# UCVSC: A Formal Approach to UML Class Diagram Online Verification Based on Situation Calculus

Li Tan, Zongyuan Yang and Jinkui Xie
Department of Computer Science and Technology
East China Normal University
Shanghai, China
darkwhite29@gmail.com, {yzyuan, jkxie}@cs.ecnu.edu.cn

***Abstract*—The gap between informal models used in a UML environment and formal verifications and proofs in academic research prevents UML from valid and efficient application. In this paper, we propose an approach to bridge the gap between UML class diagram and situation calculus via our formal verification tool, UCVSC (UML Class diagram online Verification based on Situation Calculus). UML class diagram describes a software system informally while situation calculus is employed as the underlying formalism to precisely specify the system. With respect to most components in UML class diagram, the strength of reasoning about actions and describing the state of the world in situation calculus can be applied to represent them appropriately. Using UML tools and predefined mapping mechanism, we transform UML class diagram to XMI, an intermediate format, and finally to situation calculus in Prolog syntax. This approach attempts to provide precise semantics of UML class diagram which can be logically verified. In addition, we automate the verification process in an online prototype system. Furthermore, a case study on an academic system is presented to illustrate and evaluate our approach.**[*]

***Keywords-UCVSC; UML class diagram; XMI; situation calculus; online prototype system***

## I. INTRODUCTION

With evolution of the process of software development, the design phase of software engineering becomes greatly concerned. For its powerful model capability and general purpose, UML [1][2] is increasingly popular in the design process of object-oriented software system nowadays. Enhancing from the rapid-changed version and embracing some new concepts in software engineering nowadays, UML tries to keep pace with emerging requirements and gradually gets more accredited by industry.

### *A. The Problems*

UML has been bringing a revolution of software design generalization, whereas due to the informal graphical notation of UML, flaws cannot be found in the design phase but in the execution phase, which may cost a lot. Recently, to provide UML, the industrial de facto standard, firm foundation of formal semantics is in the spotlight and it is getting popular to analyze UML models from a viewpoint of formal methods such as formal specification [3] and formal verification [4]. As part of the modern software engineering, the UML Formal Verification (UFV) emphasizes the correctness and soundness checks of system modeling throughout the design process, and this behavior attempts to use formal methods, the precise mathematical fundamentals, to make up for the lack of formalism of UML itself. Hence, the inconsistency of different understanding to system design can be eliminated.

Using the components of a class and the relationships among classes, UML class diagram specifies the structure of a system by a static pathway and it is often regarded as the foundation of other UML diagrams such as UML statechart and collaboration. Obviously, though UML class diagram holds a concise definition and a small size, the information it contains is vital and indispensable to the whole design process. Thus, it is also important to make some verification on UML class diagram.

Although research related to UFV exists, a great deal of related work focuses on not the verification process but the mapping mechanism between UML models and formal languages, our work try to provide a more integrative and dynamic design environment for UML users, i.e., a web-based prototype system for UML class diagram formal verification is presented for UML users to dynamic verify their design draft.

### *B. How Situation Calculus Deals with These Problems*

In UCVSC, we focuses on situation calculus [5] from the perspective of describing UML graphic notation formally, especially class diagram. With respect to all aspects in UML class diagram, the strength of reasoning about actions and describing the state of the world in situation calculus can apply to represent them appropriately.

The reason why situation calculus is employed as the underlying formalism is given as follows. Though statically deployed as a whole, UML class diagram also has some dynamic features locally such as generalizations, dependencies and associations, the three main relationships between classes. As we know, the strength of situation calculus is its capability of representing and reasoning about actions. Moreover, the static ingredient in UML class

[*] This work is supported by National Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20060269002, Natural Science Fund of Shanghai Municipality under Grant No. 09ZR1409500 and Key Project in Basic Research of Science and Technology Commission of Shanghai Municipality under Grant No. 09JC1405000.

diagram can be described by situation element in a first-order way easily. Hence, it's sufficient for situation calculus to precisely specify the semantics in UML class diagram and the attempt to formalize UML class diagram by situation calculus is feasible logically.

The rest of this paper is organized into four sections. Firstly, the formalization of UML diagrams in previous work of others will be simply reviewed in Section 2. In Section 3, brief introduction to XMI and situation calculus will be given. Then, our formal approach and the implementation of an online prototype system are elaborated in Section 4. Finally, the conclusion and future work are put forwarded in Section 5.

