Enhancing Secure Coding Assistant System with Design by Contract and Programming Logic

Wenhui Liang, Cui Zhang, and Jun Dai(✉)

Department of Computer Science, California State University, Sacramento 6000 J Street, Sacramento, CA 95819, USA
{wenhuiliang,zhangc,jun.dai}@csus.edu

Abstract. The system titled Secure Coding Assistant was developed to automate early detection for a subset of the Java secure coding rules specified by the SEI CERT at the Carnegie Mellon University. This system can help Java programmers significantly reduce security vulnerabilities in their code caused by the violations of secure coding rules. Since other software defects can also lead to security vulnerabilities, efforts have been taken to extend Secure Coding Assistant aiming at empowering programmers to detect, locate and remove code errors during coding time. This paper presents an enhancement to Secure Coding Assistant by a combination of Design by Contract and Programming Logic. Java programmers using this system are advised to provide their design contracts, i.e., logic assertions, for program structures of methods, if-then-else statements and while-loop statements. The design contracts defined by programmers can be automatically checked at the time of their program execution. To further facilitate the process of detecting and locating of code errors, using the programmers-defined design contracts, sub-design contracts can be automatically generated by the system based on the inference rules for the if-then-else statement and the while-loop statement in programming logic. The sub-design contracts generated by the system can also be automatically checked at dynamic time. In addition, based on the assignment axiom and the inference rule for the sequence statement in programming logic, the weakest pre-conditions of certain assignment sequences can be automatically generated from the post-conditions of the sequences, enabling programmers to statically analyze the correctness of the corresponding design contracts they specify. With the enhancement presented, Secure Coding Assistant can assist programmers for the early detections of not only secure coding rule violations but also errors in code. These early detections are performed in unison with the coding process to pursue software security.

Keywords: Secure coding · Software security · Design by contract · Programming logic

1 Introduction

Network attacks have become more and more common in recent decades. Cybersecurity Ventures predicts that in the next five years, the cost of global cybercrime will grow at

K.-K. R. Choo et al. (Eds.): NCS 2021, LNNS 310, pp. 124–140, 2022.
https://doi.org/10.1007/978-3-030-84614-5_10

an annual rate of 15%, reaching $10.5 trillion USD by 2025, more than $3 trillion USD in 2015 [4]. Since 2018, McAfee estimate the cost of cybercrime worldwide to be more than $1 trillion. They estimate that the money loss caused by cybercrime is about 945 billion US dollars. In addition, global cybersecurity spending is expected to exceed $145 billion by 2020. Today, it is a trillion dollar drag on the global economy [18]. Software enterprises face many challenges in diverse areas such as system security, application security, sensitive information protection. Developing software security is an inevitable trend.

To address the increasing needs for software security, the SEI CERT at the Carnegie Mellon University specified secure coding standards for a group of programming languages [15]. One of the standards is the SEI CERT Oracle Coding Standard for Java [17]. To facilitate the application of the Java secure coding rules to the software development practice, the system titled Secure Coding Assistant, that is available at http://benw408701.github.io/SecureCodingAssistant/, was developed at California State University Sacramento, in the Eclipse development environment, to automate the early detection for a subset of the Java secure coding rules during coding time [19, 20]. This system can help Java programmers significantly reduce security vulnerabilities in their code caused by the violations of secure coding rules. Since it is also a common knowledge that many other software defects can lead to security vulnerabilities, it is highly desirable to extend Secure Coding Assistant not only for detecting violations of secure coding rules but also for assisting programmers to detect, locate and remove errors in their code. It is also highly desirable to perform all these detections during coding time. To this end, efforts have been taken at California State University Sacramento in recent years through a number of projects.

