of *D* and *R* and for every item in *K*, identify the corresponding item in *D* with the corresponding item in *R,* This yields the *gluing graph E.*
5) Embed the right hand side *R* into the context graph *D* according to the embedding relation *emb:* For each removed dangling edge incident with a node *v* in *D* and the image of a node *v'* of *L* in *G,* and each node *v''* in *R*, a new edge (with the same label) incident with *v* and the node *v''* is established in *E* provided that (*v', v''*) belongs to *emb.*

Fig. 1 illustrates the steps which have to be performed when applying a rule *(L, R, K, glue, emb,*

appl)

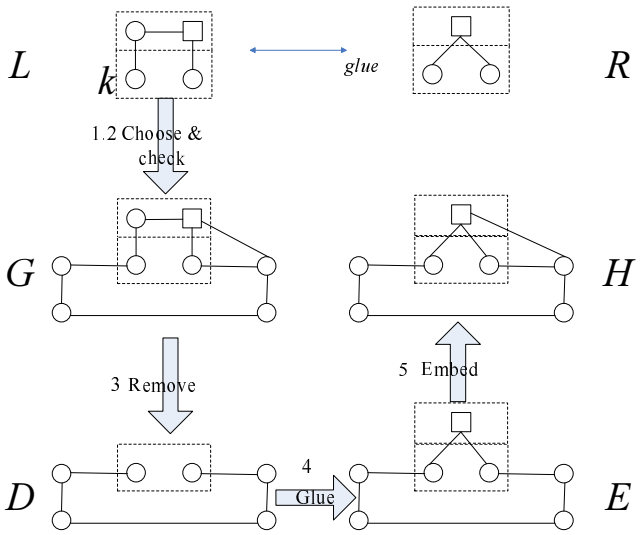

Fig. 1 Illustration of a graph transformation step

Given the notions of a rule and a direct derivation, graph transformation systems can be defined [11]. A set *P* of rules is the simplest form of a *graph transformation system*, a set *P* of rules together with an *initial* graph *S* and a set *T* of *terminal* labels forms a *graph grammar.* Given a set *P* of rules and a graph G_0, a sequence of successive direct derivations $G_0 \rightarrow G_1 \rightarrow \cdots \rightarrow G_n$ is a *derivation* from G_0 to G_n by rules of *P,* provided that all used rules belong to *P.* The graph G_n is said to be *derived* from G_0 by rules of *P.* The set of all graphs labeled with symbols of *T* only that can be derived from the initial graph *S* by rules of *P,* is the *language* generated by *P, S* and *T.* When we use the graph transformation system, we first have to choose one of the rules applicable to a given graph. Furthermore the chosen rule may be applicable at several occurrences of its left hand side. So the result of a graph transformation depends on these choices which are still completely arbitrary. This non-determinism may be restricted by control conditions in several ways:

1) By prescribing an order in which rules have to be applied.
2) By determining the next rule depending on the previous one(s).
3) By applying a rule according to its priority.

The graph transformation rules and system can realize the automation of the transformation process, but cannot verify the transformation itself. The formal verification of UML diagrams based on model transformation requires not only that the target model language has the ability to verify the properties we need, but also that the transformation can transform the source model correctly and even effectively. In [9], the fundamental properties of a correct transformation are summarized as that the transformation should be complete, unique, syntactic correct, semantic correct and could terminate. And for a transformation in the interest of verification, we add a performance requirement that the transformation should be effective, i.e. the contents that need to be verified of the source model should be transformed to some properties of the target model that can be analyzed. This additional requirement is correlated with the semantic correctness, because the semantics are usually one part of the contents to be verified. In graph transformation research area, the completeness, uniqueness and termination of transformation can be verified through the existing approaches. But the semantic correctness and the effectiveness have not dealt with, except that a primary consideration of the verification of semantic correctness comes out in [9].

So we try to analyze the reflection on the transformation rules of the requirements of semantic correctness and effectiveness through the following case study.

