Combining Design by Contract and Inference Rules of Programming Logic towards Software Reliability

Nuha Aldausari*, Cui Zhang and Jun Dai
Department of Computer Science, California State University, Sacramento, CA 95819, U.S.A.

Keywords: Software Security, Software Reliability, Program Specifications, Error Detection, Design by Contract, Programming Logic.

Abstract: Detecting errors in software products is very important to software reliability because many security vulnerabilities are caused by the defects in software. Design by contract (DBC) is an effective methodology that dynamically checks whether a program meets its specifications, which are also called design contracts, and whether there are errors in the program. The contracts for object-oriented programs are defined in terms of *preconditions* and *postconditions* for methods as well as *invariants* for classes. However, if there is an error in a large piece of code that has a design contract, it is still difficult to identify the exact location of that error. To address this issue, a tool named *Subcontractor* has been developed. Subcontractor is implemented in Eclipse environment using libraries such as Java Development Tools (JDT), Plugin Development Environment (PDE), and JFace. The tool Subcontractor is built upon an open source DBC tool, OpenJML Runtime Assertion Checking (RAC), which is a tool that verifies specifications at runtime. Subcontractor combines this DBC tool with inference rules of program logic for *if-statements* and *loop-statements* to automatically generate subcontracts for programs. When the programs, with subcontracts automatically generated and inserted by Subcontractor, are verified using OpenJML Runtime Assertion Checking (RAC), identification of errors in the code can be facilitated.

1 INTRODUCTION

A plethora of applications are created every day; however, 64% of developers are not confident in their applications' security according to Bill Gates at RSA conference in 2005 (Gates, 2005). Design flaws and coding defects cause applications to be vulnerable to attacks or incorrect entry of data. Since Design By Contract (DBC) (Meyer, 1992) is an effective methodology that dynamically checks whether a program meets its specifications and whether there are errors in the program, introducing DBC in software development processes and providing DBC support in the programming environment can improve robustness and reliability of produced software, which is important to software security. There are multiple tools that support DBC for Java programming language such as Java Modeling Language (JML) (Leavens, 2013), Jass (University of Oldenburg, 2001), and Contracts for Java (C4J) (Bergström, 2012). In these tools, DBC assists in improving the quality of the code. However, one main limitation in these DBC tools is that when the code is lengthy, it is still difficult to identify errors. For example, if the precondition of a large piece of code is valid, but the postcondition is not valid, the programmer needs to search the whole code to find the errors. If inference rules of programming logic (Finkel, 1996) can be used to automatically generate subcontracts, and if the subcontracts can be automatically inserted into the large source code for if-statements and/or loop-statements, the large verification task can be automatically partitioned into smaller ones. Therefore, automating the generation of sub assertion statements aid in spotting the errors and tracing the program.

For this purpose, we have developed a tool called Subcontractor which was built upon an open source DBC tool called OpenJML (Cok, 2015). Subcontractor is a developed Eclipse plugin that provides the benefit of automatically generating and

* Nuha received scholarship from Princess Nourah bint Abdulrahman University to complete MS in Computer Science at California State University, Sacramento.

Aldausari, N., Zhang, C. and Dai, J.
Combining Design by Contract and Inference Rules of Programming Logic towards Software Reliability.
DOI: 10.5220/0006849603640371
In *Proceedings of the 15th International Joint Conference on e-Business and Telecommunications (ICETE 2018) - Volume 2: SECRYPT*, pages 364-371
ISBN: 978-989-758-319-3

adding into code subcontracts based on inference rules for if-statements and/or loop-statements. Subcontractor combines DBC, dynamic program analysis technique, with inference rules of programming logic, which are originally for static program verification. When these rules are integrated with DBC tools, they can help locate defects in the code to improve software reliability.