## II. RELATED WORK

The formalization of UML diagrams has been widely studied [6][7][8]. Some of them are really good jobs and the core concepts they've proposed are accepted in a large scale. A formal framework is provided to support visual simulation of UML models composed of class, object, state, sequence and collaboration diagrams and an integrated semantics of these models is presented in [6]. However, it only focus on the semantics building and transformation rules of UML diagrams, but the verification of modeling process is not considered. Furthermore, a graphical yet formal approach is offered to specify the behavioral semantics of statechart diagrams using graph transformation techniques in [7]. Additionally, a powerful transformation tool was developed by Alexander Knapp's group [9], i.e., Hugo/RT, an excellent UML model translator for model checking, theorem proving, and code generation, is famous for its comprehensiveness in the variety of destination languages. The formal languages that Hugo/RT doesn't refer to are situation calculus and Pi-Calculus, etc.

The fact that UML models are represented in the XMI format greatly facilitates the transformations. To adapt to XMI, a web-based tool for the selection and invocation of design pattern evolutions represented by UML models is developed in [8]. Hence, a web interface is preferred in our prototype system, UCVSC. In this condition, verification service can be provided online and easy to access via the Internet.

## III. BACKGROUND KNOWLEDGE

### A. UML in an XMI Way

In this era of daily changing Internet and knowledge explosion, XMI (XML Metadata Interchange) [10], which attempts to describe system model in the syntax of XML, was introduced to express and communicate the design of software efficiently on the Internet. From the initiate idea of the introducer, XMI is a framework for defining, interchanging, manipulating and integrating XML data and objects, typically used as interchange format for UML models.

Currently, many tools have realized the interchange format transformation from UML to XMI. Rational Rose, Poseidon For UML and ArgoUML are excellent ones among them. Therefore, data model of system design can be communicated among different UML modeling tools and data warehouses based on MOF (Meta Object Facility) [11]. As a uniform expression of internal data models, XMI can separate the description of system from different design tools, which achieves the isolation between the design process and design tools and provides great convenience to save and load data models in a structured way.

In this paper, we choose ArgoUML, a lightweight and free UML tool, to model a target system and export the primitive UML class diagram into an XMI file. Then we parse this XMI file and transform the information needed into situation calculus automatically. In the final step, we use Prolog engine to verify the correctness of the modeling process in the design phase.

### B. Situation Calculus

Introduced by John McCathy in 1963 [5], situation calculus has been widely applied in Artificial Intelligence related research and other fields. This formalism is considered as a dialect of logic language and mostly used in dynamic domain modeling.

The key concepts in situation calculus include *action*, *situation* and *fluent*.

An *action* represents any possible change to the world, denoted by a function, for example, *drop(A)*, *clean(B)* and *check_in(ID)*.

A *situation* represents a possible world history, simply a sequence of actions, denoted by a first-order term. The constant S0 is used to denote *the initial situation*, namely, the empty sequence of actions [12].

A *fluent* represents a relation or a function whose truth values varies from one situation to the next, called *relational fluent* or *functional fluent* respectively. For example, *hunger_status(person, s)* and *weather_condition(location, season)*.

Additionally, introduce two predefined binary symbols as follows:

Function symbol *do* is defined as $do: Action \times Situation \rightarrow Situation$, which maps an action $a$ and a situation $s$ to a new situation called successor situation, which results from performing the action $a$ in the situation $s$. This successor situation is denoted as $do(a, s)$.

Predicate symbol *Poss* is defined as $Poss: Action \times Situation$. Similarly as above, $Poss(a, s)$ means it is possible to execute the action $a$ in the situation $s$.

## IV. THE PROTOTYPE SYSTEM BASED ON A FORMAL APPROACH

### A. Outline

So far we have briefly introduced and reviewed the core concepts of our idea of verification process of UML class diagram. To describe it more directly, the overall architecture of our prototype verification system, UCVSC, is shown in Figure 1.

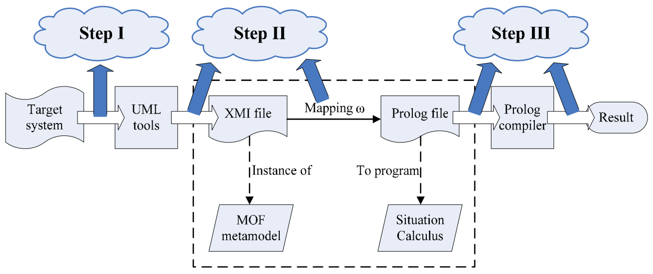


Figure 1. Outline of our online prototype system, UCVSC

**Step 1 (Manually):** *Design a target system model in UML tools.*

In this paper, ArgoUML 0.28 is used as a UML modeling tool in the above figure.