Inspired by Bertrand Meyer's Design by Contract methodology effective for developing robust and reliable software products [10], one project was conducted to enhance Secure Coding Assistant by using Design by Contract [6]. This enhancement was implemented by integrating Secure Coding Assistant with an existing open-source software tool Cofoja that provides the functionality for Design by Contract [7]. Programmers using this enhanced system are advised to provide design contracts, i.e., logic assertions of pre-conditions and post-conditions, for the methods defined in their code. The programmers defined design contracts can then be automatically and dynamically checked to help programmers detect and remove defects in their code. This project demonstrated that detecting violations of secure coding rules and detecting other software defects can be automated or semi-automated in the same system. However, the program structures covered then needed to be extended, and how to help programmers detect and finally locate errors in lengthy code remained a challenge.

Another project separated from Secure Coding Assistant was conducted to experiment the combination of Design by Contract and Programming Logic [1]. A tool called Subcontractor for Java was built upon an open-source tool OpenJML for Design by Contract [13]. Programmers provided design contracts are automatically checked at dynamic time of their program execution. In addition, for program structures of if-then-statements and while-loop statements, this tool can generate sub-design contracts from the programmers provided design contracts. The automatic generation of sub-design

contracts is based on the inference rules for the if-then-else statement and the while-loop statement in programming logic [11, 14]. The tool generated sub-design contracts can be automatically inserted into the proper places in code for dynamic checking. This tool (i.e., Subcontractor) demonstrated the feasibility of combining Design by Contract and Programming Logic. By decomposing an original large program verification problem into smaller program verification problems, this tool can facilitate the process of detecting, locating and removing of code defects. However, this tool development was heavily relying on OpenJML, and making the detection of secure coding rule violations and the detection of software defects by the demonstrated combination both available in the Java development environment such as Eclipse remained a challenge.

Building upon the experiences of the above-mentioned two projects, a new project has been designed and conducted to address the challenges for the extension of Secure Coding Assistant. As a result, this paper presents an enhancement to Secure Coding Assistant by a combination of Design by Contract and Programming Logic. Java programmers using this enhanced system are advised to provide their design contracts of logic assertions for three program structures, i.e., methods, if-then-else statements and while-loop statements. For each method or if-then-else statement, its design contract is formed by its pre-condition and post-condition. For each while-loop statement, its design contract consists of its pre-condition, post-condition and loop-invariant. The programmers-defined design contracts can be automatically inserted into proper places in the code and checked at dynamic time. Based on the inference rules for the if-then-else statement and the while-loop statement in programming logic [11, 14], sub-design contracts can be automatically generated from the programmers-defined design contracts. The system-generated sub-design contracts can also be automatically inserted into code and dynamically checked. As an additional feature, based on the assignment axiom and the inference rule for the sequence statement [11, 14], the weakest pre-conditions of certain assignment sequences can be automatically generated from the post-conditions of the sequences, enabling programmers to statically analyze the correctness of the corresponding design contracts they defined. As for the implementation of this project, all functionalities extended are implemented mainly by augmenting source code and in a way cohesive with the implementation of the original system without using any open-source tools for Design by Contract. Compared with previous versions of the system, the current Secure Coding Assistant with the enhancement presented can better help programmers for the early detections not only for violations of secure coding rules but also for errors in code. All the detections are performed in unison with the coding process of software development to pursue software security.

2 Background and Related Work

2.1 Design by Contract

Bertrand Meyer's Design by Contract methodology was the first implemented in the programming language *Eiffel* [9, 10]. With the *Eiffel* programming environment, programmers are encouraged to specify design contracts during coding development. Each design contract is formed by logic assertions on obligations and benefits. The clients or users of the software functionality must meet the obligations specified. The suppliers

of the software functionality must provide the benefits specified if the obligations are met. The obligations and benefits are usually specified in terms of pre-conditions, post-conditions, and invariants. The *Eiffel* language provides programmers with key words to specify design contracts. The system automatically checks the programmers-defined design contracts at dynamic time of their program execution. This methodology significantly improves the robustness and reliability of software products. In addition, in *Eiffel* system, design contracts are used to create software documents that are semantically consistent with the software products finally delivered.