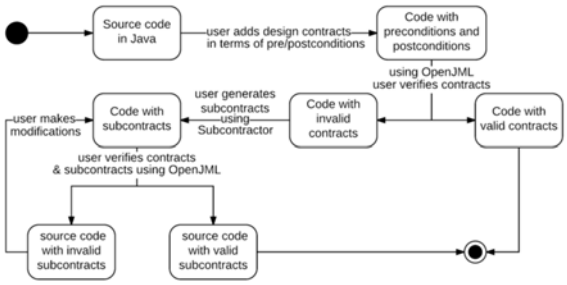

Figure 1: State diagram of the programming code in Subcontractor.

Figure 1 illustrates the life cycle of the programing code in response to different events with the support of Subcontractor. First, the programmer provides the source code and design contracts in terms of preconditions and postconditions. Then, OpenJML Runtime Assertion Checking (RAC) can be used to verify the preconditions, postconditions. If the contracts are valid, there is no need to add subcontracts. However, if the contracts are not valid, Subcontractor can be used to automatically generate and insert subcontracts for the programmer-selected if-statements and/or loop-statements. Again, OpenJML Runtime Assertion Checking (RAC) can be used to verify the preconditions, postconditions, and JML assert statements. The output after running OpenJML can facilitate finding the locations of the errors. Afterwards, the programmer can modify the code, and then uses OpenJML and Subcontractor to check the contracts again until the errors are fixed.

The rest of the paper is structured as follows: Section 2 discusses important concepts that are used to construct the tool Subcontractor. Section 3 describes the design and implementation of Subcontractor including how this tool is built upon Java libraries and Eclipse plugins. Section 4 presents an example using Subcontractor. Section 5 summarizes the purpose of Subcontractor and suggests multiple future enhancements.

2 BACKGROUND AND RELATED WORK

2.1 Programming Logic

Axiomatic semantics for a programming language is specified as programing logic, which has both axioms and inference rules. The axioms and inference rules are used for program verification, a static approach to reasoning about program correctness (Finkel, 1996). The axioms and inference rules are defined in the format of Hoare logic triple {P} C {Q} (Meyer, 1992), where P is the precondition, Q is the postcondition, and C is the implementation code. A Hoare triple for a program specification is valid if the precondition P is satisfied and the code C is executed. In this case, the postcondition Q is guaranteed to be true. When the precondition is false, the postcondition might be true or false (Hoare and Antony, 1969). This means the program does not have to guarantee anything when preconditions is not satisfied. While the axioms are for primitive statements, the inference rules are for compound statements. The inference rule for any sequence of statements is as in)) while the inference rules for if-statements and loop-statements are as in (2) and (3) respectively (Finkel, 1996).

$$\frac{\{P\}\ S1\ \{R\},\{R\}\ S2\ \{Q\}}{\{P\}\ S1;S2\ \{Q\}} \quad (1)$$

$$\frac{\{P \,\&\, B\}\ S1\ \{Q\},\{P \,\&\, (\text{not } B)\}\ S2\ \{Q\}}{\{P\}\ \text{if } B \text{ then } S1 \text{ else } S2\ \{Q\}} \quad (2)$$

$$\frac{P \Rightarrow Inv,\{Inv \,\&\, B\}\ S\ \{Inv\},Inv \,\&\, (\text{not } B) \Rightarrow Q}{\{P\}\ \text{while } B \text{ do } S\ \{Q\}} \quad (3)$$

Taking into consideration that *Inv* is the loop invariant. The loop invariant should be true before and after each execution of the loop body, but the loop invariant does not have to be always satisfied in the middle of each execution of the loop body (Hoare and Antony, 1969).

2.2 Design by Contract

DBC was first introduced by Bertrand Meyer, and was first implemented in Eiffel programming language (Meyer, 1992). DBC is a dynamic approach. A design contract for a class consists of preconditions and postconditions for methods, and class invariants. Preconditions, postconditions, and invariants are written as Boolean assertions that are evaluated at runtime. Preconditions are checked before entering the functions, while the postconditions are checked after the function

execution. In the case of class invariants, the conditions must be satisfied when the class instances are created, as well as before and after functions are executed (Meyer, 1992).

