Bulletin of Electrical Engineering and Informatics
Vol. 7, No. 4, December 2018, pp. 665~672
ISSN: 2302-9285, DOI: 10.11591/eei.v7i4.1358

Improving Consistency of UML Diagrams and Its Implementation Using Reverse Engineering Approach

Vasanthi Kaliappan, Norhayati Mohd Ali
Faculty of Computer Science and Information Technology,
Universiti Putra Malaysia, Serdang, 43400 Selangor Darul Ehsan, Malaysia

Article Info

Article history:

Received Aug 17, 2018
Revised Oct 26, 2018
Accepted Nov 14, 2018

Keywords:

Inconsistency checking
Model Driven Engineering
Reverse engineering
UML diagrams

ABSTRACT

Software development deals with various changes and evolution that cannot be avoided due to the development processes which are vastly incremental and iterative. In Model Driven Engineering, inconsistency between model and its implementation has huge impact on the software development process in terms of added cost, time and effort. The later the inconsistencies are found, it could add more cost to the software project. Thus, this paper aims to describe the development of a tool that could improve the consistency between Unified Modeling Language (UML) design models and its C# implementation using reverse engineering approach. A list of consistency rules is defined to check vertical and horizontal consistencies between structural (class diagram) and behavioral (use case diagram and sequence diagram) UML diagrams against the implemented C# source code. The inconsistencies found between UML diagrams and source code are presented in a textual description and visualized in a tree view structure.

Corresponding Author:

Norhayati Mohd Ali,
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia, Serdang, 43400 Selangor Darul Ehsan, Malaysia.
Email: hayati@upm.edu.my

1. INTRODUCTION

Software life cycle deals with various changes either in software operating environment or requirements. Software evolution cannot be avoided and it is an important activity in software development life cycle because development processes are vastly incremental and iterative. Three explicit maintenance activities in software evolution are corrective to fix defect, adaptive to adapt new technologies and environment, and perfective to enhance and improve software quality. In software development, inconsistencies between architectural artifacts and the implemented source code might occur due to erroneous implementation of the design architecture or the separate and uncontrolled changes or amendments in the code [1]. To plan and repair these inconsistencies, software developers have to revise their workflow further to reinvestigate the model changes that contribute to these inconsistencies. Other than fixing inconsistencies, software developers also have to fix other model changes that were dependent on the erroneous model elements [2]. In times where development schedule and timeline are tight or urgently required projects, manual inconsistency detection and fixing may easily breach model consistency conformance due to errors and mistakes made by human or misunderstanding of the model. According to [3], the manual recovery of UML class diagrams is a time consuming and expensive operation, which led industries lack of interest in maintenance activities. Furthermore, their study discovered that most automatic reverse-engineering tools perform poorly. The tools mostly focused on producing simple class diagrams whereby design abstractions were not represented properly and correctly. In such scenarios, checking consistency between a designed model and its implementation is much required to ensure that function of models are implemented as they should be during various changes in software lifetime. Thus, consistency

checking can supports developers in models' understanding. This can be achieved by implementing its design properties which helps developers to use model driven design approaches more effectively.

The UML diagrams have been widely used in software design, requirement analysis [4], software testing [5] and other software engineering activities.There are numerous related works about inconsistency checking among source code and UML diagrams and between different UML diagrams. Table 1 summarizes the studies in models consistency management. The finding shows that previous studies concentrated more on the Java and C++ languages. In addition, class diagram received more attention compared to other diagrams.