Java does not directly support Design by Contract as a built-in feature. However, there are several tools that are developed to provide Design by Contract for the Java programming language, such as Cofoja [7], iContract [5], Jass [3], and OpenJML [13].

2.2 Secure Coding Assistant

There are several existing tools that can help programmer to detect vulnerability of their code. However, most of these tools are close source static analysis tools and do not provide early detection of security vulnerabilities in the code. The first version of Secure Coding Assistant developed at California State University Sacramento is an open source tool based on early detection for static analysis of software security vulnerabilities [19, 20]. The design of this tool is inspired and based on a subset of the security rules for Java developed by SEI CERT at Carnegie Mellon University. Secure Coding Assistant automates the detection of violations of security rules during the time of coding development. Since its birth, the Secure Coding Assistant system has gone through a series of updates, including the automation of a subset of the SEI CERT securer coding rules for the C programming language [12, 16].

One of the important updates is the enhancement by the Design by Contract methodology. The open source tool Cofoja is integrated with Secure Coding Assistant to provide contract programming for Java programmers using the Eclipse development environment [6, 7]. This enhancement demonstrated that detecting violations of secure coding rules and detecting other software defects can be automated or semi-automated in the same system. However, this enhancement heavily relied on the open source tool Cofoja. Only the program structure of methods was covered. Furthermore, how to help programmers detect and finally locate errors in lengthy code was not addressed.

2.3 Subcontractor

Subcontractor is a tool that combines Design by Contract and Programming Logic [1]**.** It was implemented based on the open source tool OpenJML. In addition to dynamically checking programmers-defined design contracts, Subcontractor can help programmers locate logic errors in their large pieces of code by automatically generating and examining sub-design contracts. The sub-design contracts are generated based on inference rules for if-then-else statements and while-loop statements in programming logic [11, 14]. Subcontractor demonstrated the feasibility of combining Design by Contract and Programming Logic. This combination enables the decomposition of bigger program verification problems into smaller ones using the inference rules. However, this tool

development was heavily relying on OpenJML, and was yet to support the detection of secure coding rule violations.

3 The Enhancement to Secure Coding Assistant

3.1 Goal

This enhancement presented is to enable both detection of secure coding rule violations and detection of program errors by using a combination of Design by Contract and Programming Logic. The enhancement aims at providing the following:

- Advising programmers to provide their design contracts of logic assertions for three program structures, i.e., methods, if-then-else statements and while-loop statements. For each method or if-then-else statement, its design contract is formed by its pre-condition and post-condition. For each while-loop statement, its design contract consists of its pre-condition, post-condition and loop-invariant.
- Generating automatically sub-design contracts for if-then-else statements and the while-loop statements based on inference rules in programming logic [11, 14].
- Checking dynamically programmers-defined design contracts and system-generated sub-design contracts.
- Generating automatically the weakest pre-conditions of certain assignment sequences from the post-conditions of the sequences based on the assignment axioms and the inference rule for sequence statements in programming logic [11, 14], to help programmers statically analyze their design contracts.

3.2 Functionality

The tool can detect if programmers do not provide design contracts, i.e., pre-conditions, post-conditions and/or invariants to the methods, if-then-else statements or while-loop statements in the source code. If the design contracts for those structures are missing in the code, a marker will be created by the system and a dialog box will be prompted to advise programmers to specify their design contracts in the source code.

Functionality for Methods. As shown in Fig. 1, a method structure missing its design contract is detected and a dialog box is prompted to advise programmers to provide their pre-condition and post-condition for the method. The syntactic format for writing pre-condition starts with //@Precodition and is followed by the programmer-provided pre-condition. The syntactic format for writing post-condition is similar. When clicking on “Add precondition and postcondition,” the insertion of user-provided pre-condition and post-condition is performed.