**Step 2 (Automatically):** *Using UML tools and mapping rules ω, transform from the model described in UML class diagram to the Prolog script in situation calculus.*

Using the formal mapping rules, it is convenient to define the corresponding model in another formalism and understand the relationships between the two types of notation.

The formal structure in UML class diagram is represented by χc, that in situation calculus by respectively, where the transformation is implemented by Perl.

**Step 3 (Automatically):** *Verify the model with Prolog compiler and give out result.*

We verify the model of the target system by SWI-Prolog and check if there are some conflicts among classes and their relationships or not.

## *B. An Academic System in UML Class Diagram and Formalization of It*

In this paper, we transform target system modeling by UML class diagram into situation calculus expression and verify the transformed model. In order to avoid unnecessary misunderstanding and ambiguous points of the modeling methods, we introduce the following six rules to design model of the target system in UML class diagram via the UML modeling tool ArgoUML 0.28.

**Rule 1:** *The whole content of the target system must only be described in the class diagram rather than other diagrams such as Use Case diagram.*

**Rule 2:** *The naming mechanism is in accordance with Java name rules.*

**Rule 3:** *For the simplicity, path type aggregation and composite are considered as two types of aggregation, shared aggregation and composite aggregation, respectively.*

**Rule 4:** *By default, each generalization has only name but no discriminator, and the return value type of each function in a class is void. We conform to the default.*

**Rule 5:** *Data type 'int' is the internal type of ArgoUML while data type 'string' is the build-in type in the class which uses it.*

**Rule 6:** *For the integrity and readability, the relationship between class 'Student' and classes 'UnderGraduate' and 'Graduate' is not generation but aggregation.*

As a case study, we show an academic system in university by terms of UML class diagram and formalize it by situation calculus as follows:

**Definition 1.** A formal structure of a UML class diagram in situation calculus is

$$\chi c = <Cl, As, G, Ag, Co, D>, \text{ where}$$

Cl: a finite set of classes,

*// 'Cl' stands for 'Class' in UML class diagram notation.*

As: C ↔ C, a bijection between two classes,

*// 'As' stands for 'Association' in UML class diagram notation.*

G: C → C, an injection from a child class to its parent class,

*// 'G' stands for 'Generalization' in UML class diagram notation.*

Ag: C → C, an injection from a part class to its shared aggregation class,

*// 'Ag' stands for 'Aggregation' in UML class diagram notation.*

Co: C → C, an injection from a part class to its composite aggregation class,

*// 'Co' stands for 'Composite' in UML class diagram notation.*

D: C → C, an injection from a friend class to its independent class.

*// 'D' stands for 'Dependency' in UML class diagram notation.*

Note: 'multiplicity' attributes of elements 'As', 'Ag' and 'Co' can be of 4 types: 0, 1, 0..* and 1..*.

Figure 2 shows the academic system model in UML class diagram.

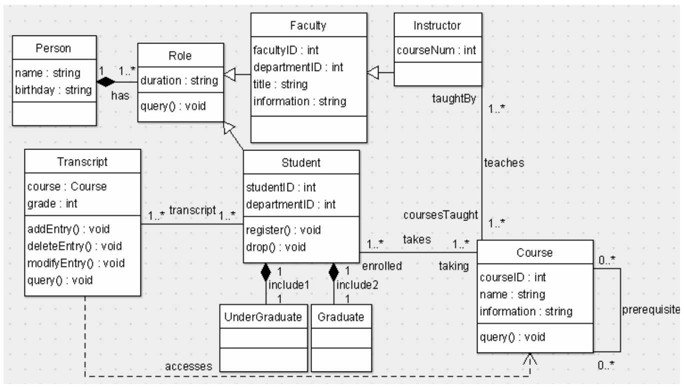


Figure 2. An academic system in UML class diagram

**Definition 2.** A mapping mechanism of between counterparts of UML class diagram and situation calculus can be defined in the following table:

TABLE I. MAPPING MECHANISM DEFINITION BETWEEN UML CLASS DIAGRAM AND SITUATION CALCULUS

| **UML class diagram** | **situation calculus** | **Comments** |
|---|---|---|
| Class | Functional Fluent | Functional Fluent may have a static effect. |
| Association | Relational | Both Relational |

| | | |
|---|---|---|
| | Fluent/Action | Fluent and Action can depict the association between classes appropriately. |
| Generalization | Action | Generalization is assumed not to change from one situation to another. |
| Aggregation | Action | Aggregation is assumed not to change from one situation to another. |
| Composition | Action | Composition is assumed not to change from one situation to another. |
| Dependency | Action | Dependency is assumed not to change from one situation to another. |

**Rule 7:** *As is hard and unnecessary, multiplicity attribute is not considered in situation calculus.*

**Definition 3.** A formal model transformation rules from basic elements of UML class diagram to Prolog script which implements situation calculus can be defined in the follow table:

TABLE II. TRANSFORMATION RULES DEFINITION AMONG UML CLASS DIAGRAM, XMI AND PROLOG SCRIPT

| **Elements in UML class diagram (in plain text)** | **XMI script** | **Prolog script** | **Comments** |
|---|---|---|---|
| Class 'Faculty' | UML:Class name = 'Faculty' | Class(Faculty, s) | Faculty is a Class. |
| Association 'takes' between Class Student and Course | UML:Association name = 'takes' UML:AssociationEnd.participant | takes(Student, Course, s) | Class 'Student' and Class 'Course' are associated with the action 'takes'. |
| Generalization from 'Instructor' to 'Faculty' | UML:Generalization name = 'FtoP' UML:Generalization.child | Inherit(Instructor, Faculty, s) | Class 'Instructor' inherits Class 'Faculty'. |
| | | | |
| Composition 'include' from 'Graduate' to 'Student' | UML:Association name = 'include1' UML:AssociationEnd aggregation = 'composite' UML:AssociationEnd.participant | Comp(Graduate, Student, s) | Class 'Graduate' is a part class to compose its composite aggregation Class 'Student'. |
| Dependency 'accesses' from 'Transcript' to 'Course' | UML:Dependency name = 'accesses' UML:Dependency.client UML:Dependency.supplier | Dep(Transcript, Course, s) | Class 'Transcript' is a friend class of Class 'Course'. i.e., the former can access the data and operation defined in the latter. |

According to the mapping mechanism and transform rules defined above, when parsing the XMI file transformed from UML class diagram of the target system, we propose the following algorithm to formally transform XMI script to situation calculus specification in Prolog in a compact format (Since the others are similar, only the algorithm for the path type of Generalization in UML class diagram is given out.):

***// Algorithm for Generalization: transform XMI script to Prolog script:***

```
Procedure transform_generalization()
1 for each UML.Model in XMI.content in XMI
2   extract attributes.name;
3   gen_exist = false; // global variable
4   for each UML.Class
5     extract attributes.name;
6     print "Class(attributes.name, s)";
7     if(UML.GeneralizableElement.generalization != null)
8       gen_exist = attributes.id;
9       for each UML.Attribute
10        extract attributes.name;
11        for each UML.Operation
12          extract attributes.name;
13          if(gen_exist != false)
14            for each UML.Generization
15              for each UML.Generization.child
16                extract child.name; // class_id to retrieve
17              for each UML.Generization.parent
18                extract parent.name; //class_id to retrieve
19              print "Inherit(child.name, parent.name, S)";
end.
```

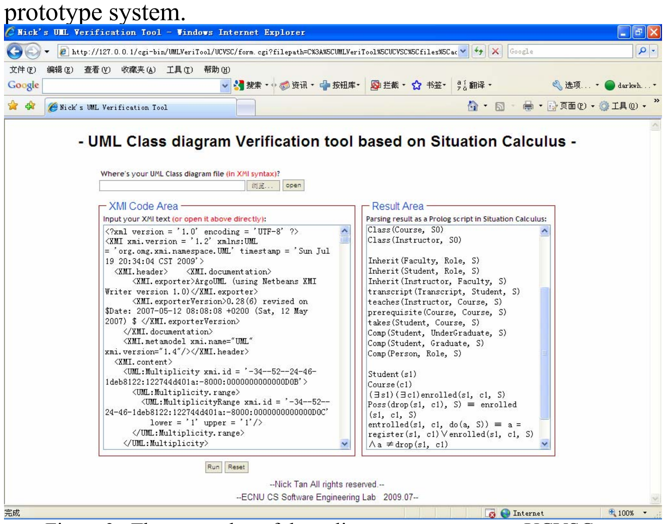
From this algorithm, we can see that though not explicitly referred, the precondition axioms and successor state axioms in situation calculus are embodied in the lines of Prolog script.

