Proceedings of the IASTED International Conference
Software Engineering (SE 2011)
February 15 - 17, 2011 Innsbruck, Austria

VERIFYING UML IN PROLOG*

Li Tan, Zongyuan Yang and Jinkui Xie,
Department of Computer Science and Technology,
East China Normal University
200241 Shanghai, China
darkwhite29@gmail.com, {yzyuan, jkxie}@cs.ecnu.edu.cn

ABSTRACT
From the viewpoint of software life cycle in Software Engineering, software architecture is the core of the structure and behavior of software. As software architecture design itself is a kind of modeling activity, how to verify the correctness of the standard modeling language of software architecture design, UML, is one big problem. This paper used an implementation of situation calculus, Prolog, as an underlying logic framework to formally describe a subset of UML diagrams, a graphic notation, and assign it an equivalent formal semantics automatically via our prototype tool, USCVSC. Then we verified possible syntax and semantic errors in UML in a Prolog interpreter, SWI-Prolog, after a definition of error types in UML. Finally, a goal was achieved that software designers could correct the design of UML diagrams prior to fixing bugs in code in the phase of test, which avoided unnecessary system overhead in the following phases of Software Engineering with the hope of enhancing the overall efficiency of the process of Software Engineering.

KEY WORDS
UML; situation calculus; Prolog; formal verification; automatic generation

1. Introduction

From the point of view of the software life cycle in Software Engineering, software architecture is the core of the structure and behavior of software. Thus software architecture design is bound to be the core of the software design, and the basis for the subsequent code development as well. The significance of software architecture design is self-evident. As software architecture design itself is a kind of modeling activity, one problem is raised: how to model precisely and correctly? That is, the model should be built without ambiguity and lost information. UML (Unified Modeling Language) [1], the standard modeling language of software architecture design, is a kind of graphic notation which hardly has formal semantics, resulting lack of preciseness and correctness. Therefore, for mechanical verification, some formal methods are employed to describe UML mathematically and assign it an equivalent formal semantics in formal languages, since manual verification is not precise and complete to ensure the quality of verification of UML. Thus, a lot of formal methods and corresponding tools have been proposed in recent three decades. As formal verification tools, there are theorem provers like ACL2 [1] and Coq [2], etc, as well as model checkers like SMV [3] and SPIN [4], etc. As a matter of fact, the input of these tools is program written in formal languages defined in underlying formal methods of these tools, not a graphic notation just like UML. Hence, if UML needs to be verified, firstly, its graphic notation should be translated into a recognizable input of the formal verification tools mentioned, and then verification steps can be conducted after importing the recognizable input into these tools. In this way, the preciseness and correctness of UML diagrams can be verified, which avoids costly testing and maintaining work of code in the following phase of implementation.

The rest of this paper is organized into four sections. Firstly, our solutions to UML verification and the previous work of others will be simply reviewed in Section 2. In Section 3, a formal semantics of UML class diagram and state diagram and a definition of possible syntax and semantic errors in UML will be given. Then, our approach of verifying UML design errors in Prolog is elaborated in Section 4. Finally, the conclusion and future work are put forwarded in Section 5.

2. Solutions to the Problem

As the evolution of software development methods, the design phase is becoming an indispensable part in Software Engineering. Concerns about the quality of software design are in the spotlight recently. On the other hand, due to its powerful modeling ability and general purpose, UML has been recognized as the de facto standard of modeling language for object-oriented software design [5].

UML has some welcome characteristics like user friendliness and interactiveness, which benefits from the fact that UML is a modeling language based on graphic notation, illustrative and easy to use. However, due to lack of preciseness, this informal notation may cause not

* This work is supported by grants from 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.

only ambiguity, but also flaws and bugs of software design. Generally, these three kinds of errors cannot be found until the phase of software testing. When errors are found, design and code can then be corrected, which is of high cost, since according to statistics, the overhead of software testing amounts to 50% or above of the whole overhead of Software Engineering [6]. More importantly, software testing is not a strict method to verify software design, since it sometimes cannot be exhaustive due to the reason of practicality. As Edsger W. Dijkstra said, program testing can be used to show the presence of bugs, but never to show their absence [7].

Overall, in this era full of severe software reliability and security problems, verifying standard modeling language of software design, UML, by means of assigning it a formal semantics and getting aid from corresponding formal verification tools, is of great necessity. Only in this way, errors in UML diagrams can be located efficiently, and then fixed in the phase of software design rather than software testing to avoid unnecessary system overhead.

2.1 Our Approach

We introduce a general framework for formalizing a subset of UML diagrams, class diagram and state diagram, based on the description of UML in a formal language, situation calculus [8].

The basic route of our approach is illustrated as Figure 1 (The area in dash line is the core component of our work).