DBC adds a great value to the software development process, especially it improves the test phase by automatically checking the DBC assertions. As stated by Dijkstra (1970), "Program testing can be used to show the presence of bugs, but never to show their absence!" According to Meyer, using proper contracts help in validating against the program specifications defined in terms of its preconditions and postconditions, which leads to program correctness (Meyer, 1992). DBC also promotes the program robustness which means that programs can handle abnormal behaviour. Correctness and robustness together lead to software reliability, which is important to software security.

A commonly cited example to understand DBC is the client-supplier example (Meyer, 1992). Both parties have some obligations and they are expecting benefits in return. In DBC, function call serves as the client while the implemented function serves as the supplier. A precondition and a postcondition form a contract for a method between the caller and the callee. The function call, the client, needs to fulfil a precondition to expect the postcondition that needs to be guaranteed by the function, the supplier. For example, these obligations and benefits of a method that calculates the factorial of a number can be expressed in Table 1. Once the precondition is violated, the method should not be executed and an exception should be raised. However, if the precondition is satisfied, but the postcondition is not fulfilled, there are most likely errors in the implemented function.

Table 1: Client-supplier example of design by contract.

Party	Obligations	Benefits
Function Call (Client)	Provide numbers that are greater than or equal zero.	Get the factorial of the number that is provided.
Function Implementation (Supplier)	Compute the factorial of the number that is provided.	Do not need to calculate the factorial of the numbers that are less than zero.

DBC is not exclusively for constructing assertions for methods and classes. In general, DBC is a vital programming methodology that facilitates verification, testing, and debugging. Moreover, it is also a way to document the responsibilities of each method or routine. A design contract is an abstraction of the semantic of the code being developed. This documentation can help users understand the functionality of the code without seeing the implementation details.

2.3 Java Modelling Language

As other programming languages, Java does not support DBC as a built-in feature. Java has the basic assertion facility using the *assert* keyword. Since the simple assert facility is not sufficient to support DBC, there are different third party tools that support DBC for Java programing language, such as Java Modeling Language (JML) (Leavens, 2013), Jass (University of Oldenburg, 2001), and Contracts for Java (C4J) (Bergström, 2012). JML is chosen for the design and implementation of our tool Subcontractor for the following reasons: (1) JML is an open source DBC tool (Leavens, 2013). (2) Its implementation uses the preprocessor approach which supports the process of automatically generating and inserting subcontracts using inference rules. (3) JML is available as an Eclipse plugin. Eclipse is one of the most popular IDEs for Java programming language, and it offers a library that facilitates accessing and editing the source code (Eclipse, 2017).

JML is a specification language that addresses both the behavioral and the syntactic aspect of Java programming language. JML tools assist different fields such as static checking, DBC, JML compiler, and documentations (Leavens, 2013). As for DBC, JML expresses contracts using special annotation comments, /*@ ... @*/, for multiple lines. For a single line specification, the annotation starts with //@. Contracts are written in the form of comments to be ignored by Java compiler in case the specifications affect the performance negatively. When contracts are taken into consideration, JML compiler evaluates the contracts as Boolean expressions. The keyword *requires* specifies the precondition, while *ensures* clause determines the postconditions. To verify a contract at a specific point in the code, JML *assert* keyword can be used. If the preconditions of a method are not satisfied, the method will not be executed. On the other hand, if the preconditions are satisfied, the program will be in a proper pre-state. In this case, if the postconditions are satisfied, the program will be in a normal post-state, however if there is an exception in the code and the program does not terminate normally, the program will be in an exception post-state (Leavens, 2013).

OpenJML is the current version of the basic JML for the current Java. There are efforts to include

different features in this version. According to the official page of OpenJML (Cok, 2015), such features include Eclipse support, static and dynamic checking, and auto testing cases. OpenJML extends JML by introducing new expressions and more flexibility in the contracts for methods.

One advantage of JML is usability, since it extends Java programming language with more quantifiers. As mentioned earlier, there are a lot of efforts that are devoted to build tools that support different aspects of this specification language. These tools are for verifications, parsing, runtime checking, and testing (Leavens, 2013). Since JML has continuous progress and attention, this counts as another advantage.