Table 1. Summary of Related Work in Model Consistency Management

Year	Authors	FE/RE/RTE	UML diagram	Source Code	Consistency Type
2017	Van C.P, Ansgar R, Sebastien G, Shuai L [6]	RTE	Class and Statechart	C++	Horizontal
2016	Michael J D, Kyle S, Michael L.C, Jonathan I.M [3]	RE	Class	C++	Vertical
2016	Bashira R, Lee S, Khan S, Chang V, Farid S [7]	FE	Multiple diagrams	-	Horizontal
2016	Chavez H, Shen W, France R, Mechling B, Li G [8]	FE	Class	Java	Vertical
2016	Rao A, Kanth T, Ramesh G [9]	FE	Multiple diagrams	Java	Horizontal
2015	Huy T, Faiz U.M, Uwe Z [10]	FE	Activity	Java	Vertical
2013	Reder A, Egyed A [11]	RE	Multiple diagrams	-	Horizontal
2013	Michael L.C, Michael J D, Jonathan I.M [12]	RE	Class	C++	Vertical
2012	Selim C, Hasan S, Bedir T [1]	FE	Sequence	Java, C++	Vertical
2011	Egyed A [2]	FE	Class, State chart and Sequence	Java	Horizontal
2008	Laszlo A, Laszlo L, Hassan C [13]	RTE	Class	Java, C#	Vertical

Legend:
FE: Forward Engineering **RE**: Reverse Engineering **RTE**: Round Trip Engineering

Inconsistencies between UML models and source code could occur due to various changes implemented during the project's lifetime at a source code level. However, design models were not updated accordingly due to constraints such as time, money, resources and separate and uncontrolled evolution [1]. Based on IEEE 2016 Programming Language Spectrum [14] rating, despite being one of the most popular object oriented language among software developers, it is observed that recent studies in inconsistency management give more attention for Java and C++ compared to C#. This has been studied and synthesized in a systematic critique conducted in [7].

Furthermore, it was found that most studies in existing literatures focused more on class, state chart and sequence diagrams compared to use case diagram. Combination of class diagram, use case diagram and sequence diagram are not explored much. The study also reveals that majority of literature for model inconsistencies were done using forward engineering. In contrast to the vertical consistency problems, horizontal consistency problems were more emphasized in studies and researches. Therefore, our work aims to propose a mechanism and to develop a tool that could improve consistency between class diagram, use case diagram and sequence diagram and its C# implementation using reverse engineering approach. Several consistency rules are adopted to detect and diagnose vertical and horizontal inconsistencies. The outcome of our work is expected to assist software developers to use model driven design approaches more effectively.

The aim of this paper is to describe the development of a UML-Code Consistency Checker Tool (UCCCT) to aid software developers to maintain design models consistency in a quick and correct way to be in line with the source code implementation, specifically in Model Driven Engineering (MDE). The focus of UCCCT tool is to detect vertical and horizontal inconsistencies between class diagram, use case diagram and sequence diagram using consistency rules. The following section is the description of our approach.

2. RESEARCH METHOD

The research method is organized in several phases to achieve the research objectives. The initial phase of this research is to review and analyse the consistency checking tools that focus on UML diagrams

and source code. Several consistency rules are identified and compiled based on the analysis. The intermediate phase concerned with the development of a prototype for the consistency checking tool between C# code and UML diagrams (i.e., use case diagram, sequence diagram and class diagram). The final phase of this research is to address the evaluation of the consistency checking tool for C# code and UML use case, sequence and class diagrams. The following paragraph explains the development of UCCCT.

The name of the developed prototype tool is UCCCT which stands for **U**ML-**C**ode **C**onsistency **C**hecker **T**ool. Figure 1 shows the architecture of the consistency checker tool. It depicts a modeling tool STAR UML [15] on the lower-right corner and a development tool Visual Studio [16] in the lower-left corner. The interface file types that are compliant for UCCCT are .CS and .XMI. The metadata reader extracts relevant information for the tool to perform consistency checking. The consistency checker reads predefined consistency rules and evaluate them to identify which consistency rules are violated between design model and implemented source code. The UCCCT is basically a Windows application which has 4 main steps to be executed in this tool.