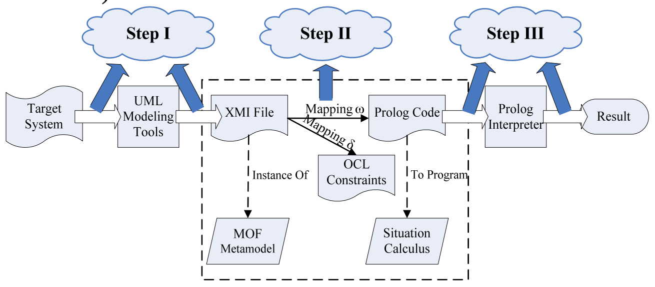


Figure 1. Basic Route of Our Approach

Step I: Design a target system in UML with the help of UML tools; Transform a subset of UML diagrams to a temporary file represented in the standard notation of data on the Internet, XMI (XML Metadata Interchange) [9].

Step II: Using predefined mapping rules ω, transform the XMI file generated in last step to situation calculus code in Prolog syntax (Note that mapping δ is not discussed in this paper); meanwhile, the domain-unrelated syntax errors in UML diagrams are checked as well.

Step III: Import the Prolog code generated into a Prolog interpreter, SWI-Prolog, as our formal verification tool; analyze and query some significant elements to verify the domain-related semantic errors in UML diagrams.

Note: Step I is conducted by a software architecture designer manually. Step II and Step III are fulfilled automatically by our prototype tool, USCVSC, illustrated in Section 4 and SWI-Prolog jointly.

So far we have briefly introduced the core concepts of our approach of verifying UML in Prolog. Our prototype system of UML verification is implemented in a Web platform by Perl and CGI, because Web applications are of excellent portability and easy to use. Furthermore, the clients only need a browser to run the applications, without any software installed. And as we know, the strength of Perl is its outstanding capability of text parsing and processing [10] which is suitable for XMI file analysis. Additional reason of the employment of Perl is its mature support for Web application and available modules in open source communities of Perl.

2.2 Related Work

Since introduction of UML, research on UML formalism has never ceased and always plays an important role on the development and evolution of UML. Researchers attempt to assign graphic UML an equivalent formal semantics in a formal language based on Mathematical Logic, in order to eliminate the underlying ambiguity and errors. The birth of OCL is a vivid proof. As a standard sub-language of UML, OCL can make up for what UML cannot present and also provide UML a precise description of systems to be modeled in semantics [11].

Apart from OCL, there is lots of other research on formalizing UML and related work. Diego Latella, et al firstly realized a formal transformation from UML state diagram to PROMELA, the input language of model checker SPIN via Hierarchical Automata, and a proof of correctness of this transformation was given as well. Finally they used SPIN to verify the safety and liveness of the generated PROMELA model [12]. Moreover, the pUML project [13] led by Andy Evans and the STL project [14] led by Juergen Dingel stressed on the formalism of UML semantics by transforming UML diagrams to formal specifications, but supporting tools for practical use are few. On the other hand, the Hugo/RT project [15] led by Alexander Knapp made a big breakthrough by extending single UML diagram to multiple ones. In addition, Hugo/RT is able to transform several UML diagrams to the input of many kinds of targets: model checkers, theorem provers and even code of High-level Programming Language such as Java. Hugo/RT performs well thanks to its general purpose.

Despite these ways of UML verification, we take situation calculus based on Basic Action Theory [16] into account. Basic concepts of situation calculus are fundamentals of First Order Logic and Set Theory in Mathematical Logic. From this point of view, it is self-evident that situation calculus has a kind of strength of action-based reasoning which is appropriate to describe some UML diagrams such as class diagram and state diagram. Nevertheless, little attention is put on formal language good at dynamic modeling and reasoning like situation calculus. Therefore, we try to verify a subset of UML diagrams in Prolog, an implementation of situation calculus.

3. UML Formal Specification in Prolog and Errors Definition

In this section, we formally describe a subset of UML diagrams, class diagram and state diagram in Prolog, as an implementation of the formal language, situation calculus, and corresponding transformation algorithms are put forward as well.

As we know, UML diagrams consist of two sub-kinds: structure diagram and behavior diagram. In the taxonomy of them, we could find class diagram and state diagram (also called state machine diagram or statechart diagram) are the core branches of structure diagram and behavior diagram, respectively. Moreover, practically, they form the basis of other diagrams as well. It is more significant to research fundamental diagrams in UML. Thereby, class diagram and state diagram are chosen as research targets in this paper.

3.1 Semantics of UML Class Diagram

Definition I: A formal semantic definition of UML class diagram is

χc ::= <**Name**, **Cl**, **As**, **Gen**, **SharedAg**, **CompAg**, **Dep**>, where

Name: the name of a class diagram. It could be void.

Cl: a finite set of all classes in the class diagram. That is, an element of the set, c1, is a class, denoted as $c1 \in$ **Cl**.