JML deals with specifications using two different ways. Extended Static Checking (ESC) is a way to check specifications at compile time. ESC can be used to prove the correctness of the specifications (Burdy et al., 2005). First ESC checks the programming errors such as uninitialized variables and out-of-bounds index. Afterwards, ESC executes the specification of the code by the automated theorem prover (Zimmerman and Kiniry, 2009). Another method for checking the program contracts is Runtime Assertion Checking (RAC). RAC tests the correctness of the program at runtime. RAC converts the program specifications into assert statements. These assert statements are inserted into the program to form an instrumented program that can be used to check whether the contracts are violated during runtime (Nelson, 2004).

OpenJML (RAC) plugin is chosen for Subcontractor due to the following two considerations: (1) the dynamic checking approach is more practical for many programmers. (2) The dynamic checking approach is a direct support to DBC. Subcontractor uses OpenJML (RAC) to check the contracts originally provided by programmers and the subcontracts automatically generated by Subcontractor.

3 DESIGN AND IMPLEMENTATION

3.1 Flow Chart of Subcontractor

Figure 2 shows the flow chart of Subcontractor. There are two main functionalities in Subcontractor, which are respectively generation and deletion of subcontracts. When the user wants to generate subcontracts for selected parts in the programming code, the system traverses the code and stores the preconditions and postconditions along with the line numbers into *HashMaps*. Afterwards, every method in the code is visited to know the allowed line numbers that the contracts might appear in. The contracts in the corresponding allowed line numbers in the HashMaps are combined to form the subcontracts based on the inference rules of the selected parts in the code. Then, every selected loop-statement and if-statement is visited to insert the generated subcontracts.

When the user wants to delete the inserted subcontracts, there is a need to specify the methods that the user wants to delete the subcontracts from. The system traverses the code and deleted the subcontracts in the selected methods.

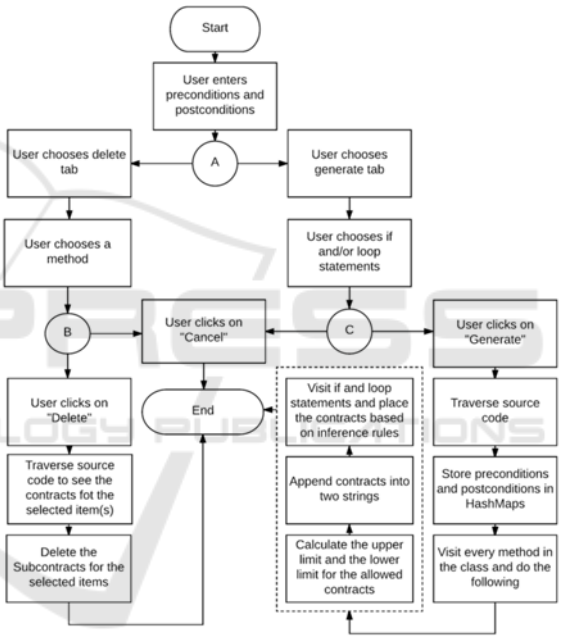

Figure 2: The flow chart of Subcontractor.

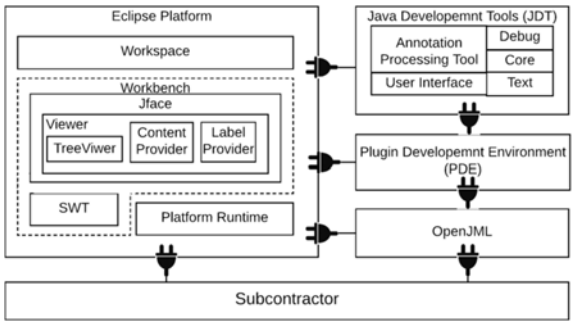

Figure 3: Subcontractor in Eclipse architecture.

3.2 Eclipse Components Used in Subcontractor

Eclipse is a popular Integrated Development Environment (IDE) with graphical user interfaces that facilitates the development process. The main language that is supported in Eclipse is Java, but there are other languages such as C++, C#, Perl, PHP, and JavaScript that can be supplemented using plugins (Eclipse, 2017).

