An Approach of Conformance Verification between Design Models and Code Based on Abstract Syntax Tree

Zhao Liu, Yang Tian, Haihua Yan

Department of Computer Science and Engineering, Beihang University, Beijing, China

Abstract. Design models and code are products of different stages in the software development process. The conformance of design models and code plays an important role in software development process, also is a key principle to improve the maintainability of the software. Currently, the conformance on different level of abstraction is verified mostly by manual inspection. In order to avoid the errors and omissions, we propose a conformance verification approach between design models and code based on abstract syntax tree. The approach verifies the conformance by comparing the class diagram, sequence diagram with the code. Firstly, we discuss how to extract and recover model information from the abstract syntax tree, then give the conformance verification approach between class diagram, sequence diagram and code, finally implement a semi-automatic verification tool based on the approach.

1 Introduction

Traditional object-oriented software development process is divided into several stages including requirement analysis, software design and implementation. As a result, corresponding software products are requirement model, design models and code. Ensuring conformance of these products is an important principle in software development life. However, there are many factors which may reduce the conformance between design models and the code. On one hand, the code is changed while the corresponding design model is not updated. On the other hand, different understanding of the system leads to inconsistency, because developers do not refer to the design models when implementing the system.

There have been some tools on consistent maintenance between design models and code. These tools are mostly based on formal language [1]. At present, conformance verification between UML design models and source code is accomplished by manual inspection. Therefore it will be better to carry out the conformance verification depends on the support of the computer aided tools.

2 Conformance verification based on abstract syntax tree

A complete software system includes two aspects of information, static structure information and dynamic behavior information. UML(Unified Modeling Language) has become a standard to describe the structure and behavior of the software system. In UML models, class diagram describes the static structure of a system by showing the classes and the relationships between them [2,3]. Sequence diagram in UML describes dynamic behavior of a system by showing objects interaction arranged in time sequence [4]. In this paper, the class model corresponding to code is obtained by analyzing the abstract syntax tree of the code, and then the interaction information is organized into a tree structure, called message tree, and finally the conformance verification between class diagram, sequence diagram and code is carried out by comparing the class model, the tree structure with the source code. The process is shown in Figure 1.

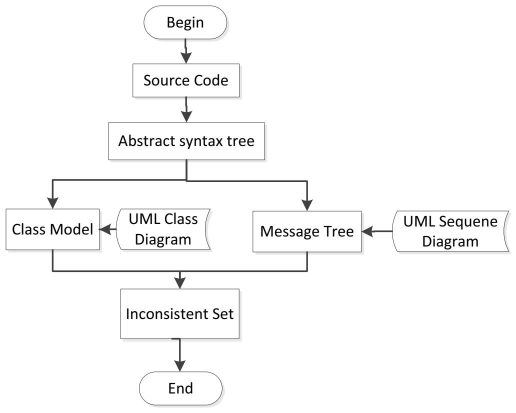


Figure 1. Conformance verification between design models and code

2.1 Abstract sytax tree of code

An abstract syntax tree is a tree of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. It can be generated after lexical analysis and grammar analysis. The information in abstract syntax tree is complete for fine-grained analyzing, at the same time, it is easier to establish rules to map from the abstract syntax tree to design model.

In this paper, changing of interface of the class is our focus. Therefore four types of nodes are processed in abstract syntax tree: attribute, operation, attribute assignment and operation calls. Although the structure of abstract syntax tree depends on the grammatical structure of language and compilation tools, for most object-oriented language, source code can be transformed into an abstract syntax tree with the unified structure, as shown in Figure 2.

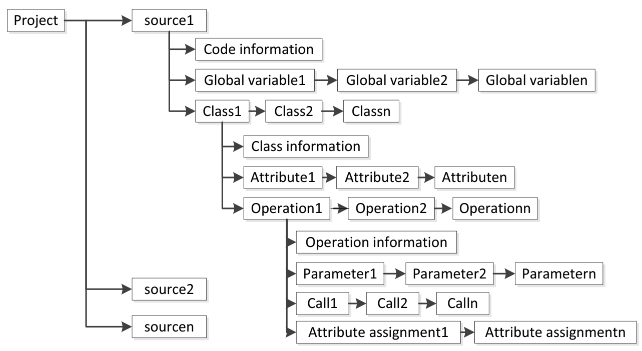


Figure 2. The structure of abstract syntax tree.