As: a finite set of all Association relationships between two classes in the class diagram. That is, an element of the set is a bijection, like $c1 \leftrightarrow c2$, denoted as $(c1, c2) \in$ **As**, iff there is an Association relationship between class c1 and c2.

Gen: a finite set of all Generalization relationships between a child class and its parent class in the class diagram. That is, an element of the set is an injection, like $c1 \rightarrow c2$, denoted as $(c1, c2) \in$ **Gen**, iff there is a Generalization relationship between class c1 and c2.

SharedAg: a finite set of all Shared Aggregation relationships between a part class and its shared aggregation class in the class diagram. That is, an element of the set is an injection, like $c1 \rightarrow c2$, denoted as $(c1, c2) \in$ **SharedAg**, iff class c1 is a part class and class c2 is its shared aggregation class.

CompAg: a finite set of all Composite Aggregation relationships between a part class and its composite aggregation class in the class diagram. That is, an element of the set is an injection, like $c1 \rightarrow c2$, denoted as $(c1, c2) \in$ **CompAg**, iff class c1 is a part class and class c2 is its composite aggregation class.

Dep: a finite set of all Dependency relationships between a client class and its supplier class in the class diagram. That is, an element of the set is an injection, like $c1 \rightarrow c2$, denoted as $(c1, c2) \in$ **Dep**, iff class c1 is a client class of supplier class c2, i.e., class c1 is dependent on class c2.

3.2 Mapping Mechanism between UML Class Diagram and Situation Calculus

A mapping mechanism of counterparts between UML class diagram and situation calculus is defined in Table 1:

Table 1. Mapping Mechanism Definition between UML Class Diagram and Situation Calculus

Elements of UML Class Diagram	*Elements of Situation Calculus*	*Comments*
Cl	Functional Fluent	Functional Fluent also has a static effect; i.e., a definition of a class can be considered as a functional fluent, like Class(c1).
As	Relational Fluent /Action	Both Relational Fluent and Action can depict Association relationships between classes appropriately.
Gen	Action	Generalization relationship is assumed not to change from one situation to another so it can be signified as a function, which satisfies Action.
SharedAg	Action	Shared Aggregation relationship is assumed not to change from one situation to another so it can be signified as a function, which satisfies Action.
CompAg	Action	Composite Aggregation relationship is assumed not to change from one situation to another so it can be signified as a function, which satisfies Action.
Dep	Action	Dependency relationship is assumed not to change from one situation to another so it can be signified as a function, which satisfies Action.

Due to the page limit and similarity between situation calculus and Prolog, specific transformation from UML class diagram to Prolog code will not be given here, but a transformation algorithm.

The six elements discussed in this paper are class, Association relationship, Generalization relationship, Shared Aggregation relationship, Composite Aggregation relationship, Dependency relationship. Due to the page limit and similarity, only the most typical transformation algorithm for Generalization relationship is presented here.

Algorithm 1: Transforming Generalization in UML Class Diagram to Prolog Code

```
Procedure transform_gen_cd($G_i$)
  1: for each UML.Model Î XMI.content$_j$ do
  2:   gen_exist ← false; // global variable
  3:   for each UML.Class$_k$ Î UML.Model do
  4:     extract(Class$_k$.name);
  5:     addProlog(Class$_k$);
  6:       if (UML.GeneralizableElement.generalization != null) then
```

```
            gen_exist ← Class_k.id;
            for each UML.Attribute_s Î UML.Class_k do
              extract(Attribute_s.name);
            end for
            for each UML.Operation_t Î UML.Class_k do
              extract(Operation_t.name);
            end for
          end if
        end for
        if (gen_exist != false) then
          for each UML.Generalization_u Î UML.Model do
            extract(Generalization_u.child.name);
            extract(Generalization_u.parent.name);
            addProlog(Generalization_u);
          end for
        end if
      end for
```