6. Case study

Within the dynamic view of UML, there are two types of diagrams to describe the dynamic behavior that state-based diagrams (including statechart and activity diagrams) and flow-based diagrams (including sequence and collaboration diagrams). Because the activity diagram is a special type of the statechart, so we first present a model transformation case from UML statechart diagrams (with some static information from class diagrams for boundary analysis) to Petri-nets in order to demonstrate the feasibility of our verification technique based on graph transformation. The case is a simple automat system with two classes, Customer and Automat. (Fig.2). And the statechart diagrams of these two classed are shown by Fig.3 and Fig.4 respectively. The target model we select is the basic Petri-net model. The following transformation rules are applied at first:

- Each state node in statechart diagram is transformed to a place in the Petri-net model.
- Whether the place has a token represents the status of the corresponding state node. A state node that is linked with the starting point is transformed to a place with one token, and other state nodes are transformed to places with no token.
- Each event or action on the state-transition edge is transformed to a place, too.
- Each transition edge in statechart diagram is transformed to a corresponding transition in the Petri-net.

According to these rules, the statechart diagrams of Automat and Customer can be transformed to the Petri-nets in Fig. 5 and Fig. 6, respectively.

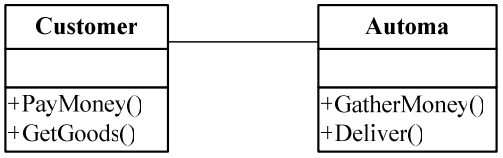

Fig .2 Class Diagrams of Automat and Customer

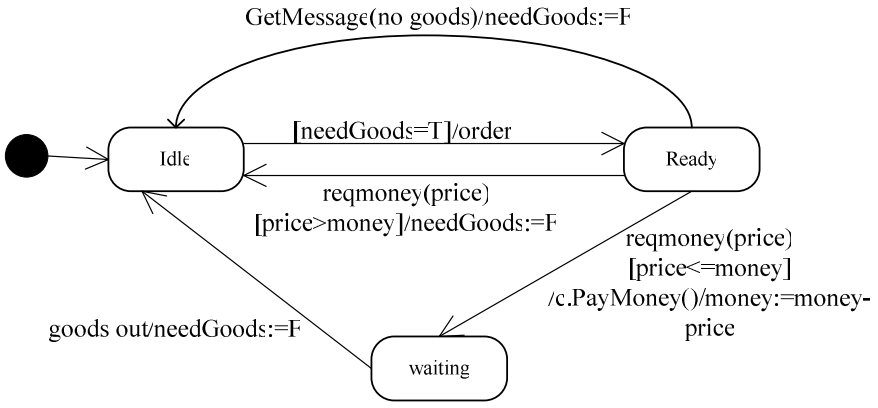

Fig. 3 Statechart of Customer

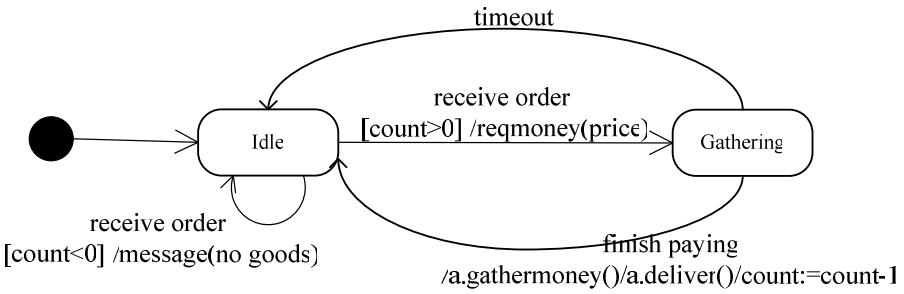

Fig. 4 Statechart of Automat

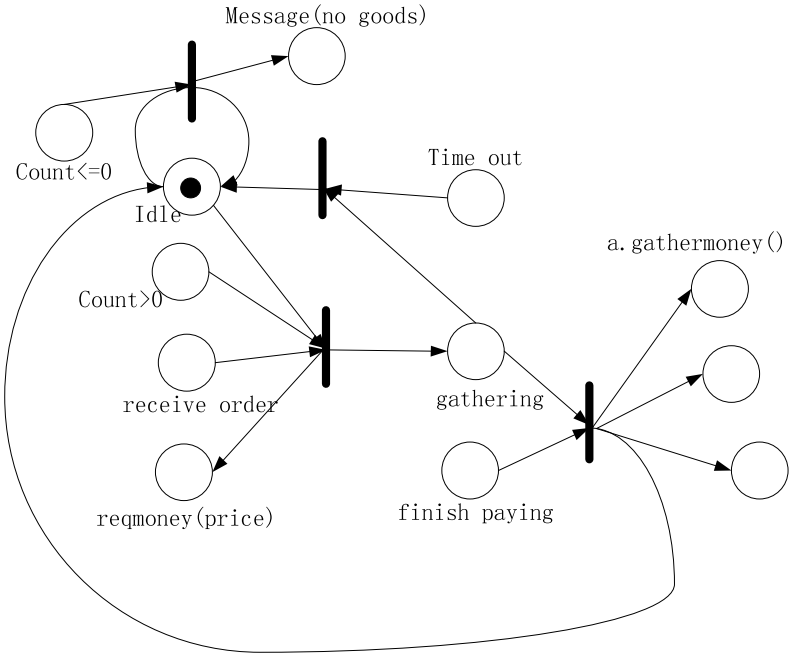

Fig. 5 the Petri-net model of Automat

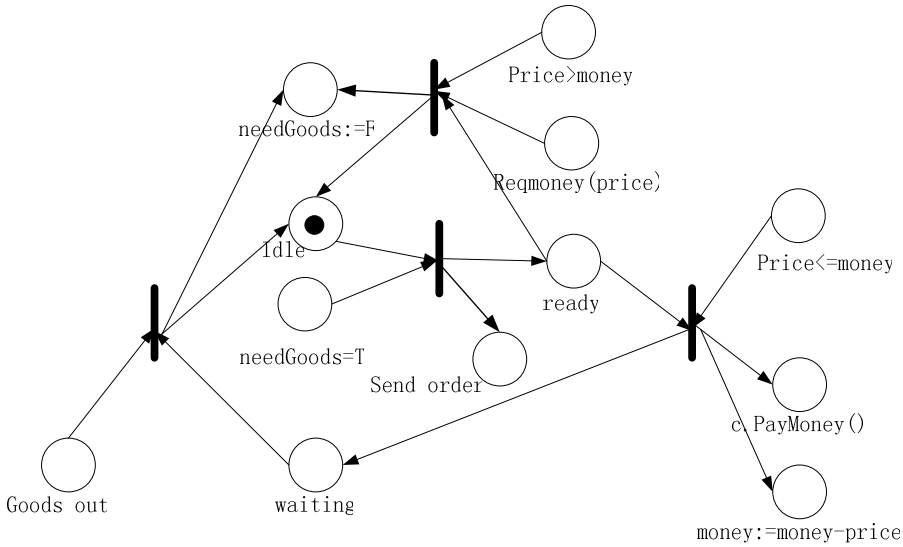