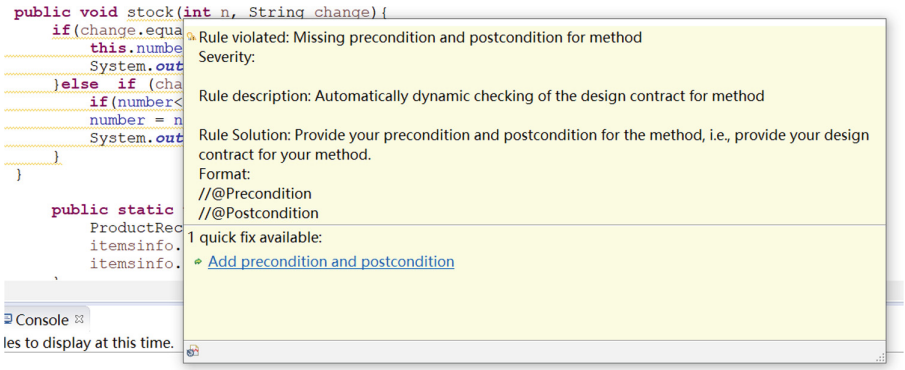


Fig. 1. Advice for the definition of design contracts for methods.

As shown in Fig. 2, after the programmer provides the pre-condition and post-condition for the method, the programmer can elect to automatically perform the dynamic checking of design contracts for the method. The system will generate code for overall pre-condition and post-condition checking based on the assertions given.

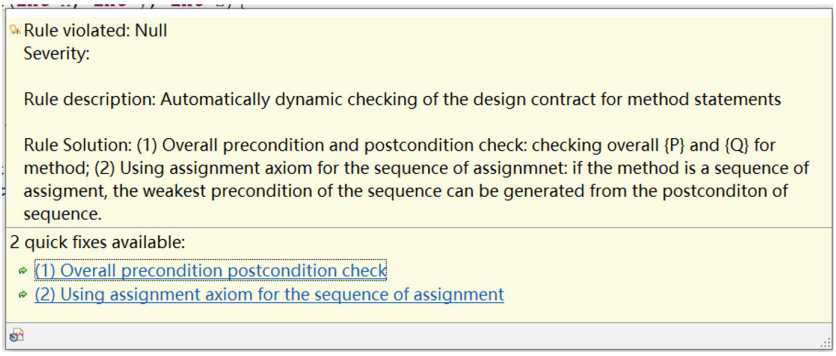


Fig. 2. Dynamic checking options for methods.

Functionality for If-Then-Else Statements. Figure 3 shows the dialog box created by the system when an if-then-else statement is detected missing its pre-condition and post-condition. The programmer is advised to provide the design contract for the statement.

The syntactic format for writing pre-condition of the if-then-else statement starts with //@Precodition and is followed by the programmer-provided pre-condition. However, more specific advice on writing the post-condition assertion is provided in the dialog. Programmers need to provide the data type of the variable if they need to use the key word *old_* followed by the variable name, to store the old value of the variable. The logic assertion before "or" is for the truth branch of the statement, and the assertion after "or" is for the false branch. When clicking on "Add precondition and postcondition," the insertion of user-provided pre-condition and post-condition is performed.

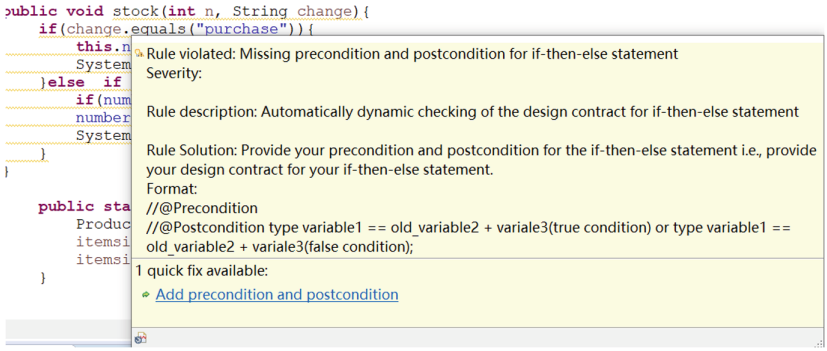


Fig. 3. Advice for the definition of design contracts for if-then-else statements.