3.3 Semantics of UML State Diagram

Definition II: A formal semantic definition of UML state diagram is
χs ::= <**Name**, **St**, **Starting**, **Ending**, **Tg**, **Gd**, **At**, **Ts**>, where
Name: the name of a state diagram. It could be void.
St: a finite set of all states (except the starting and ending states) in the state diagram. That is, an element of the set, st1, is a state, denoted as st1 ∊ **St**, with the fact that st1 is a triple <**name**, **stVar**, **atv**>, denoted as st1 ::= <**name**, **stVar**, **atv**>, where **name** is the name of the state; **stVar** is the state variable and **atv** is the activity in the state.
Starting: the starting state in the state diagram, denoted as **Starting**(**name**), where **name** is the name of the starting state. Generally, there is only one starting state in a state diagram.
Ending: a finite set of all ending states in the state diagram. That is, an element of the set, e1, is an ending state, denoted as e1(**name**), where **name** is the name of the ending state.
Tg: a finite set of all *triggers* which trigger transitions in the state diagram. That is, an element of the set, tg1, is a trigger, denoted as tg1 ∊ **Tg**, with the fact that tg1 is a binary tuple <**name**, **type**>, denoted as tg1 ::= <**name**, **type**>, where **name** is the name of the trigger; there are four types of the parameter **type**: 'calling', 'changing', 'time' and 'signal', i.e., **type** ∊ ('calling', 'changing', 'time', 'signal').
Gd: a finite set of all *Guards* which make corresponding transitions succeed in the state diagram. That is, an element of the set, gd1, is a guard, denoted as gd1 ∊ **Gd**, with the fact that gd1 is a binary tuple <**name**, **boolExp**>, denoted as gd1 ::= <**name**, **boolExp**>, where **name** is the name of the guard and **boolExp** is the Boolean Logic expression of the guard.
At: a finite set of all *Actions* activated when transitions happen in the state diagram. That is, an element of the set, at1, is an action, denoted as at1 ∊ **At**, with the fact that at1 is a binary tuple <**name**, **boolExp**>, denoted as at1 ::= <**name**, **boolExp**>, where **name** is the name of the guard and **boolExp** is the Boolean Logic expression of the action.
Ts: a finite set of all *Transitions* between states in the state diagram. That is, an element of the set, ts1, is a transition, denoted as ts1 ∊ **Ts**, with the fact that ts1 is a quintuple <**name**, **tg**, **gd**, **stSrc**, **stTg**>, denoted as ts1 ::= <**name**, **tg**, **gd**, **stSrc**, **stTg**>, where **name** is the name of the transition; **tg** is the trigger which triggers the transition; **gd** is the guard which make its transition succeed; **stSrc** is the source state of the transition and **stTg** is the target state of the transition.
Note: The starting state and the ending states are ones of all states in a state diagram. The reason why they are listed separately from the other normal states is in most cases, the starting state and the ending states have no parameters **stVar** and **atv**, and transitions involved in the starting state and the ending states have no parameters **tg** and **gd** as well, which differentiate from the normal states in structure.

3.4 Mapping Mechanism between UML State Diagram and Situation Calculus

A mapping mechanism of counterparts between UML state diagram and situation calculus is defined in Table 2.

Table 2. Mapping Mechanism Definition between UML State Diagram and Situation Calculus

Elements of UML State Diagram	*Elements of Situation Calculus*	*Comments*
St	Functional Fluent	Functional Fluent also has a static effect; i.e., a definition of a state can be considered as a functional fluent, like State(st1).
Tg	Action	Trigger is assumed not to change from one situation to another so it can be signified as a function, which satisfies Action.
Gd	Action	Guard is assumed not to change from one situation to another so it can be signified as a function, which satisfies Action.
At	Action	Action is assumed not to change from one situation to another so it can be signified as a function, which satisfies Action.
Ts	Relational Fluent /Action	Both Relational Fluent and Action can depict transitions between states appropriately.

Note: There is no starting state and ending states in this table, since the starting state, the ending states and transition involved are relatively simple. For the simplicity, the starting state and the ending states are considered as the normal states in category.

The transformation algorithm for transition (trigger, guard and action also included) in UML state diagram to Prolog code is presented here.

Algorithm 2: Transforming Transition in UML State Diagram to Prolog Code (Trigger, Guard and Action Included)

```
Procedure transform_trst_sd(T$_i$)
 1: for each UML.Model Î XMI.content$_j$ do
 2:   index ← 0; // global variable
 3:   trstArray[] ← null; // global variable
 4:   for each UML.Transition$_k$ Î UML.Model do
 5:     extract(Transition$_k$.name);
 6:     trstArray[index++][1] ← Transition$_k$.name;
 7:     for each UML.Guard$_s$ Î UML.Transition$_k$ do
 8:       extract(Guard$_s$.name);
 9:       trstArray[index-1][2] ← Guard$_s$.name;
10:       addProlog(Guard$_s$);
11:     end for
12:     for each UML.Action$_t$ Î UML.Transition$_k$ do
13:       extract(Action$_t$.name);
14:       trstArray[index-1][3] ← Action$_t$.name;
15:     end for
16:     for each UML.Trigger$_u$ Î UML.Transition$_k$ do
17:       trstArray[index-1][0] ← Trigger$_u$.id;
18:     end for
19:   end for
20:   for each UML.Event$_v$ Î UML.Model do
21:     if (Event$_v$.id == trstArray[index-1][0]) then
22:       addProlog(Transition$_k$, Trigger$_u$, Guard$_s$, Action$_t$);
23:     end if
24:   end for
25: end for
```

3.5 Error Types Definition in UML

When designing a target system by UML, software designers often make mistakes due to a mismatch of UML syntax rules or violation against basic logic. It is hard to discover these errors in the design phase until code is written. With the help of a compiler, bugs in code can then be found. We attempt to find UML syntax and semantic error directly in the design phase rather than the test phase to avert unnecessary overhead. After previous transformation from a subset of UML to Prolog code, an interpretation of this Prolog code can lead to a discovery of design errors in UML diagrams. Before the UML verification step, several typical errors will be defined in advance to make some preparation for a reasonable verification.