Fig. 6 the Petri-net model of Customer

The completeness, termination uniqueness and syntactic correctness of the above transformation

process can be easily proved. But when we consider the semantic correctness, this transformation should be modified.

A statechart diagram has one dynamic semantic that when the source state of a transition edge is activate and the conditions on the edge are all satisfied, the system is moved from the source stated to the target state of this edge, and one basic semantic constraint that every two state nodes cannot be activated simultaneously. Additionally, for the different statechart diagrams of various objects in one same system, another static semantic which describes the relationships among these diagrams exists, that in the statechart of one object, the events or signals on the transition edges maybe come from i other objects in the system, so the events or signals should be consistent with the corresponding output events/actions in the statechart diagrams of the source objects. From the above case, no semantic of the three can be transformed correctly to the target Petri-net, and we should debug the transformation rules.

- According to the foregoing transformation rules, a guard condition on the transition edge of statechart diagram is also regarded as an event and is transformed to a place without token and input transition. Because of no input transition, the places from conditions have no chance to be activated, and the dynamic semantics of these two models conflict. Thus we add an additional rule for the guard condition, which transforms a condition event into a structure shown by Fig.7 with graph transformation rule, where the new *Place3'* represents the decision of this condition and another new *Place4'* represents the result of decision. The whole structure is regarded as one input for the output of the transition edge where the guard condition is.