One reason for choosing Eclipse is that it is an open source application that is built upon plugins. Thus, it is more straightforward to extend an IDE with a plugin than establishing a new IDE implementing the functionality of Subcontractor. In addition, according to RebelLabs' report in 2012 (ZeroTurnaround, 2012), Eclipse is the most used IDE for Java programming. Eclipse provides libraries that have the benefit of accessing and modifying the source code.

Figure 3 demonstrates the Eclipse architecture and the relationships between the plugin Subcontractor and Eclipse elements, which are presented in the following subsections. Figure 3 is based on the diagram that is presented in (Eclipse, 2017) with modification and extension to cover the scope of Subcontractor.

3.2.1 Plugin Development Environment

One of the neat functionalities in Eclipse is expandability. The user can add/extend a feature in Eclipse by adding a plugin. The Plugin Development Environment (PDE) is a plugin that can facilitate building other plugins, features, and Rich Client Platform (RCP) applications. When PDE is installed in the Eclipse platform, new functionalities can be added to support the plugin life cycle, such as establishing, debugging, running, maintaining, and publishing a plugin. As shown in Figure 3, PDE extends the functionalities of JDT, and the developed Subcontractor tool extends the capabilities of PDE, JDT, and OpenJML. Plugins can provide a variety of functional and non-functional requirements such as testing, diagramming, compiling, and more (Eclipse, 2017). The Eclipse marketplace website (Eclipse marketplace, 2017) has a myriad of plugins. The main advantage of having plugins is that it makes the component of this environment loosely coupled. Moreover, it maintains lightweight core application, since the user has the choice to install only the needed plugins.

3.2.2 Java Development Tools

Subcontractor uses the core of Java Development Tools (JDT) library to access and modify source code. In JDT core, the source code is represented by two different forms, a Java model or abstract syntax tree (AST). AST is an alternative to the Document Object Model (DOM) tree of an XML file. AST is more informative than a Java model. As a result, it takes longer to be created than the Java model (Eclipse JDT, 2017). Subcontractor handles the source code using AST because the other approach, the Java model, does not contain enough details about the source code. For example, in the Java model, the name, signature, and return type of a method can be retrieved. However, the content of that method cannot be provided, which is an important component for Subcontractor to analyze and to find the if-statements and loop-statements.

In Subcontractor, the AST of the source code can be traversed by extending ASTVisitor. Afterwards, an ASTNode can be accessed using a visit() method. Visitor pattern is one of the popular design patterns in software engineering. The visitor pattern can be used to represent an operation for different elements instead of representing the same operation for every element. One advantage of this pattern is that the operation can be extended without changing the elements. Another benefit of using the visitor pattern is that the operation is separated from the structure (Barclay and Savage, 2003).

Subcontractor uses OpenJML preconditions, postconditions, and assert statements that reside in comments. Comment, Expression, Statement and VariableDeclaration are different types of an ASTNode. According to Eclipse documentation (Eclipse AST, 2004), the nodes in AST could be any part of the code except the whitespaces and the comments. In Subcontractor, it is problematic to deal with comments since the AST is not preserving information about the comments. This issue can be addressed through calling *accept()* method for each comment in the list that is returned from *getCommentList()*.

In Subcontractor, after forming the subcontracts, the code needs to be edited to insert these subcontracts. One approach to editing the abstract syntax tree is prettyprinting, which is simple and easy to implement, as it does not maintain the locations of the comment and the source code in the modification process. The second approach uses textual edits. In this approach, each node has a corresponding offset and length. When changes are made, the modifications are applied to the AST

directly, or into ASTRewrite and then to the AST. This approach is more complicated to implement, but it preserves the code formatting (Overbey and Johnson, 2008). Subcontractor uses textual edits since it is important to preserve the locations of the comments in the source code.