In the general case, a project is consisted of several source codes. Each source code contains the code information, global variables and several classes. Each class contains a number of attributes and operations. We can get the abstract syntax tree of each source code with common compiler tools, and then we analyze the abstract syntax tree to organize a project into the structure above. The work is for the next stage of comparison.

2.2 Conformance verification between class diagram and code

Conformance verification between class diagram and code includes two steps: model extraction and model comparison. In the step of model extraction, class model is extracted from the code by analyzing the abstract syntax tree, called implementation class model, and then in the step of model comparison, the generated implementation class model is compared with design class diagram, as a result, the program records and displays the inconsistency.

Step1: (model extraction) There are no direct mappings from UML class diagram to code for some elements in the object oriented programming language, and the implementation of some UML elements are not unique. These problems reduce the traceability in model refinement and implementation. However, we can still extract some model features by analyzing the basic grammar elements of the code. There is a good comparability between the design class diagram and the implementation class model which reverse from the code [5].

Class, attribute and operation in class diagram can directly map to the class, member variables and functions in the code and they can be obtained directly in the abstract syntax tree [6].

To obtain the relationships between classes, we have to analyze the usage and dependency relationships between classes. Three situations below can be extracted as the relationships in UML.

Class A inherits another class B. This situation describes the relationship between a subclass and a superclass. The relationship should be abstracted as generalization between two classes. Most object-oriented programming languages provide relevant keywords to implement generalization and implementation relationships, such as java using “extends” show inheritance relationship between two classes, using “implements” shows a class implements an interface, and the C++ using colon operator shows class inheritance relationship.

A member function of class A uses an instance of class B as its parameters, return values or local variables. This situation describes usage relationship between two classes. In the UML, the relationship should be defined as dependency. Dependency often refers to an occasional, weak usage relationship. Every member function of class A in the abstract syntax tree is traversed to extract the relationships. If its type of parameters, return value or local variable is class B, A dependency is established between class A and class B.

Class A uses an instance of class B as a member attribute. This situation is also a kind of usage relationship. When an instance of class B is a member attribute, to some extent, class B is a part of class A. It is a genus relationship and has a concept of “partial – whole”, so the relationship is defined as association. Every member variable in abstract syntax tree is traversed to extract the relationship. If the type of member attribute is class B, then an association is established between class A and class B.

Although variable types in most object-oriented programming languages are the same with the data types in UML, there are still some exceptions, for example, C++ provides a pointer or reference type, and the types of modifiers such as const, mutable, volatile, while there are no such types in UML, when generating the implementation class model reversely, removing all pointers, references and modifiers does not affect the comparison with UML class diagram.

Step2: (model comparison) After the implementation class model is extracted from the source code, it is compared with the design class diagram. During the process, the compared elements include classes and their relationships.

The comparison is a two-way matching process. On one hand, design class diagram is compared with implementation class model. For a class in design class diagram, if there is a corresponding class in the implementation class model, then it is consistent for the two classes, if not, there may be a lack of the implementation of this class in source code. On the other hand, implementation class model is compared with design class diagram, and we can find the elements which are in the implementation class model while not in the design class diagram.

There are four kinds of comparison results. They are “Same”, “Lack”, “Append” and “Inconsistent”. “Same” indicates an element has the same name, visibility, and other attributes in the design class and the implementation class diagram. “Lack” indicates an element exists in the design class diagram while not exists in the implementation class model. “Append” indicates the elements appended into the implementation class model based on the design class diagram. “Inconsistent” indicates an element has the same name but different type, visibility or other attributes.

The comparison of relationships is relatively complex, so the comparison result is classified with five kinds of situations: “Lack”, “Append”, “Consistent”, “Compatible”, and “Inconsistent”. Firstly, if there is a relationship exactly same in both design class diagram and implementation class model, that is to say the relationship has the same begin class and the same end class, at the same time, its type is also the same, then the comparison result is consistent. Secondly, there are four kinds of relationships in design class diagram: dependency, association, aggregation and composition while there are two kinds of relationships in our implementation class model: dependency and association. These relationships themselves have the characteristics, namely dependency < association < aggregation < composition, so if a stronger relationship in design class diagram is implemented with a weaker relationship in implementation class model, we consider the comparison result is compatible.

2.3 Conformance verification between sequence diagram and code

In object-oriented program language, different variables maybe point or refer to the same object and it is hard to recognize polymorphic objects only by analyzing the source code, so our conformance verification approach between sequence diagram and code just verify whether the message sequence in sequence diagram is implemented by order [8,9] in class level.