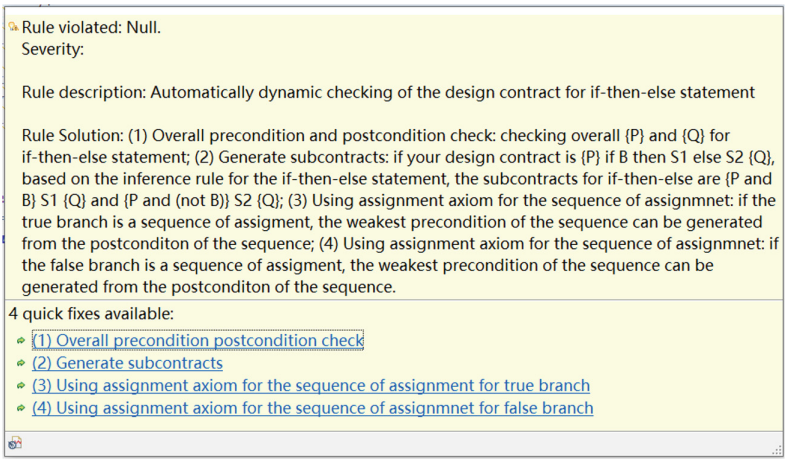


Fig. 4. Dynamic checking options for if-then-else statements.

As shown in Fig. 4, after the programmer provides the pre-condition and post-condition for the if-then-else statement, the programmer can use "Overall precondition and postcondition check" to dynamically check the design contracts provided by the programmer. If the programmer wants to further analyze the code to locate errors, electing "Generate subcontracts" will lead to the automatic generation of sub-design contracts, based on the inference rule shown below for the if-then-else statement in programming logic [11, 14].

$$\frac{\{P \wedge B\}\, S1\, \{Q\},\ \{P \wedge (not\ B)\}\, S2\, \{Q\}}{\{P\}\ if\ \ B\ \ them\ S1\ \ else\ \ S2\, \{Q\}}$$

In the rule above, {P} and {Q} indicate the pre-condition and post-condition given by the programmer. The two sub-design contracts generated by the system for if-then-else statement are $\{P \wedge B\}\, S1\, \{Q\}$ and $\{P \wedge (not\ B)\}\, S2\, \{Q\}$. B is the condition of the if-then-else statement.

After the programmer elects "Overall precondition and postcondition check" and/or "Generate subcontracts", the system will generate code for the overall pre-condition and post-condition checking based on the assertions given by the programmer, and also generate code for the dynamic checking of the sub-design contracts created.

Functionality for While-Loop Statements. For a while-loop statement, there is an additional logic assertion called loop-invariant in the design contract. When a while-loop statement is detected missing its pre-condition, post-condition, and loop-invariant, a dialog box as shown in Fig. 5 is prompted, to advise the programmer to define the design contract for the statement. The syntactic format for writing the design contracts for while-loop statements is similar to that for methods and if-then-else statement. When clicking on "Add precondition, postcondition and invariant," the programmer-provided pre-condition, post-condition and loop-invariant are inserted to the code.

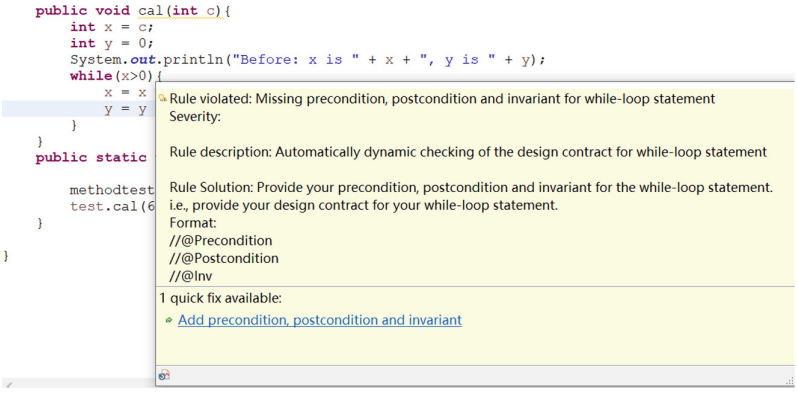


Fig. 5. Advice for the definition of design contracts for while-loop statements.