A. Syntax Errors

As is said before, UML syntax errors are domain-unrelated. That is, this type of errors can be typically called Non-requirement-oriented Errors. Some typical syntax errors are given here in UML class diagram and state diagram.

a) Circular Inheritance Error

When a UML class diagram is designed, it is possible the Generalization relationship among three or more classes forms a closed circle. If this kind of relationship occurs, this design of UML class diagram violates the syntax rules of UML, called a ***Circular Inheritance Error***.

b) Same Name Conflict Error

If in a UML model, two elements have same name and same type as well, a conflict occurs since these two elements can be told from by no means. This design of UML diagrams violates the syntax rules of UML, called a ***Same Name Conflict Error***.

c) Multiple Starting States Error

According to the formal semantic definition of UML state diagram in Section 3.3, there is only one starting state in UML state diagram. That is, for each object, there is only one entrance of transition of states. If there are multiple starting states in a UML state diagram, this design of UML state diagram violates the syntax rules of UML, called a ***Multiple Starting States Error***.

d) Guard's Boolean Expression Contains No Logical Operator Error

In the process of transition between states in UML state diagram, it is possible guard exists. According to its definition, guard is a logic condition which makes its transition succeed. i.e., Guard should be in terms of a Boolean expression. Therefore, except trivial Boolean constants “True” and “False” (They may have different forms), it is necessary that guard contains one or more logical operators defined in Boolean expressions. Otherwise, this design of UML state diagram violates the syntax rules of UML, called a ***Guard's Boolean Expression Contains No Logical Operator Error***.

B. Semantic Errors

As mentioned above, UML semantic errors are domain-related. This type of errors occurs due to the incomplete requirements of system as an issue of completeness or the requirements themselves have some basic logic errors as an issue of correctness. That is, this type of errors can be typically called Requirement-oriented Errors. Some typical semantic errors are given here in UML class diagram and state diagram.

a) Completeness: Requirements Not Complete Error

Because of mistakes in the previous stage of Software Engineering, such as the problem definition, feasibility analysis and requirement analysis stages, if there is some lost significant information in UML diagrams which brings about considerable flaws of system design, this design of UML diagrams contains a semantic error, called a ***Requirements Not Complete Error***.

b) Correctness: Requirements Logic Error

Because of wrong understanding or judgment in the previous stage of Software Engineering, such as the problem definition, feasibility analysis and requirement analysis stages, if there is some wrong significant information in UML diagrams which brings about considerable logic errors of system design, this design of UML diagrams contains a semantic error, called a ***Requirements Logic Error***.

In the next section, specific cases of UML design errors will be illustrated, and a demonstration on how to verify these errors in a Prolog interpreter will also be shown.

4. UML Design Errors Verifying in Prolog

According to the previous discussion, possible UML design errors in a detailed case of designing of a target system, a University Academic System, are to be verified via our prototype tool, USCVSC, an acronym of **U**ML **S**tate/**C**lass diagram **V**erification tool based on **S**ituation **C**alculus.

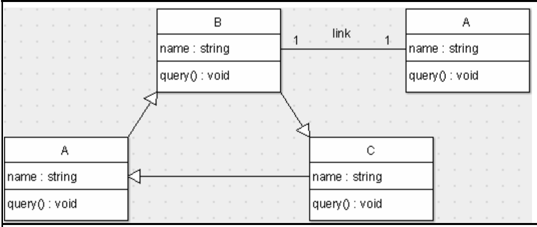
4.1 Syntax Errors Checking

Firstly, syntax errors checking is fulfilled in our prototype tool, USCVSC. By means of analysis of generated Prolog code, USCVSC will find all UML syntax errors defined in Section 3.5 and print an error warning, error type and line included as well. Due to the page limit and similarity, only more typical cases: Same Name Conflict Error and Guard's Boolean Expression Contains No Logical Operator Error are presented here.

A. Same Name Conflict Error

As for the Same Name Conflict Error in UML diagrams, Checking details are given in Table 3.

Table 3. Same Name Conflict Error Checking



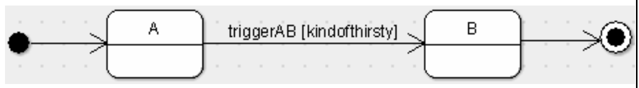
A Case of Same Name Conflict Error

```
Interpreting...
CI_SNC.xmi
Checking Time: 0.074519 seconds

Error(Line 8): There is an invalid circular
inheritance among class "A", "B" and "C".
Error(Line 4): There is an invalid same name
conflict on "class(A)".
CI_SNC.xmi - 2 error(s), 0 warning(s)
```

B. Guard's Boolean Expression Contains No Logical Operator Error

As for the Guard's Boolean Expression Contains No Logical Operator Error in UML state diagram, Checking details are given in Table 4.