### C. Implementation of the Prototype System and Verification of the Model

Based on the structure and function of different parts described in Subsection 4.2, the prototype system of UML class diagram verification tool is implemented by Perl.

After reading the XMI file generated by ArgoUML, the verification tool parses every node, extracts and then transforms the information needed to Prolog script. Finally, the tool calls SWI-Prolog to compile this Prolog script and displays the result given out by SWI-Prolog.

Figure 3 shows the user interface of the implemented prototype system.


Nick's UML Verification Tool - Windows Internet Explorer
http://127.0.0.1/cgi-bin/UMLVeriTool/UCVSC/form.cgi?filepath=C%3A%5CUMLVeriTool%5CUCVSC%5Cfiles%5Cac
Run
Reset
open

- UML Class diagram Verification tool based on Situation Calculus -

Where's your UML Class diagram file (in XMI syntax)?

XMI Code Area

Input your XMI text (or open it above directly):

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML
= 'org.omg.xmi.namespace.UML' timestamp = 'Sun Jul
19 20:34:04 CST 2009'>
  <XMI.header>    <XMI.documentation>
      <XMI.exporter>ArgoUML (using Netbeans XMI
Writer version 1.0)</XMI.exporter>
      <XMI.exporterVersion>0.28(6) revised on
$Date: 2007-05-12 08:08:08 +0200 (Sat, 12 May
2007) $ </XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML"
xmi.version="1.4"/></XMI.header>
  <XMI.content>
    <UML:Multiplicity xmi.id = '-34--52--24-46-
1deb8122:122744d401a:-8000:000000000000D0B'>
      <UML:Multiplicity.range>
        <UML:MultiplicityRange xmi.id = '-34--52--
24-46-1deb8122:122744d401a:-8000:000000000000D0C'
          lower = '1' upper = '1'/>
      </UML:Multiplicity.range>
    </UML:Multiplicity>
```

Result Area

Parsing result as a Prolog script in Situation Calculus:

```
Class(Course, S0)
Class(Instructor, S0)

Inherit(Faculty, Role, S)
Inherit(Student, Role, S)
Inherit(Instructor, Faculty, S)
transcript(Transcript, Student, S)
teaches(Instructor, Course, S)
prerequisite(Course, Course, S)
takes(Student, Course, S)
Comp(Student, UnderGraduate, S)
Comp(Student, Graduate, S)
Comp(Person, Role, S)

Student(s1)
Course(c1)
(∃s1)(∃c1)enrolled(s1, c1, S)
Poss(drop(s1, c1), S) ≡ enrolled
(s1, c1, S)
entrolled(s1, c1, do(a, S)) ≡ a =
register(s1, c1)∨enrolled(s1, c1, S)
∧a ≠drop(s1, c1)
```

--Nick Tan All rights reserved.--

--ECNU CS Software Engineering Lab 2009.07--

Figure 3. The screenshot of the online prototype system, UCVSC

If we use SWI-Prolog to compile the Prolog script generated as above in Figure 3, it will pass on the premise that the design of the academic system described in Subsection 4.2 is correct. What if the design is bad? Now, let's take a look at another design version of the same academic system in Figure 4:

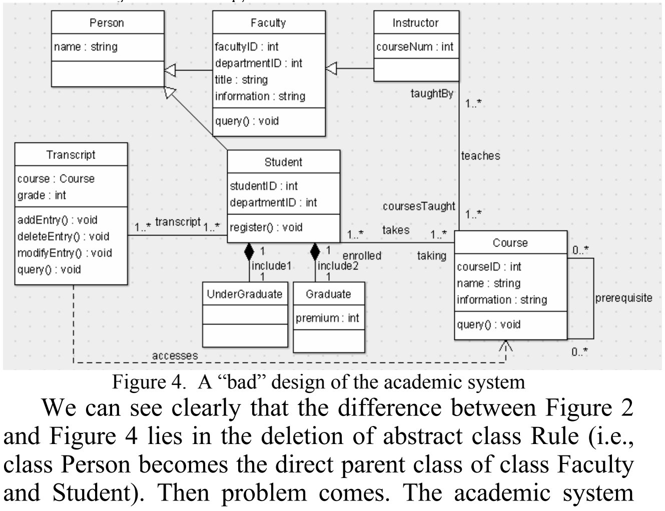


Figure 4. A "bad" design of the academic system

We can see clearly that the difference between Figure 2 and Figure 4 lies in the deletion of abstract class Rule (i.e., class Person becomes the direct parent class of class Faculty and Student). Then problem comes. The academic system described in Figure 4 does not permit a teacher to be a student. As a matter of fact, it's a commonplace for a teacher to get further study in the same university. Therefore, it's incorrect and unreasonable to delete abstract class Role in Figure 4. Fortunately, with the help of Prolog compiler, the Prolog script as follows cannot pass:

***// A student and teacher case in Prolog script***