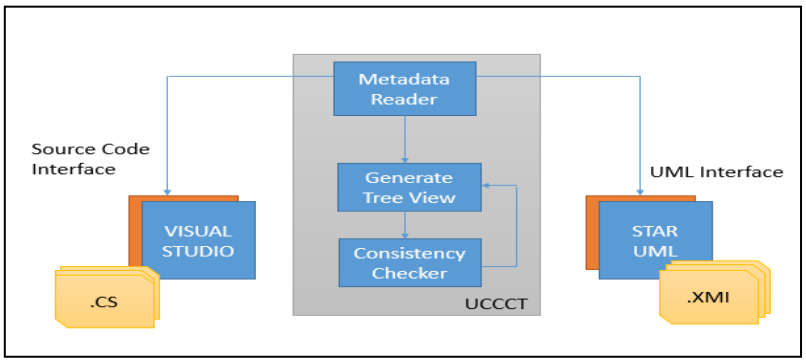

Figure 1. Architecture of UCCCT prototype tool

Step 1: Processing of XMI of UML Diagrams

UML diagrams are the input to the system. Since the diagrams are not in a textual format, intermediate representations of the diagrams are required. XML Metadata Interchange (XMI) is a metadata representation of the model, which is used for the conversion of the model to text and vice versa. XMI format will be the combination of the XML tags and the UML elements as per standard specified by Object Management Group (OMG) [17] for exchanging information of the models. The STAR UML tool has the capability of supporting UML 2.x version and also XMI exporting feature. Thus, it is chosen for the UML modeling task in this project. The XMI representation for use case diagram, class diagram, and sequence diagram will be parsed to retrieve relevant information which in turn used to generate a design Tree View. Parsing an XMI file means that reading the information embedded in XML tags using Visual Studio C# XmlReader to retrieve the relevant UML diagram elements. The XmlReader is a faster and consume less memory. The XmlReader read through the XML string one element at a time, while allowing reader to look at the attributes, and then moves on to the next XML element. The classes, operations, attributes and relationship information are extracted using the parser. The XMI representation of UML diagrams contains more information than needed for the consistency checking tool. Thus, only relevant and essential information are extracted and stored into the design Tree View. Metadata information extracted from XMI is used to generate a design Tree View for class diagram, sequence diagram and use case diagram. The design Tree Views will be used in later part of the checker tool to perform vertical and horizontal consistency checking. Figure 2 depicts the tree view structure for information extracted from respective UML diagrams (use case diagram, class diagram and sequence diagram).

Step 2: Reverse Engineering of Source Code

Reverse engineering distinguishes the system's components and their interrelationships, and generates representations of the system in another form or at a higher level of abstractions. In our method, the C# source code will be reverse engineered into an implemented class model tree view. The prototype tool filters valid .CS type file to be compiled from project selected. When a .Net project is compiled, the language compiler compiles the code and converts it to intermediate language codes called CIL (Common Intermediate). The output of the build process is an assembly (.dll) which contains .NET metadata. In Microsoft .NET framework, assembly refers to a certain data structure that describes the high-level structure of the code.