3.2.3 JFace

Subcontractor uses the JFace library to create user interface components such as a TreeViewer and a button. JFace supports having dynamic content for the TreeViewer. A TreeViewer needs to be assigned to a data model (JFace, 2010). When the subcontracts need to be generated in Subcontractor, the TreeViewer dynamically presents the locations of if-statements and/or loop-statements within a method. In other words, for each method in the source code, TreeViewer displays the name of the method, and a list of if-statements and loop-statements in the order that they appear in the source code. The dialog box in Figure 5 is the corresponding TreeViewer for the source code in Figure 4. On the other hand, if the user chooses to delete the subcontracts, the TreeViewer illustrates the different methods.

For each TreeViewer, there is a need to assign a content provider to it. The content provider is responsible for traversing the data model and specifying the elements in parent/child structure for displaying purposes (Gast, 2015). After retrieving the content of the tree from the content provider, the label provider is called for each element to provide a string and icon, if there are any, then display the value in the tree. In other words, the label provider simply converts the content/data to image/text.

4 EXAMPLE USING SUBCONTRACTOR

Figure 4 shows an example of source code with preconditions and postconditions. This program does a simple withdraw or deposit operation for a bank account. The precondition is *(amount > 0)* while the postconditions are *((operation == "deposit") ==> balance == \old(balance) + amount)* or *((operation == "withdraw") ==> balance == \old(balance)-amount)*. There is an intentional error for testing in code line 15 which is: *this.balance = this.balance – amount*. The statement should be *this.balance = this.balance + amount*.

The output of OpenJML (RAC), as shown in Figure 7, indicates that the postcondition is not satisfied.

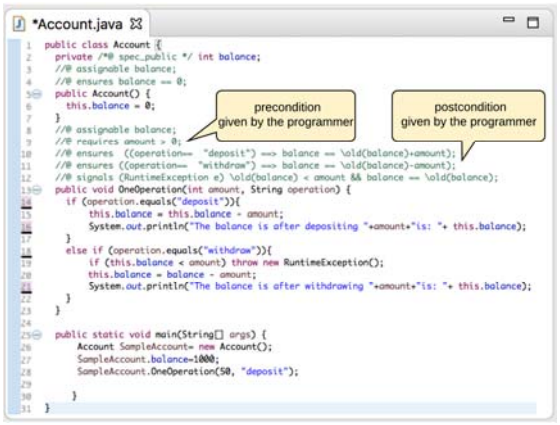

Figure 4: Source code of Account example with a precondition and postconditions.

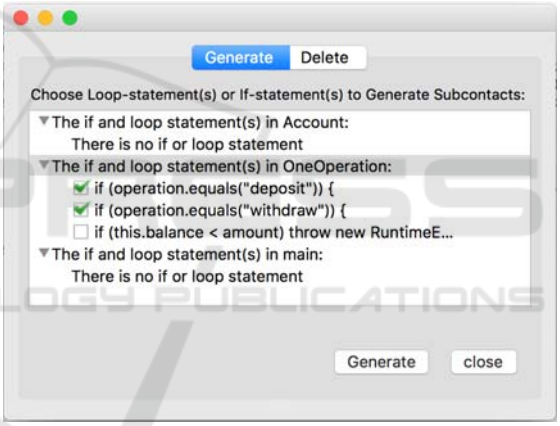

Figure 5: Dialog box that shows the chosen if-statements.

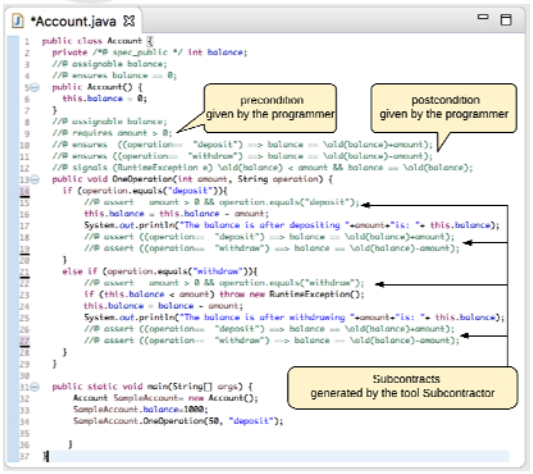

Figure 6: Source code of Account example after generating subcontracts.