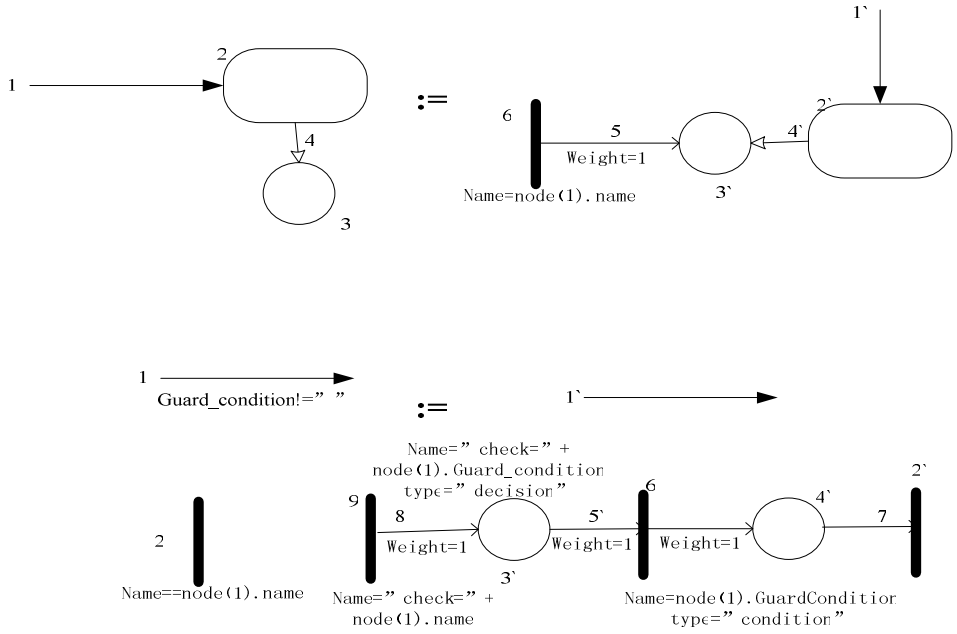

Fig.7 transformation rule for the guard condition

- For a concrete Petri-net model induced by the above transformation rules, the semantic constraint can be described directly by the graph. For example, in the Petri-net model of Customer, this constraint can be defined by three places (Idle, Ready and Waiting) with no two places filled with tokens simultaneously. But in the meta-model layer, this constraint is impossible to be defined for Petri-net. And the reason can be found that we transform both the events and the states into the places. Then we modify the meta-model of the basic Petri-net by assigning an

attribute “type” to place. The places coming from the state nodes are assigned “state” for attribute “type”, while those places coming from the events or actions are assigned “event” for “type”. Then the constraint can be defined as that every two places whose value are both “state” can’t have tokens simultaneously, or that the number of places whose value are “state” after each transition can’t be greater or equal than 2.

- Observing the above statechart diagrams of Customer and Automata, two types of interaction between objects can be summarized then: the generation and receival of signals (e.g. showmessage( )), and the call and feedback of operations (e.g. gathermoney()).In order to identify the connective events/actions during transformation automatically, the label of them should be unique. In our approach, the generation of signal *sig* is labeled with *send_sig*, while the receival of it is labeled with *receive_sig*. On the other hand, the call of operation method is labeled with *call_method*, while the feedback of it is labeled with *return_method*. With these labels, one additional transformation rule for combination of disjoint Petri-net models transformed from different statechart diagrams can be attached: unite the event place coming from *send_sig* and the event place from *receive_sig* into one place and label it with *sig*, and unite the event place coming from *call_method* and the event place from *return_method* into one place and label it with *method*.

After the debug for the semantic correctness, we transform the statechart diagrams of Customer and Automa into a united Petri-net model as shown in Fig.8.