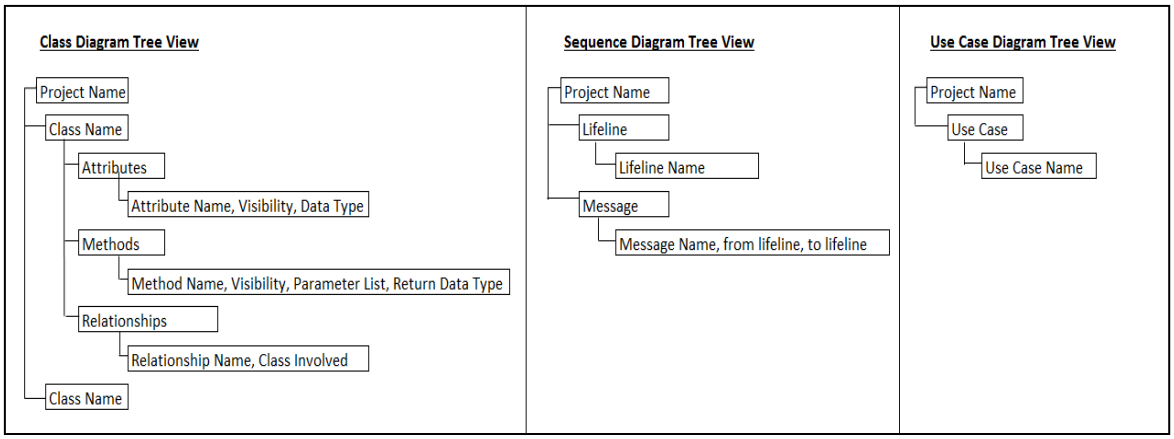

Figure 2. Tree view structure for class, sequence and use case diagrams

Step 3: Perform Inconsistency Checking

Horizontal and vertical consistencies checking are used in UCCCT with constraints and rules are defined for the UML meta-model to trace the inconsistencies. Horizontal inconsistencies checking are performed between a use case diagram vs implemented class diagram, and a sequence diagram vs implemented class diagram. The vertical inconsistencies checking are performed between designed class diagram vs implemented class diagram. The list of horizontal consistency rules are selected from study conducted by [9]. The rules defined for inconsistency checking are shown in Table 2.

Table 2. List of Inconsistency Rules

Diagrams	ID	Description
Use Case Diagram vs Implemented Class Diagram	UC01	Use cases represent functions or services of information systems. Operations of classes are ultimate undertakers of these functions or services. So use cases in Use case diagrams must be assigned to operations of classes.
Sequence Diagram vs Implemented Class Diagram	SQ01	An object in Sequence Diagrams must be an instance of a normal class in Class Diagrams. An object cannot be created by itself, only be created by a class. Since we cannot create an instance from an abstract class, the class an object belongs to must be a normal class.
	SQ02	When the name of a class is modified in Class Diagrams, the name of the corresponding class must be updated synchronously in Sequence Diagrams. Objects and messages in Sequence Diagrams are derived from classes in Class Diagrams. Therefore modification on the class should be updated in all correlative Sequence Diagrams.
	SQ03	If an object sends a message to another object in Sequence Diagrams, there must be a dependency relationship between the two classes that the two objects belong to respectively. Contrariwise, if there is a dependency relationship between two classes, there must be at least one message interaction between the corresponding objects.
	SQ04	A message of Sequence Diagrams must correspond to an operation of the receiver (an object), and the operation is visible to the sender (an object). A message in Sequence Diagrams is an order that an object sends to another object. The order must be an action that the receiver can complete. The action ultimately is represented as an operation of the receiver.
	SQ05	If a class is deleted in Class Diagrams, the corresponding objects and messages of the class should be deleted synchronously in Sequence Diagrams. As stated in Rule SQ02, objects and messages in Sequence Diagrams derive from classes in Class Diagrams. Therefore, correlative objects and messages should be deleted when a class is deleted.
Class Diagram vs Implemented Class Diagram	CD01	A class object implemented exactly same in both design class diagram and implementation class model.
	CD02	Class attributes are implemented exactly same in both design class diagram and implementation class model, with same visibility and data type.
	CD03	Class methods are implemented exactly same in both design class diagram and implementation class model, with same visibility, parameter list and return type.
	CD04	A relationship exactly same in both design class diagram and implementation class model and has the same begin class and the same end class.

Step 4: Display Consistency Checking Result

Extracted metadata information from UML diagrams and implemented source code are visualized as Tree Views. For each type of diagrams involved, respective consistency rule checking will be applied. The

element or tree node that violates the defined consistency checking rules is highlighted in red color together with its description.

3. RESULTS AND DISCUSSION

Implementation of the UCCCT is performed by using Microsoft Visual Studio Community 2017 with .NET Framework 4.6.1. The UCCCT prototype is an executable Windows application which was created using C# language. The tool is easy and simple to use. There are 2 input fields on the Main Screen to browse a source code project folder and XMI files. This is shown in Figure 3. Upon selection of project folder and XMI file, a click on "Check Consistency" action button will display a consistency checking result for each diagram. This is shown in Figure 4 and Figure 5.

Figure 4 shows the inconsistencies between design class diagram and implemented class diagram. Figure 5 illustrates the inconsistencies between design sequence diagram and implemented class diagram. The consistency results can be saved and print.