As shown in Fig. 6, after the programmer provides the pre-condition, post-condition and loop-invariant for the while-loop statement, the programmer can use "Overall pre-condition and postcondition check" to dynamically check the design contracts provided by the programmer. If the programmer wants to further analyze the code to locate errors, electing "Generate subcontracts" will lead to the automatic generation of sub-design contracts, based on the inference rule shown below for the while-loop statement in programming logic [11, 14].

$$\frac{P \Rightarrow Inv,\ \{Inv \wedge B\}\, S\, \{Inv\},\ Inv \wedge (not\ B) \Rightarrow Q}{\{P\}\ while\ B\ do\ S\ \{Q\}}$$

In the rule above, {P}, {Q}, and Inv indicate the pre-condition, post-condition, and the loop-invariant provided by the programmer. The three sub-design contracts generated by the system for while-loop statement are $P \Rightarrow Inv$, $\{Inv \wedge B\}\, S\, \{Inv\}$, and $Inv \wedge (not\ B) \Rightarrow Q$. $P \Rightarrow Inv$ must be satisfied before the execution enters the loop, $\{Inv \wedge B\}\, S\, \{Inv\}$ ensures the preservation of the loop-invariant property after each time of executing the loop body. $Inv \wedge (not\ B) \Rightarrow Q$ must be satisfied right after the execution of the entire loop statement.

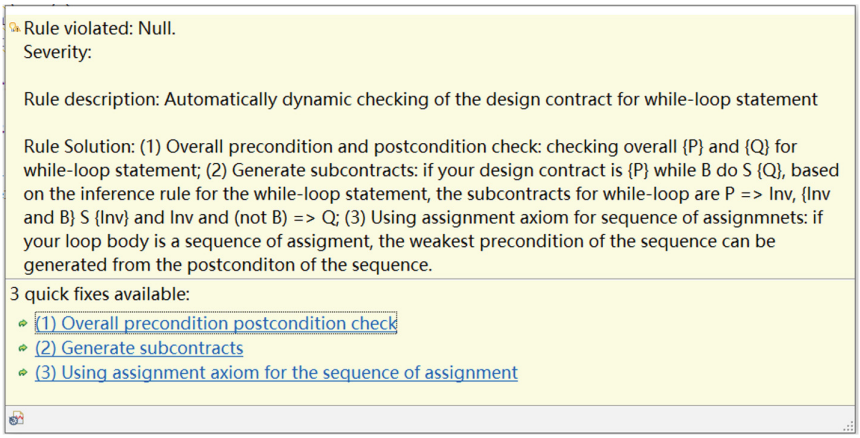


Fig. 6. Dynamic checking options for while-loop statements.

Functionality for a Sequence of Assignments. If the body of a method, a branch of an if-then-else statement, or a body of a while-loop statement is a sequence of assignments, there is an additional option called "Using assignment axiom for sequence of assignment", available in the dialog boxes as shown in Fig. 2, Fig. 4 and Fig. 6. This option will not be available for other types of code structures. The axiom with backward method for assignments and the inference rule for the sequence statements are used to generate the weakest pre-condition of the sequence from the post-condition of the sequence. Below are the axiom for assignments [11, 14] and the inference rule of sequence of statements [11, 14] used in this process:

$$\{P\}v := E\{Q\}$$

$$\frac{\{P\}\,S1\,\{R\},\ \{R\}\,S2\,\{Q\}}{\{P\}\,S1;\ S2\,\{Q\}}$$

3.3 Workflow

As a system for early detection, Secure Coding Assistant can provide immediate feedback to programmers. It runs in the background to monitor code changes and detect rule conflicts [19, 20]. Abstract Syntax Tree (AST) is used to represent the structure of the source code for analysis. When a rule violation is detected, a marker is created where the violation takes place. Any subsequent code changes will clear all existing markers and trigger a new round of AST node traversal [6].