To illustrate our approch, we define a data structure called message tree, Messagetree=(V, E), where V is the tree nodes, V=(receiveObject, receive Class, Message Name, MessageType, ParameterList, ReturnValue), E is the parent-child relationships between nodes, E=(V, V). Messagetype includes synchronous message, asynchronous message, creating message, destroying message, return message. Asynchronous message are not considered. Our conformance verification between secuence diagram and code includes two steps: tree extraction and tree comparison. In the step of tree extraction, sequence diagram and abstract syntax tree are transformed into two message trees, then in the step of tree comparison, generated message trees are compared to verify whether the sequence diagram is implemented in the source code in class level.

Step1: (tree extraction) The message tree nodes can be obtained by messages in the sequence diagram, so a sequence diagram can be transformed into a message tree by resolve the message information. Firstly, Information from sequence diagram described in XML is extracted, these information includes the message sequence, composite fragment and its interaction domain. Secondly, a message node Vm is established and the corresponding message information is recorded in the node. At last, for each message a in the sequence diagram, we find the message b before message a, which is a synchronous message or creating message and the receive objects of message b is the send object of message a, then a parent-child relationship is added between the node a and node b. The Figure 3 shows a sequence diagram and corresponding message tree.

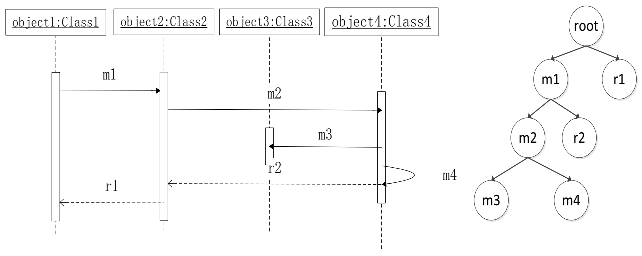


Figure 3. Design sequence diagram and corresponding message tree.

On the other hand, message tree nodes can also be obtained by analyzing the function calling, object creating and return statements in abstract syntax tree. These statements are called message statement. The algorithm below describes the process how to generate the message tree starting with a function based on abstract syntax tree.

Algorithm generate the message tree based on abstract syntax tree

```
visit(AST ast, Function function, MessageNode V)
{
Search the node corresponding to the function in ast
for(every message statement in node)
        if(statement does not contain function)
                make node V’
                add parent-child relationship(V,V’)
                visit(ast, function, V’)
}
```

Step2: (tree comparisons) Comparing two message trees is the second step. During the process, the compared elements include tree nodes and their order in the level. For a node in message tree of sequence diagram, if there is a corresponding node in message tree of source code and their relative orders are the same, then it is consistent for the two nodes. That is to say, the message corresponding to the node in sequence diagram is implemented in source code in class level, or its comparison result is inconsistency.

3 A prototype tool

In this section we put an example library management system to verify this method. The example is a typical information management system, readers can borrow books, return books, search books in the system, the design models of the system include a class diagram and several sequence diagram. Figure 4 are the interface of the prototype tool. Developer can draw class diagram and sequence diagram as design model in the tool.

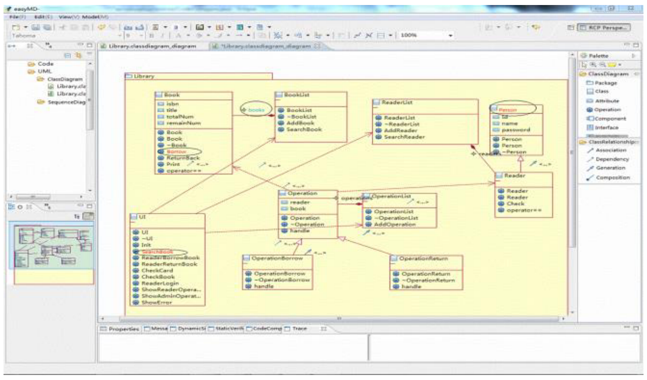
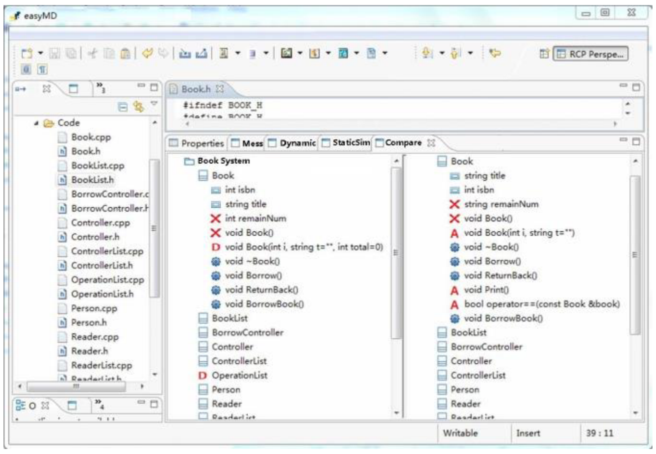


Figure 4. Drawing class diagram.with the tool.