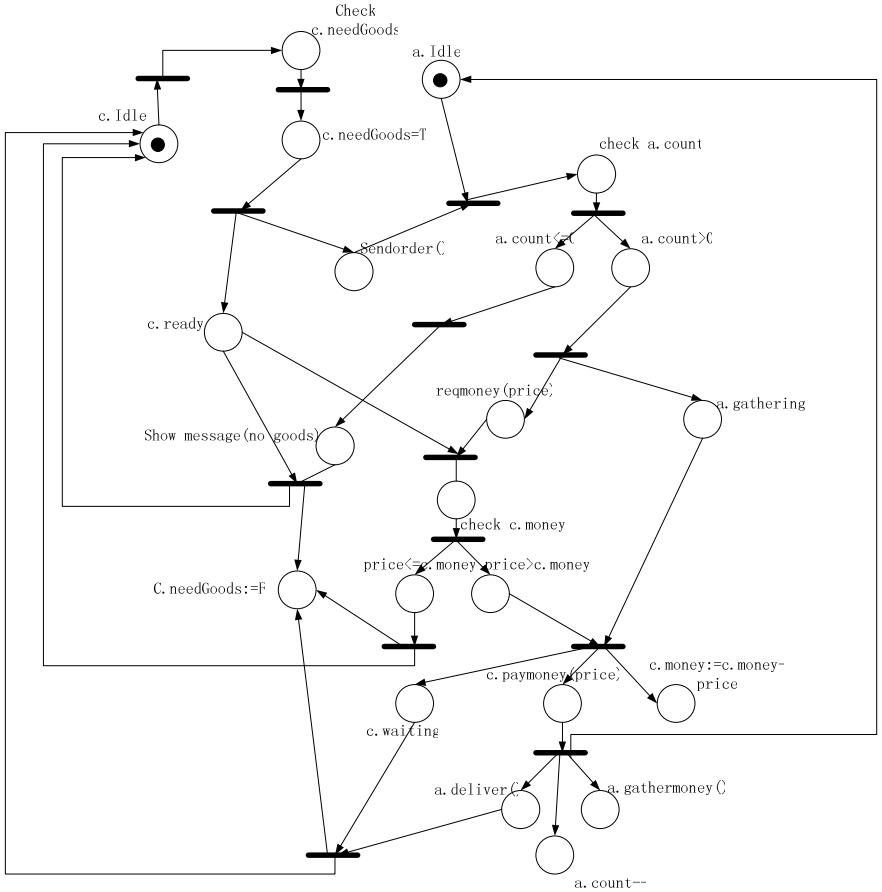

Fig. 8 United Petri-net of Customer and Automa

Then some general transformation rules at meta-model layer from UML statechart diagram to

Petri-net model can be induced from the above example. First the meta-models of statechart and Petri-net is shown by Fig.9 and Fig.10, respectively. (The meta-model of Petri-net has been extended according to the above discussion).

Based on the meta-models, we denote the ordinal transformation rules by graph transformation grammars as shown in Fig.11 and Fig.12.

Rule 1: States to Places. To transform all states in the statechart to the places with type valued "state" of Petri-net.

Rule 2: Generate Transitions: To transform all transition edges to the transitions and to maintain the link with the corresponding places.

Rule 3: OutputEvents to Places: To transform all output events on the transition edges to the places with type valued "event" whose input transitions come from the transition edges by Rule 2.

Rule 4: Guard_condition to Place: To transform the guard conditions on the transition edged to the corresponding structure of Petri-net according to the foregoing discussion.

Rule 5: Input Events to Places I: To transform all input events on the transition edges to the places with type valued "event" whose output transitions come from the transition edges by Rule 2. This Rule is applicable for those transition edges where have no guard condition.

Rule 6: Input Events to Places II: To transform all input events on the transition edges to the places with type valued "event" whose output transitions come from the guard condition on the same transition edges by Rule 4. This Rule is applicable for those transition edges where have guard condition.

Rule 7: Combine Event-pairs I: To combine the pair places representing the sending and receiving of one same signal into one place.

Rule 8: Combine Event pairs II: To combine the pair places representing the call and feedback of one same method into one place.

Rule 9: Initial State to Token: To add one token in the places whose corresponding states are linked with the starting point in the statechart.

Rule 10: Delete Edges: To delete the transition edges.

Rule 11: Delete States: To delete the state nodes.