Table 4. Guard's Boolean Expression Contains No Logical Operator Error Checking



A Case of Guard's Boolean Expression Contains No Logical Operator Error

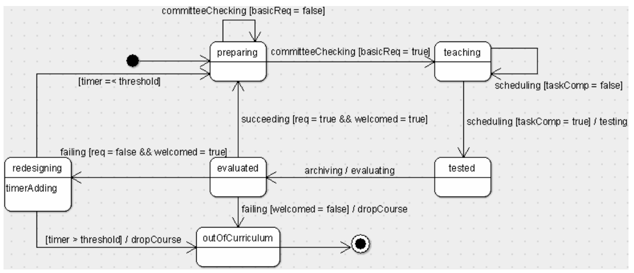
```
Interpreting...
Guard No Logical Operator.xmi
Checking Time: 0.086764 seconds

Error(Line 4): There is an illegal guard's Boolean
expression. It should contain at least one logical
operator in guard's Boolean expression in UML state
diagram.
Guard No Logical Operator.xmi - 1 error(s), 0
warning(s)
```

4.2 Semantic Errors Verifying

On the other hand, semantic errors verifying is fulfilled in a Prolog interpreter, SWI-Prolog. By interpreting generated Prolog code, SWI-Prolog will find two kinds of UML semantic errors defined in Section 3.5 in terms of returning results of query on significant elements in UML diagrams.

A. Completeness: Requirements Not Complete Error

Guard Missing in the Transition from the State "tested" to "evaluated"


committeeChecking [basicReq = false]
preparing
committeeChecking [basicReq = true]
teaching
[timer =< threshold]
scheduling [taskComp = false]
succeeding [req = true && welcomed = true]
scheduling [taskComp = true] / testing
failing [req = false && welcomed = true]
redesigning
timerAdding
evaluated
archiving / evaluating
tested
failing [welcomed = false] / dropCourse
[timer > threshold] / dropCourse
outOfCurriculum

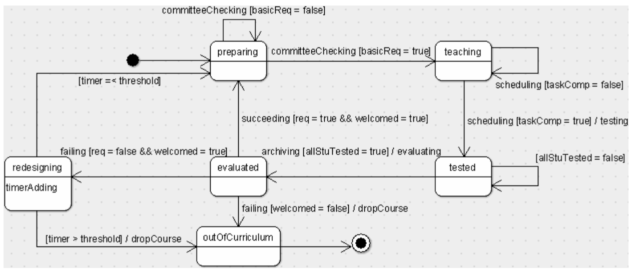

committeeChecking [basicReq = false]
preparing
committeeChecking [basicReq = true]
teaching
[timer =< threshold]
scheduling [taskComp = false]
succeeding [req = true && welcomed = true]
scheduling [taskComp = true] / testing
failing [req = false && welcomed = true]
archiving [allStuTested = true] / evaluating
[allStuTested = false]
redesigning
timerAdding
evaluated
tested
failing [welcomed = false] / dropCourse
[timer > threshold] / dropCourse
outOfCurriculum

Figure 2 and Figure 3. A Case of Requirements Not Complete Error (Above: wrong, Below: right)

In the UML state diagram of the University Academic System, it is necessary that there exists a guard "testValidation" in the transition from the state "tested" to the state "evaluated". That is, the state "tested" will not succeed in transiting to the state "evaluated" until it satisfies the logical condition "allStuTested = true". From the viewpoint of requirement completeness, this logical condition is indispensable, since a course should be evaluated only when all students involved have taken a final test in any form of this course. Or else, data of the students who have taken this course but haven't taken a test are missing. Then even if an evaluation of this course could be carried out, it is not complete or precise yet. To make it clear, two UML state diagrams are shown in contrast with each other in Figure 2 and Figure 3.

In the Prolog code generated respectively from the wrong and right design of UML state diagrams, we can see that the Prolog code for the guard of the transition from the state "tested" to "evaluated" has been changed from "guard(testValidation)" to "guard(void)".

On the basis of transformation from these two UML state diagrams to Prolog code, USCVSC will also generate additional Prolog code for the next step of importing all Prolog code into SWI-Prolog for final verification. The additional Prolog code for this case is as follows:

```
 guardTestedExist(X) :- transition(ts2e, trigger(Y),
            guard(X), action(Z), s0).
```

Import all Prolog code into SWI-Prolog and give a corresponding query in SWI-Prolog for the element satisfies the additional Prolog code above. Detailed running results for the wrong design and right design are shown in Table 5 respectively.