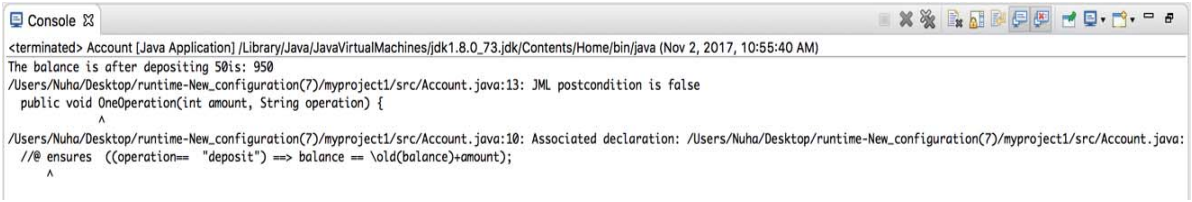

Figure 7: The output of OpenJML shows invalid postcondition.

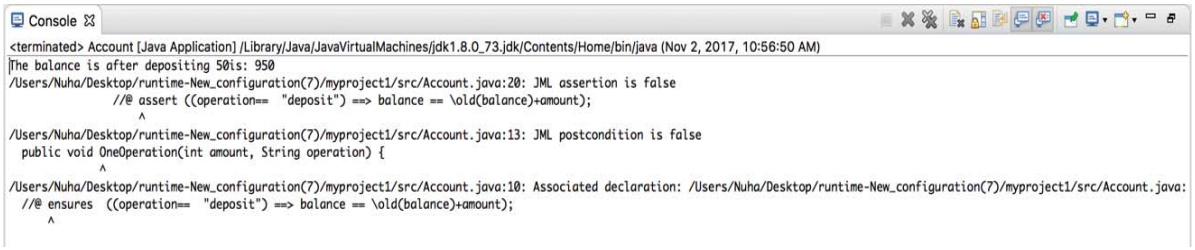

Figure 8: The output of OpenJML shows invalid postcondition and assert statement.

As shown in Figure 5, a dialog box will pop up after choosing to generate subcontracts. This action is triggered by right-clicking on the Java file. The TreeViewer represents three methods. Inside the second method, there is an if-else-if-statement that has a nested if-statement in the else-if part. It is important to note that ASTVisitor interface deals with the else-if part as a separate if-statement. Therefore, the viewer shows three if-statements. In this example, the user chose the first two if-statements.

Figure 6 shows the code after inserting the subcontracts automatically using the inference rules for if-statement. When comparing to Figure 4, there are four inserted subcontracts, or smaller verification tasks. The first one is *amount > 0 && operation.equals("deposit")* which is an instance of *{P and B}* in the inference rule for if-statement, see the inference rule as in (2) in Subsection 2.1. The third verification task is *amount > 0 && operation.equals("withdraw")* which is an instance of *{P and (not B)}* in the inference rule for if-statement, see also the inference rule as in (2) in Subsection 2.1. The second and last verification tasks are *((operation == "deposit") ==> balance == \old(balance) + amount)* or *((operation == "withdraw") ==> balance == \old(balance)-amount)* which are instances of *{Q}* in the inference rule for if-statement, see also the inference rule as in (1) in Subsection 2.1.

The output of OpenJML (RAC) is shown in Figure 8. When comparing to Figure 7, before subcontracts were automatically generated and inserted, there is an extra assert statement that is false. As a result, this assert statement facilitates finding the location of the error. In Figure 7, since there is a false contract that is at the beginning of the source code, there is a need to search in the whole program to find where the error is. On the other hand, in Figure 8, since there is an additional assert statement that is false, then the error should be between these localized false contracts.

5 CONCLUSION AND FUTURE WORK

Although DBC is a technique for improving the reliability of software by ensuring the program correctness and robustness, the effectiveness of DBC diminishes with large blocks of code that has invalid contracts. Subcontractor addresses this issue by generating and inserting subcontracts using inference rules for if-statements and loop-statements in the large pieces of code. After that, the contracts can be verified using the DBC open source tool named OpenJML (RAC). In conclusion, by combing design by contract and inference rules of programming logic, Subcontractor can facilitate error detection in code to improve program reliability, important to software security. More implementation details can be found in (Aldausari, N., 2017).