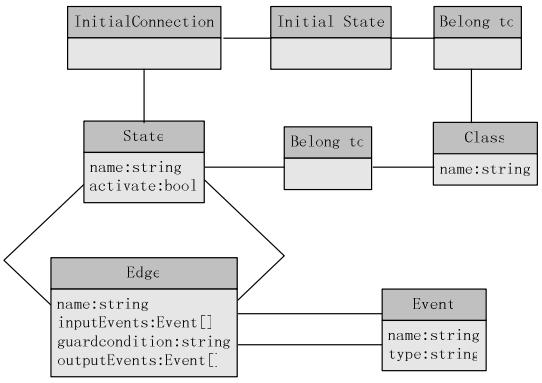

Fig. 9 Meta- Model of StateChart

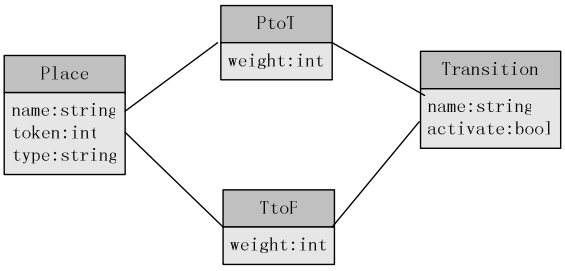

Fig. 10 Modified Meta- Model of Petri-net

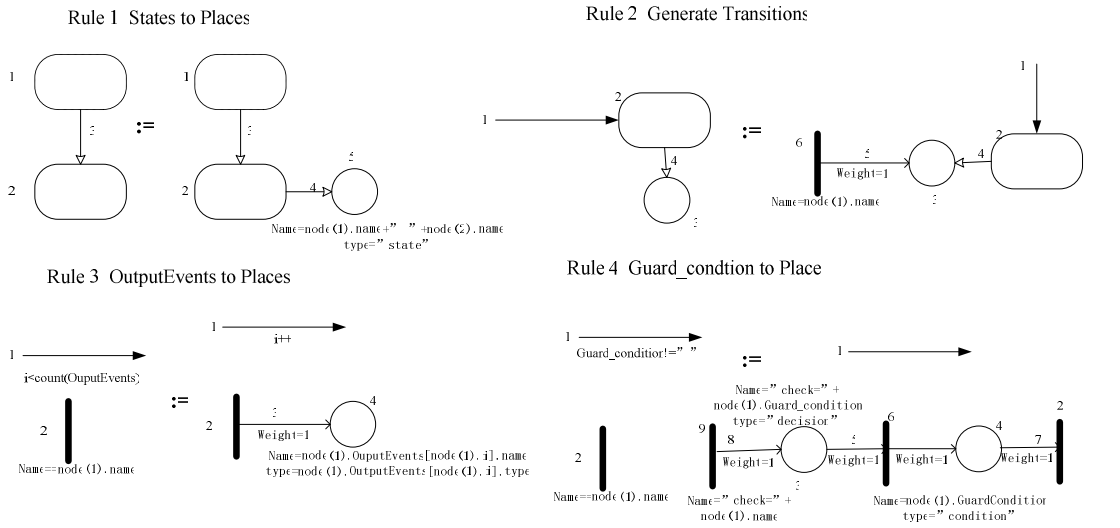

Fig. 11 Graph Transformation Rules 1-4

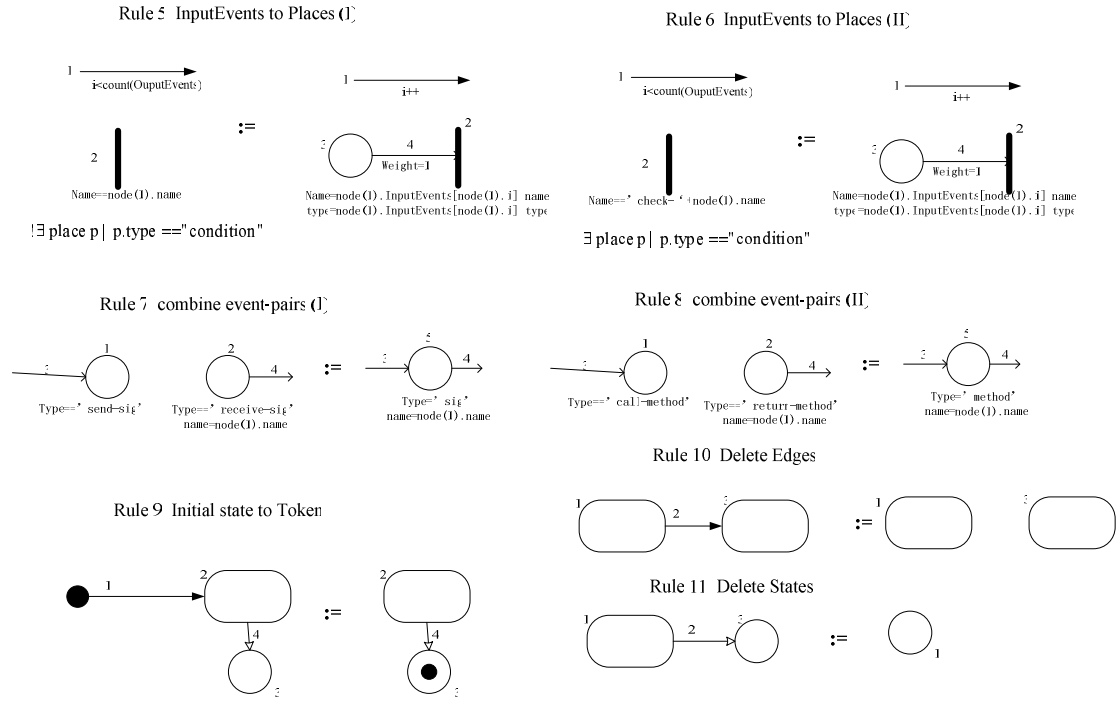

Fig. 12 Graph Transformation Rules 5-11

7. Conclusions and Future Work

Through our experiment, we demonstrate a meta-level technique to verify UML diagrams by transforming them into Petri-nets based on meta-modeling and graph transformation techniques. Besides the elements of every single diagram, we emphasize the relationships among the various UML diagrams in the contents for transformation and verification, and discussed the layers of these relationships and the potential target Petri-net models and analysis approaches. As a correct transformation, we regard the semantic correctness and effectiveness as the fundamental requirement for transformation for verification, besides the completeness, uniqueness, termination and syntactic

correctness. Due to the lack of the verification technique of the semantic correctness and effectiveness of transformation, we propose a debugging approach to modify the transformation rules according to the concrete semantic constraints through a case study. Although we have only conducted experiments on the verification of relatively simple UML statechart diagrams, the approach we have proposed can be easily adapted to more complex statechart diagrams and provide meaningful guidance for the verification of other UML diagrams.

In the experiment, we use transformation rules which are executed in sequential steps. In the future, we aim to research on the design of more complex process for rule execution in order to cut down the number of rules and verify our method through tool implementation. Furthermore, as the feedback of verification for UML models, the back-annotation technique from Petri-nets will be considered in the future work. Also, the other diagrams such as flow-based dynamic diagrams and the furthermore conjunction contents worthy for verification between the dynamic and static views will be experimented on.