Table 5. Requirements Not Complete Error Verifying

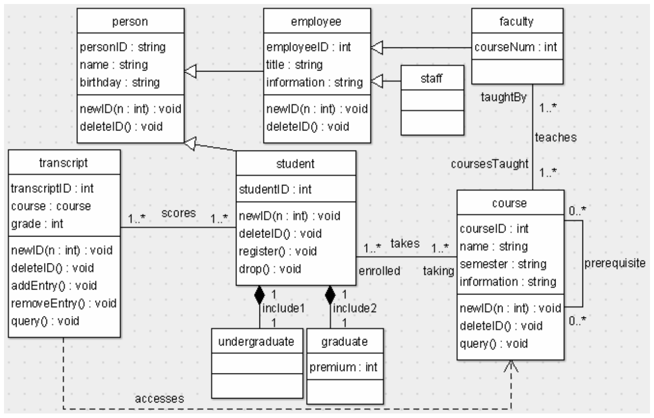
Results for the Wrong Design	Results for the Right Design

`1 ?-`
`consult('F:/UMLVeriTool/U`
`SCVSC/sccode.txt').`
`%`
`F:/UMLVeriTool/USCVSC/scc`
`ode.txt compiled 0.00`
`sec, 5,396 bytes`
`true.`
 	
`2 ?- guardTestedExist(X).`
`X = void.`
	`1 ?-`
	`consult('F:/UMLVeriTool/U`
	`SCVSC/sccode.txt').`
	`%`
	`F:/UMLVeriTool/USCVSC/scc`
	`ode.txt compiled 0.00`
	`sec, 5,812 bytes`
	`true.`
	
	`2 ?- guardTestedExist(X).`
	`X = testValidation.`

Note: “testValidation” is the name of the missing guard in the wrong design.

B. Correctness: Requirements Logic Error

A Teacher Can also Be a Student



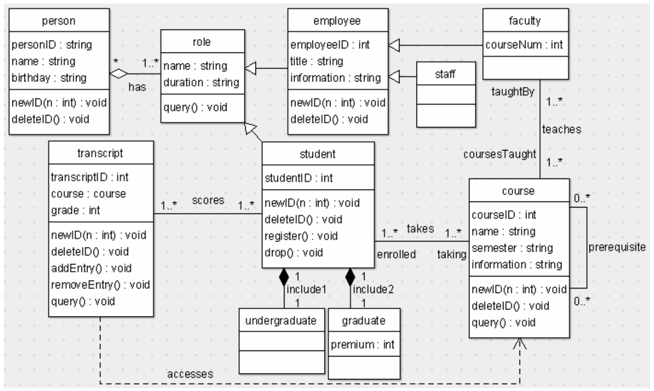


Figure 4 and Figure 5. A Case of Requirements Logic Error (Above: wrong, Below: right)

In the UML class diagram of the University Academic System, it is necessary that there exists an abstract class "role" in the middle of the parent class "person" and its child class "employee" and "student". If there is no abstract class "role", then the class "person" will be the parent class of the class "employee" and "student". That is, either the class "person" is inherited by the class "employee", or the class "student". From the viewpoint of an object, this design restricts that a "person" in this system must be either an "employee" or a "student". i.e., If a "person" is an "employee", he/she cannot be a "student" and vice versa. As a matter of fact, it is a commonplace for a teacher as an employee to get further study in the same university (pursuing a PhD degree); it is also a commonplace for a student to work for his/her university as an employee (working as a graduate or teaching assistant). Therefore, it is necessary to let a "person" be both an "employee" and a "student". We realize a more reasonable design with the help of an abstract class "role" in the middle of these classes. After this modification, a "person" can own one or more "roles" and a "role" can be either an "employee" or a "student". Then, an "employee" can also be a "student" and vice versa. To make it clear, two UML class diagrams are shown in contrast with each other in Figure 4 and Figure 5.

In the Prolog code generated respectively from the wrong and right design of UML class diagrams, we can see that in the wrong design, some relevant code of the abstract class "role" is missing. Due to the page limit, the missing code of the abstract class "role" is not presented here.

On the basis of transformation from these two UML class diagrams to Prolog code, USCVSC will also generate additional Prolog code for the next step of importing all Prolog code into SWI-Prolog for final verification. The additional Prolog code for this case is as follows:

```
takes(faculty, course, s0).
samePositionExist(X) :- teaches(X,_,_), takes(X,_,_).
```

Import all Prolog code into SWI-Prolog and give a corresponding query in SWI-Prolog for the element satisfies the additional Prolog code above. Detailed running results for the wrong design and right design are shown in Table 6 respectively.