```
...
Course(CS01, S0);
Course(CS04, S0);
...
Student(Nick, S0);
...
take(Nick, CS04, S);
teach(Nick, CS01, S);
```

With the help of UCVSC, the formal verification tool for UML class diagram we develop, software architecture engineers who design a software system can find the incorrectness in this system in advance, rather than to re-design when codes have been written and some problems have been found then.

## V. Conclusion and Future Work

In order to eliminate ambiguity among different viewpoints of system design and ensure completeness, people attempt to use formal methods to specify and verify the UML diagrams. In this paper, we focus on a formal verification way for UML class diagram and implement corresponding online prototype system, UCVSC, to deal with the verification task and thus firmly support our idea. The core concepts in our approach are as follows: Since XMI is an XML-based format which has been widely used for model interchange in UML tools, we transform pre-drawn UML Class to XMI format via UML modeling tool itself automatically. Then according to the mapping mechanism we design, XMI is transformed to a formal and verifiable language, situation calculus in Prolog syntax automatically. Finally, we use a Prolog compiler, SWI-Prolog, to verify the correctness of the Prolog script generated automatically by our online prototype system.

There is still much work for further research. In current version of UCVSC, the final step is not automatic but manual. We hope to call the Prolog Compiler in codes and then display the verification result on the web page directly, which can make our online prototype system more integrated and easier to manipulate. Moreover, most concepts in UML class diagram are referred to in our approach but not the whole. We can further define and extend our mapping mechanism. In addition, we only implement basic verification for UML class diagram. Other UML diagram can also be considered to verify. Virtually, another verification platform for more UML diagrams and more formal languages is the next item on our agenda. In addition, more intelligent facts such as an accessibility of verification options configurable by users will be concerned in our following work.

## References

[1] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Pearson Higher Education, Boston, 2005.

[2] OMG: UML 2.0 Superstructure Specification, version ptc/04-10-02. Object Management Group, Inc., Needham, MA, http://www.omg.org, 2004.

[3] M. Gogolla, F Büttner. M. Richters, "USE: A UML-based specification environment for validating UML and OCL," Science of Computer Programming, vol. 69, pp. 27–34, 2007.

[4] R.V.D. Straeten, T. Mens, J. Simmonds, V. Jonckers, "Using description logic to maintain consistency between UML models," LNCS, vol. 2863, pp. 326–340, Springer, Heidelberg, 2003.

[5] J. McCarthy, "Situations, actions and causal laws," Stanford Artificial Intelligence Project, Memo 2, 1963.

[6] M. Gogolla, P. Ziemann, S. Kuske, "Towards an Integrated Graph Based Semantics for UML," Electr. Notes Theor. Comput. Sci, vol. 72, 2003.

[7] J. Kong, K. Zhang, J. Dong, D. Xu, "Specifying Behavioral Semantics of UML Diagrams through Graph Transformations," The Journal of Systems and Software, vol. 82, pp. 292–306, 2009.

[8] J. Dong, Y. Zhao, Y. Sun, "XSLT-based evolutions and analyses of design patterns". Software: Practice and Experience. vol. 39, pp. 773–805, 2009.

[9] A. Knapp, S. Merz, Hugo/RT, http://www.pst.ifi.lmu.de/projekte/hugo/, 2004.

[10] OMG, XML Metadata Interchange, version 1.2, Object Management Group, Inc., Needham, MA, http://www.omg.org/, 2002.

[11] OMG, Meta Object Facility Specification, version 1.4, Object Management Group, Inc., Needham, MA, http://www.omg.org/, 2002.

[12] B. Li, J. Iijima, "A Survey on Application of situation calculus in Business Information Systems," Proc. International Conference on Convergence Information Technology (ICCIT 07), pp. 425–431, 2007.