References

[1]AtoM^3 home page: http://atom3.cs.mcgill.ca.

[2]de Lara, J. , Vangheluwe, H. 2002 AtoM^3: A Tool for Multi-Formalism Modeling and Meta-Modeling. In ETAPS/FASE'02, LNCS 2306, pp,:174-188. Springer.

[3]GROOVE home page: http://wwwhome.cs.utwente.nl/~groove/groove-index

[4]G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. VIATRA: Visual automated transformations for formal verification and validation of UML models. In Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering, Edinburgh, UK, September 23--27 2002.

[5]OMG. OMG Unified Modeling Language Specification 1.5. http://www.omg.org/uml/

[6]OMG. OMG Meta Object Facility (MOF) Specification 1.4. http://www.omg.org/mof/

[7]TADAO MURATA. Petri-nets: Properties, Analysis and Applications. Proceedings of the IEEE, VOL.77, NO, 4, April 1989.

[8]G. Rozenberg (ed.). Handbook of Graph Grammars and Computing by Graph Transformations: vol.1. Foundations. World Scientific, 1997.

[9]D. Varró, A. Pataricza. Automated Formal Verification of Model Transformations. Submitted to CSDUML 2003 Workshop on Critical Systems Development with UML, October 20-24, 2003, San Francisco, CA, USA. http://hobbit.inf.mit.bme.hu/FTSRG/Publications/uml03b_vp.pdf

[10]Lucinao Baresi, Reiko Heckel. Foundations and Applications of Graph Transformation: an Introduction from a software engineering perspective. Presentation at ICGT2002.

[11]Lucinao Baresi, Reiko Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. Proceedings of the First International Conference on Graph Transformation (ICGT2002, Lecture Notes in Computer Science, 2505), Springer, 402-429

[12]D. Varró. Towards automated formal verification of visual modeling languages by model checking. Journal of Software and Systems Modeling, 2003. Submitted to the Special Issue on Graph Transformation and Visual Modeling Techniques.

[13] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, Andrew Wood. Transformation: The Missing Link of MDA. Proceedings of the First International Conference on Graph Transformation (ICGT2002, Lecture Notes in Computer Science, 2505), Springer, 90-105