Figure 7 shows the workflow of functionalities extended to the Secure Coding Assistant. Specifically, this enhancement to Secure Coding Assistant inherits the previous design for all monitoring in the background and AST node traversal. The function *violated()* in Secure Coding Assistant travels across AST nodes. If there is a method, an if-then-else statement or a while-loop statement, a marker will be created. If any design

contract, i.e., pre-condition, post-condition and/or loop-invariant, is detected missing, the system will advise programmers to define the design contract in the form of logic assertions. After logic assertions are provided, the function *getSolutions()* of Secure Coding Assistant will be called to provide programmers with the option of inserting code for dynamic checking of programmer-defined overall design contracts. If the execution of the program does not lead to expected results, and if the programmer wants to further analyze the code to locate errors, the system provides another option, called "Generate subcontracts", to generate sub-design contracts for if-the-else statements and while-loop statements. The code for this dynamic checking will be automatically inserted into the programmer's code. When the augmented code is executed in the Eclipse environment, the programmer can analyze the result of the dynamic checking of all the assertions. If the body of a method, a branch of an if-then-else statement, or a body of a while-loop statement is a sequence of assignments, the function *getSolutions()* is called to provide programmers with the option of generating the weakest pre-condition of the sequence from the post-condition of the sequence.

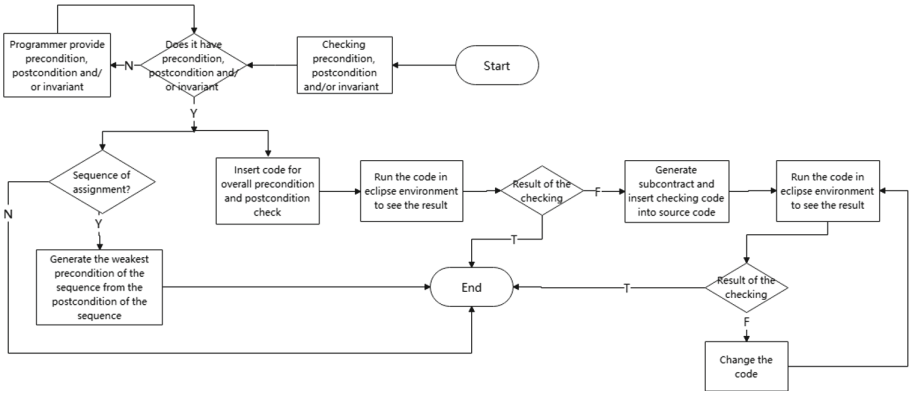


Fig. 7. Workflow of functionalities extended to the Secure Coding Assistant.

3.4 Implementation

To implement Design by Contract, the key words *@Precondition*, *@Postcondition* and *@Inv* are defined, and programmer can use comments to write logic assertions to the source code if needed. Functions in Java's *IWorkbenchWindow* interface are used to get active page for the workbench window and the Java function *getPath()* is used to get the path of the source code. Eclipse JDT *ASTParser* is used to parse Java source files by using the path and the Java function *getCommentList()* from the *CompilationUnit* interface is used to get comments. The Java function *accept*() is used to make the comments accessible. The Java function *visit()* from ASTVisitor class is overridden to store pre-condition, post-condition and/or loop-invariant into the Java *Map* structure. The function *violated()* in Secure Coding Assistant is overridden to identify methods, if-then-else

statements or while-loop statements in the source code. If one of these structures is identified, a marker will be created in that location.

To insert code for dynamic checking of the overall pre-condition and post-condition checking, and to insert code for dynamic checking of the system-generated sub-design contracts, source code needed to be edited. The Java function *getASTRewrite()* from *ASTRewrite* class is used in Secure Coding Assistant function *getSolutions()*, which is rewritten to insert the generated code into the proper places in the source code. The *getASTRewrite()* returns an ASTRewrite instance from which this ListRewriter was created [2]. The Java functions *insertFirst()*, *insertLast()*, *insertAfter()* and *insertBefore()* are used to insert code for dynamic checking of logic assertions to their corresponding location.