Table 6. Requirements Logic Error Verifying

Results for the Wrong Design	Results for the Right Design

`1 ?-`
`consult('F:/UMLVeriTool/U`
`SCVSC/sccode.txt').`
`%`
`F:/UMLVeriTool/USCVSC/scc`
`ode.txt compiled 0.00`
`sec, 3,476 bytes`
`true.`
 	
`2 ?-`
`samePositionExist(X).`
`false.`
	`1 ?-`
	`consult('F:/UMLVeriTool/U`
	`SCVSC/sccode.txt').`
	`%`
	`F:/UMLVeriTool/USCVSC/scc`
	`ode.txt compiled 0.00`
	`sec, 3,864 bytes`
	`true.`
	
	`2 ?-`
	`samePositionExist(X).`
	`X = faculty.`

5. Conclusion and Future Work

The design of software architecture is the core component of Software Engineering and the basis of the following phases. Software design itself is a process of modeling. The modeling language, UML, has been regarded as a standard since it is very qualified to do the job of software design. Due to its imprecise informal graphic notation, software design in UML can bring about ambiguity and flaws. Therefore, various research focuses on the verification of UML. Definitely, manual verification is a waste of system overhead and there is no guarantee of preciseness and completeness. Software testing also has its shortcomings like impreciseness and high cost, which affects the overall efficiency of Software Engineering. So formal verification based on Mathematical Logic by means of lots of formal methods is in the spotlight which overcomes the weakness of previous ways of UML verification. This paper opts for a formal language good at reasoning based on actions, situation calculus as a major method to formally describe UML and finally verify UML in the implementation of situation calculus, Prolog.

The verification work in this paper can be concluded as follows: As for the domain-unrelated design errors in UML, i.e., UML syntax errors, are checked via our prototype tool, USCVSC directly; As for the domain-related design errors in UML, i.e., UML semantic errors, are verified by the combination of USCVSC and Prolog interpreter, SWI-Prolog. That is, a subset of UML diagrams are transformed to Prolog in advance, and some additional Prolog code is generated as well for a final verification. The next step is to import all Prolog code into SWI-Prolog to fulfill the final verification via carrying out query for Prolog elements needed.

There is still much work for further research of UML formal verification based on our current work. Firstly, the application domain should be extended. Only a small core subset of UML diagrams is chosen in this paper, i.e., class diagram and state diagram. However, these two diagrams cannot be suitable for representing all kinds of target systems in the real world. 12 UML diagrams shown in Fig. 2 other than class diagram and state diagram may be qualified to represent some systems with specific needs, such as real-time systems, concurrent systems and context-aware systems. Therefore, more UML diagrams need to be considered for verification. Secondly, the types of formal language for UML specification should be extended. In this paper, we choose situation calculus to formally describe UML class diagram and state diagram, whereas situation calculus is not always the most appropriate carrier for describing every UML diagram. Take UML sequence diagram and timing diagram for example, it is not very convenient for situation calculus to describe this kind of time-based UML diagrams [17]. More suitable formal language or tool should be considered for them, like Computation Tree Logic (CTL) [18], a famous branch of Temporal Logic [19].

References

[1] M. Kaufmann, P. Manolios and J. Moore, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, Massachusetts, USA, 2000.

[2] B. Werner, The Coq Proof Assistant. http://coq.inria.fr/.

[3] K. McMillan, Symbolic model checking – an approach to the state explosion problem. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[4] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, USA, 2004.

[5] International Organization for Standardization (ISO), Unified Modeling Language Specification, Version 1.4.2, 2005.

[6] G. Myers, *The Art of Software Testing, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2004.

[7] Department of Computer Science, University of Texas at Austin, E. W. Dijkstra Archive: the manuscripts of Edsger W. Dijkstra. http://userweb.cs.utexas.edu/users/EWD/.

[8] J. McCarthy: Situations, actions and causal laws. Stanford Artificial Intelligence Project, Memo 2 (1963)

[9] International Organization for Standardization (ISO), XML Metadata Interchange Specification, Version 2.0.1, 2005.

[10] P. Nugues, *An Introduction to Language Processing with Perl and Prolog*. Springer-Verlag, 2006.

[11] Object Management Group (OMG), Object Constraint Language Specification, Version 2.2, 2010.

[12] D. Latella, I. Majzik and M. Massink, Automatic Verication of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6), 1999, 637-664.

[13] Department of Computer Science, The Univerisity of York, The precise UML (pUML) group. http://www.cs.york.ac.uk/puml/.

[14] School of Computing, Queen's University, STL: UML 2 Semantics Project. http://research.cs.queensu.ca/~stl/internal/uml2/index.html.

[15] Department of Programming and Software Engineering, Institute of Computer Science, Ludwig-Maximilians-Universität München (LMU Munich), Hugo/RT. http://www.pst.ifi.lmu.de/projekte/hugo/.

[16] S. Lassen, Basic Action Theory. *BRICS Report Series*. Basic Research in Computer Science (BRICS), Department of Computer Science, University of Aarhus, 1995.

[17] S. Burmester, H. Giese, M. Hirsch, D. Schilling and M. Tichy, The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. *in Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, St. Louis, Missouri, USA, ACM Press, New York, NY, USA 2005, 670-671.

[18] E. Emerson, Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1), 1985, 1-24.

[19] L. Lamport, Introduction to TLA. *HP Labs Technical Reports* SRC-TN-1994-001. http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-1994-001.pdf, 1994.