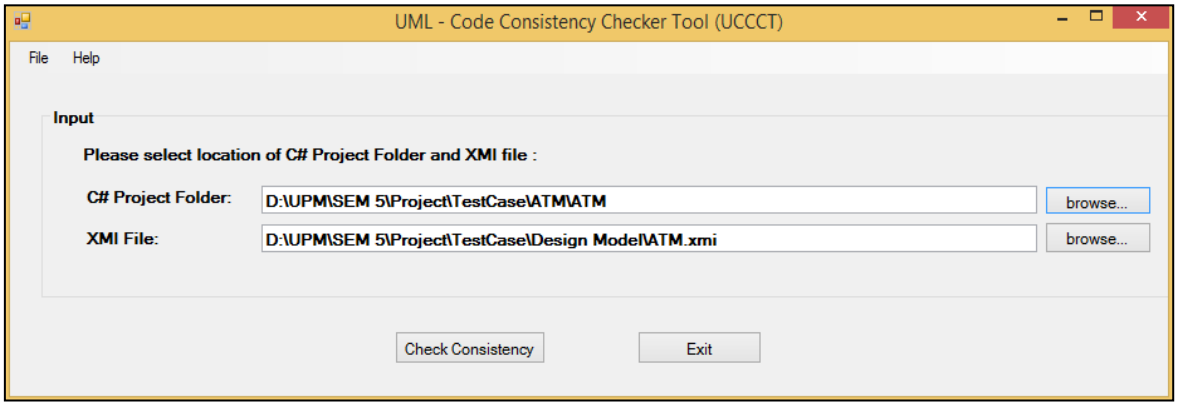

Figure 3. Main screen to browse C# project folder and XMI file

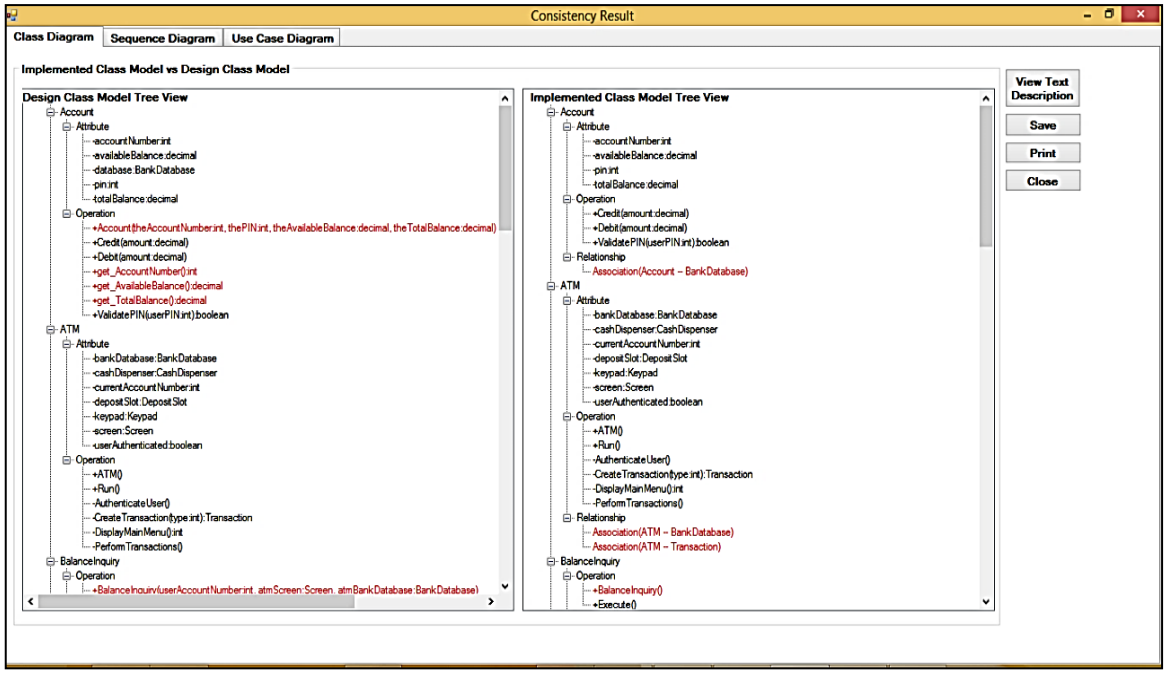

Figure. 4. Consistency checking result for class diagram

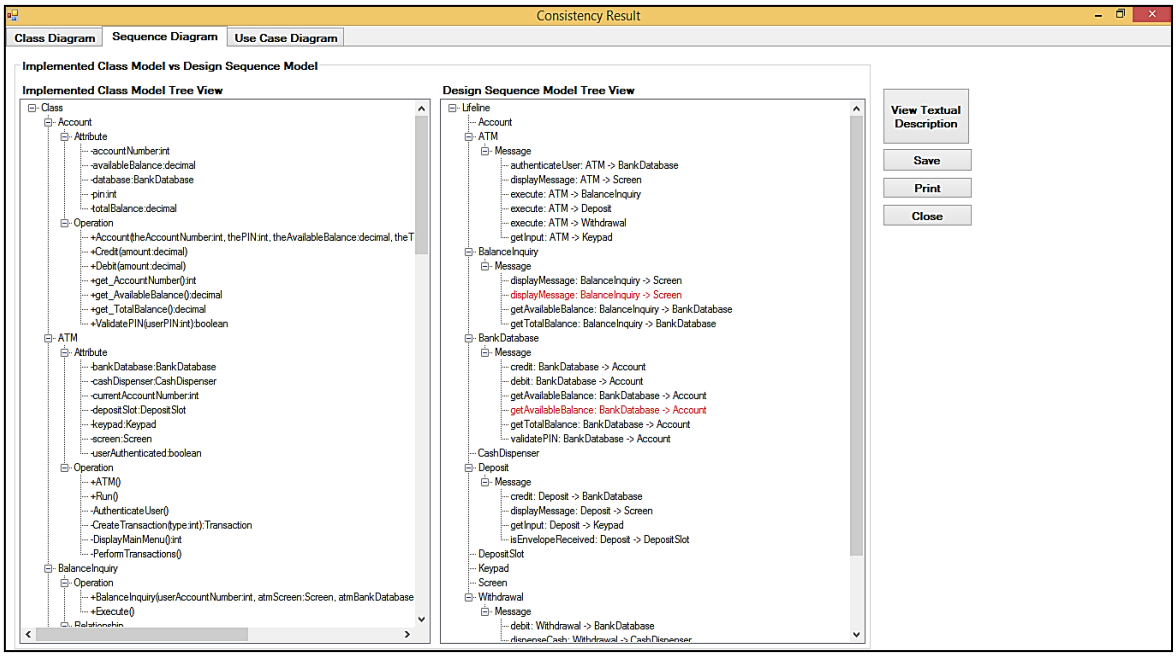

Figure 5. Consistency checking result for sequence diagram

3.1. Evaluation

An evaluation was conducted on UCCCT to make sure it works as it should be in terms of correctness, completeness and quality of the developed prototype. The evaluation involves an experiment that has following objectives. The first objective was to evaluate UCCCT in terms of its simplicity and ease of learning. The second objective was to evaluate the tool's effectiveness. Participants were asked to perform consistency checking between design model and implemented source code using manual way and UCCCT.

A case study of ATM System was provided for participants' evaluation. After completing the assigned tasks, the participants were asked to complete a questionnaire regarding their experience with the tool. The evaluation has been conducted with 10 software developers from one of the private bank's IT Department. All the participants have more than 2 years experiences in software development using C# language. [18, 19] stated in their study that ten seems a small number but it is proved that ten is enough in evaluating system usability. They explained that by increasing the number of participants will not affect the usability score since the error encounter within small number of participants will just be repeated and other participant might encounter the same error.