A number of future enhancements can provide additional features to Subcontractor. (1) Handling nested structures is an important functionality that can help the enhancement of the usability of Subcontractor. For instance, if there is a loop-statement that has if-statement inside it, there is a

need to generate the subcontracts based on the inference rule of the loop-statements first. Then, there is a need to use the inference rules for if-statement to further generate subcontracts. (2) The inclusion of inference rules for sequence statements would give additional subcontracts that facilitates detecting the errors in code. (3) Additional future enhancements can also include uploading the plugin Subcontractor in the Microsoft website rise4fun (Microsoft, 2017) to make Subcontractor accessible. The website rise4fun has a collection of software engineering tools. In this website, the user can experiment the uploaded tools without diving into the hassle of installing and configuring the tools. The website is categorized based on the institute that created the tools. The common factor of these tools is that they can be used in verification, testing, or security purposes. Examples of tools that are existing in this website are z3 and OpenJML (ESC).

REFERENCES

Aldausari, N., 2017. Enhancing A Design By Contract Tool Using Inference Rules of Programming Logic. In *Master Project*, California State University, Sacramento, 2017.

Barclay, K., Savage, J., 2003. Object-Oriented Design with UML and Java. In *Oxford: Elsevier Science*, 2003.

Bergström, J., 2012. Design By Contract for Java. http://c4j.sourceforge.net/.

Burdy, L., et al., 2005. An overview of JML tools and applications. In *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212-232, 2005.

Cok, D., 2015. OpenJML. http://www.openjml.org/.

Dijkstra, E. W., 1970. Notes on structured programming, Department of Mathematics, Technological University, 1970.

Finkel, R. A., 1996. Advanced programming language design. In *Addison-Wesley Reading*, 1996.

Gast, H., 2015. How to use objects: code and concepts. In *Addison-Wesley Professional*, 2015.

Gates, B., 2005. Gates Highlights Progress on Security, Outlines Next Steps for Continued Innovation. https://news.microsoft.com/2005/02/15/gates-highlights-progress-on-security-outlines-next-steps-for-continued-innovation/.

Hoare, C., Antony, R., 1969. An axiomatic basis for computer programming. In Communications of the ACM, vol. 12, no. 10, pp. 576-580, 1969.

Leavens, G., 2013. The Java Modeling Language (JML). http://www.eecs.ucf.edu/~leavens/JML//index.shtml.

Meyer, B., 1992. Applying design by contract. In *Computer*, vol. 25, no. 10, pp. 40-51, 1992.

Microsoft, 2017. rise4fun. http://rise4fun.com/

Nelson, G., 2004. Extended static checking for java. In *Springer International Conference on Mathematics of Program Construction*, pp. 1-1, 2004.

Overbey L., J., E. Johnson, R., 2008. Generating rewritable abstract syntax trees. In *Springer International Conference on Software Language Engineering*, pp. 114-133, 2008.

The Eclipse AST, 2014. Class AST. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FAST.html

The Eclipse Foundation, 2017. https://eclipse.org/.

The Eclipse Foundation, 2017. What is Eclipse? http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fint_eclipse.htm.

The Eclipse JDT, 2017. JDT Core Component. https://eclipse.org/jdt/core/.

The Eclipse JFace, 2010. https://wiki.eclipse.org/JFace

The Eclipse marketplace, 2017. Eclipse marketplace. https://marketplace.eclipse.org/.

University of Oldenburg, 2001. The Jass Page. http://csd.informatik.uni-oldenburg.de/~jass/.

ZeroTurnaround, 2012. Developer Productivity Report 2012: Java Tools, Tech, Devs & Data. https://zeroturnaround.com/rebellabs/developer-productivity-report-2012-java-tools-tech-devs-and-data/.

Zimmerman, D. M., R. Kiniry, J., 2009. A verification-centric software development process for Java. In *9th IEEE International Conference on Quality Software*, pp. 76-85.