The comparison results are displayed in the result bar, and different comparison results are represented in different marks. Figure 5 is a comparison results. There are some labels in the result bar. Elements marked with “D” are deleted in the design model. Elements marked with “A” are added into the implementation model, in another way to say, they are added into the code. Elements marked with “×” are different between design model and code.



```
#ifndef BOOK_H
```

Code
Book.cpp
Book.h
BookList.cpp
BookList.h
BorrowController.c
BorrowController.h
Controller.cpp
Controller.h
ControllerList.cpp
ControllerList.h
OperationList.cpp
OperationList.h
Person.cpp
Person.h
Reader.cpp
Reader.h
ReaderList.cpp

Book System	Book
Book	string title
int isbn	int isbn
string title	× string remainNum
× int remainNum	× void Book()
× void Book()	A void Book(int i, string t="")
D void Book(int i, string t="", int total=0)	void ~Book()
void ~Book()	void Borrow()
void Borrow()	void ReturnBack()
void ReturnBack()	A void Print()
void BorrowBook()	A bool operator==(const Book &book)
BookList	void BorrowBook()
BorrowController	BookList
Controller	BorrowController
ControllerList	Controller
D OperationList	ControllerList
Person	Person
Reader	Reader

Figure 5. The comparison results.

4 Conclusions

This paper presents an approach which extracts and recovers system structure and behavior characteristics based on the abstract syntax tree, then verify the conformance between design models and code including conformance between class diagram and code, conformance between sequence diagram and code. The verification process can be accomplished by computer, and sometimes it needs a further manual inspection, so it is a semi-automation approach. The method has been embedded in conformance verification tool. Its results can locate the inconsistency between design models and code. It is an important reference for the project development and maintenance personnel.

At present, the comparison of the design model and the code is only based on element name. In the process of program, developers have to consider the specific program language, algorithm, and framework to refine the elements in design model such as adding auxiliary class, or implementing the relationship between classes with a different style. These behaviors lead that design model and code seem different but they perform the same function. In the next stage, we should classify and summarize the different implementation ways of the design model in order to compare the design model and the code more fine-grained.

At the same time, extraction and comparison rules are still relatively simple, therefore in the next step, we will summarize more detailed extraction and comparison rules to detect the consistency. If possible, we can execute the source code to obtain the object information and verify the conformance between sequence diagram and code in object level.

References

1. David Evans,John Guttag,James Horning,and Yang meng Tao. LCLint:A Tool for Using Specifications to CheckCode.
2. Booch G,Rumbaugh J,Jacobson I. The Unified Modeling Language User Guide. Addison-Wesley Publishing Company, 2009.
3. OMG Unified Modeling Language (OMG UML), Superstructure 2011.
4. Sharp R, Rountev A. Interactive exploration of UML sequence diagrams[C]//Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on. IEEE, 2005: 1-6.
5. Liu Chao, Li Jian, Shen Haihua. Reversed Automatic Generation of Visualized Class Diagram of Object-Oriented Program. Journal of Beijing University of Aeronautics and Astronautics. 1998.
6. Sutton A, Maletic J I. Recovering UML class models from C++: A detailed explanation[J]. information and software technology, 2007, 49(3): 212-229.
7. Egyed A. Automated abstraction of class diagrams[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2002, 11(4): 449-491.
8. Kung D, Hsia P. A reverse engineering approach for software testing of object-oriented programs[C]//Application-Specific Systems and Software Engineering and Technology, 1999. ASSET'99. Proceedings. 1999 IEEE Symposium on. IEEE, 1999: 42-49.
9. Kollmann R, Gogolla M. Capturing dynamic program behaviour with UML collaboration diagrams[C]//Software Maintenance and Reengineering, 2001. Fifth European Conference on. IEEE, 2001: 58-67.