The effectiveness of this prototype is measured by recording the number of inconsistency identified manually ($Total_M$) and number of inconsistency identified by using UCCCT ($Total_A$). Participants were given ($Time_M$) 30 minutes time to complete each task. The time taken to run the UCCCT prototype ($Time_A$) was recorded less than 5 minutes, including generating XMI file from modelling tool. Table 3 shows the number of inconsistency found using manual approach vs using UCCCT. Based on the result, (refer to Table 3) total inconsistency found by UCCCT prototype was 71 but the average inconsistency found using manual approach is 18.3 per participant. This clearly shows that UCCCT is fast, efficient and time saving tool. The results also clearly present software developers difficulty in finding inconsistency for Rule CD03, CD04 and SQ03 which requires more time and effort.

Next, is to evaluate the usability aspects via a set of questionnaire that consists of nine questions. System Usability Scale (SUS) [20] with minor adjustment is used as it provides a standardized, simple, ten-item scale which gives a global view of subjective assessments of usability. The set of questions were chosen selectively from previous studies.

Table 3. Inconsistencies Discovered via UCCT and Manual Approach

Inconsistency Check	Rule	Total inconsistency found using UCCCT prototype, ($Total_M$)	Total inconsistency found via manual approach, ($Total_A$)									
			Participants									
			1	2	3	4	5	6	7	8	9	10
Class Diagram vs Implemented Class Diagram	CD01	2	2	2	2	2	2	2	2	2	2	2
	CD02	4	3	5	4	3	2	2	3	2	3	6
	CD03	25	5	6	4	5	7	5	4	5	2	4
	CD04	7	0	1	1	2	0	1	0	1	1	1
Sequence Diagram vs Implemented Class Diagram	SQ01	0	1	1	0	0	0	0	1	0	2	1
	SQ02	3	3	3	3	3	2	3	2	3	3	2
	SQ03	27	1	1	0	0	0	1	0	1	0	2
	SQ04	0	0	0	0	1	0	1	0	0	2	0
	SQ05	0	0	0	0	1	0	1	0	0	2	1
Use Case Diagram vs Implemented Class Diagram	UC01	3	3	2	3	3	2	3	3	3	2	3
Total		71	18	21	17	20	15	19	15	17	19	22

Table 4 shows the usability responses based on the responds from ten participants. The usability responses shown in Table 4 were highly positive. Even though 80% participants responded that the value of UCCCT is good and the consistency checking result is correct, they prefer to perform a double check before accepting the inconsistency result blindly. One of the reasons from participant's viewpoint, is they have to check and do analysis on the inconsistency before making any fixes. A thorough impact analysis must be conducted before making any changes based on the inconsistency list. Thus, we can conclude that majority of participants has responded that UCCCT tool is effective and easy to use.

Table 4. Usability Responses

Question	EG	G	Ave	B	EB	Y	N	BA	DC
The tool was easy to understand.	3	5	1	1					
The tool could easily be used in a typical software project frequently.	6	2	2						
How easy is it to understand the graphical user interface (GUI) of the consistency checker tool?	6	2	2						
How easy is it easy to interpret the inconsistency result?	3	2	4	1					
Overall value of the consistency check tool	3	5	2						
Do you think consistency rules set in this tool are relevant and easy to understand?						7	3		
Do you trust the manual checking more than the results from the checker tool?						8	2		
Do you think you could make use of the tool in your software development?						8	2		
Did you blindly accept the results from the tool or did you double check the answers manually?								4	6

Legend:
EG:Extremely Good, **G**: Good, **Ave:**Average, **B**:Bad, **EB**:Extremely Bad, **Y**:Yes, **N**: No, **BA**:Blindly Accept, **DB**:Double Check

4. CONCLUSION AND FUTURE WORK

We have described our consistency checking tool, UCCCT which extracts metadata of design diagram and implemented source codes using reverse engineering approach, and then perform consistency checking between them. We believe this approach could assist software developers in maintaining design models consistency in a quick and correct way against its source code implementation. The list of consistency rules to check vertical and horizontal consistencies between structural (class diagram) and behavioral (sequence diagram and use case diagram) UML models against the implemented C# source code were presented. We also explained briefly our methodology, overview of UCCCT development and its end user evaluation. Our plans for future work includes extension of various UML design diagrams and improve flexibility by incorporating different types of UML modelling tools. For comprehensive consistency checker tool, more vertical and horizontal consistency rules can be considered. The framework can be also enhanced to provide possible solutions to fix the detected inconsistencies.

ACKNOWLEDGEMENT

The authors are grateful to Ministry of Education Malaysia (MOE) and Universiti Putra Malaysia (UPM) via the Fundamental Research Grant Scheme (FRGS, Vot no:5524832) for funding the project.

REFERENCES

[1] Selim C, Hasan S, Bedir T. *An Approach for Detecting Inconsistencies between Behavioral Models of the Software Architecture and the Code*. IEEE 36th International Conference on Computer Software and Applications. Izmir. 2012; 257-266.
[2] Egyed A. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions of Software Engineering*. 2011; 37(2): 188-204.
[3] Michael J D, Kyle S, Michael L.C, Jonathan I.M. *A Tool for Efficiently Reverse Engineering Accurate UML Class Diagrams*. IEEE International Conference on Software Maintenance and Evolution. Raleigh. 2016; 607-609.
[4] Hui H, J, Dongyue Y. UML-based Requirements Analysis on Risk Pre-control System in Coal Enterprise. *Indonesian Journal of Electrical Engineering and Computer Science*. 2013; 11(7):4012-4019.
[5] Sahoo P, Mohanty J.R. Early Test Effort Prediction using UML Diagrams. *Indonesian Journal of Electrical Engineering and Computer Science*. 2017; 5(1):220-228.
[6] Van C.P, Ansgar R, Sebastien G, Shuai L. *Bidirectional Mapping between Architecture Model and Code for Synchronization*. IEEE International Conference on Software Architecture. Gothenburg. 2017; 239-242.
[7] Bashira R, Lee S, Khan S, Chang V, Farid S. UML models consistency management: Guidelines for software quality Manager. *36th International Journal of Information Management*. 2016; *36*(6):883-899.
[8] Chavez H, Shen W, France R, Mechling B, Li G. An Approach to Checking Consistency between UML Class Model and Its Java Implementation. *IEEE Transactions of Software Engineering*. 2016; *42*(4):322-344
[9] Rao A, Kanth T, Ramesh G. A Model Driven Framework for Automatic Detection and Tracking Inconsistencies. *Journal of Software*. 2016; 11(6): 538 – 553.
[10] Huy T, Faiz U.M, Uwe Z. *A graph-based approach for containment checking of behavior models of software systems*. Proceedings of the 2015 IEEE 19th International Conference on Enterprise Distributed Object Computing. Adelaide .2015; 84-93.
[11] Reder A, Egyed A. Determining the Cause of a Design Model Inconsistency. *IEEE Transactions of Software Engineering*. 2013; 39(11): 1531-1548.
[12] Michael L.C, Michael J D, Jonathan I.M. *srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration*. 29th IEEE International Conference Proceedings on Software Maintenance (ICSM). Eindhoven. 2013; 516-519.
[13] Laszlo A, Laszlo L, Hassan C. *A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering*. 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. Belfast. 2008; 463-472.
[14] *http*://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016
[15] https://docs.staruml.io
[16] https://docs.microsoft.com/en-us/dotnet/csharp
[17] http://www.omg.org/spec/XMI
[18] Lene N, Sabine M. *The usability expert's fear of agility: an empirical study of global trends and emerging practices*. Proceedings of the 7th Nordic Conference on Human-Computer Interaction: Making Sense Through Design. Copenhagen. 2012; 261-264.
[19] Roobaea A, Pam J. M. *How many participants are really enough for usability studies?*. Science and Information Conference (SAI). London. 2014; 48-56.
[20] Brooke, J. SUS-A quick and dirty usability scale. In: Jordan P, Thomas B, Weerdmeester B, McClelland I. Usability Evaluation in Industry. London: Taylor & Francis. 1996: 189-194.