More design and implementation details for the enhancement presented are available from the project report [8].

4 Examples

To evaluate the enhancement presented in this paper, we run the system for several examples. Below please find illustrative results for an if-then-else statement, a while-loop statement, and a sequence of assignments statements.

Example for an If-Then-Else Statement. An example of an if-then-else statement is shown in Fig. 8. Following the system advice, the programmer provides the pre-condition and the post-condition for this if-then-else statement.

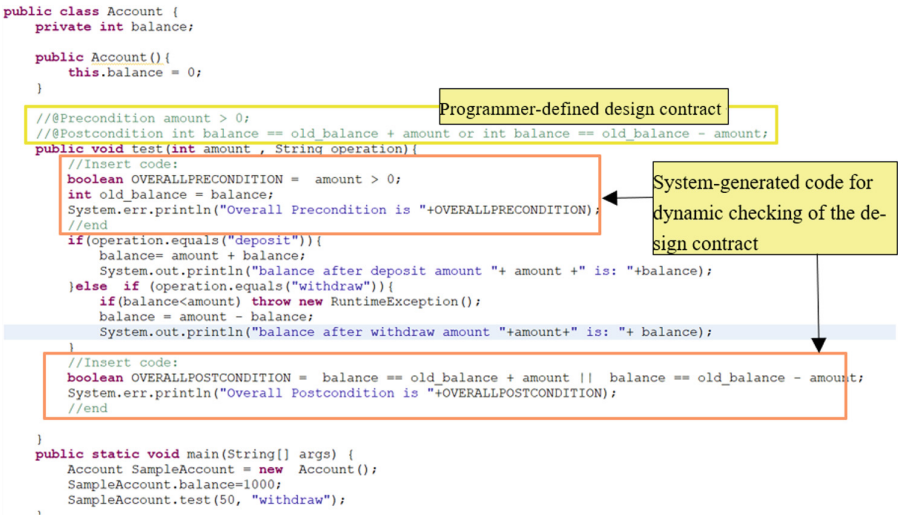
```
public class Account {
    private int balance;

    public Account(){
        this.balance = 0;
    }
    //@Precondition amount > 0;                    Programmer-defined design contract
    //@Postcondition int balance == old_balance + amount or int balance == old_balance - amount;
    public void test(int amount , String operation){
        if(operation.equals("deposit")){
            balance= amount + balance;
            System.out.println("balance after deposit amount "+ amount +" is: "+balance);
        }else  if (operation.equals("withdraw")){
            if(balance<amount) throw new RuntimeException();
            balance = amount - balance;
            System.out.println("balance after withdraw amount "+amount+ " is: "+ balance);
        }
    }
    public static void main(String[] args) {
        Account SampleAccount = new  Account();
        SampleAccount.balance=1000;
        SampleAccount.test(50, "withdraw");
    }
```

Fig. 8. Source code with programmer-defined design contract for an if-then-else.

If the programmer clicks the option called “Overall precondition and postcondition check” as shown in Fig. 4, the system will automatically generate code for dynamical checking of the design contract. Figure 9 shows that this code along with system-generated comments is inserted into the original code. When the programmer executes the augmented code in Eclipse, the result of the overall pre-condition and post-condition

checking is given in Fig. 10. The result shows that the pre-condition is true, the post-condition is false, and the result of the program execution is −950 that is not correct. This means that there is some error either in the code or in the design contract.




```
public class Account {
    private int balance;

    public Account(){
        this.balance = 0;
    }
    //@Precondition amount > 0;
    //@Postcondition int balance == old_balance + amount or int balance == old_balance - amount;
    public void test(int amount , String operation){
        //Insert code:
        boolean OVERALLPRECONDITION =  amount > 0;
        int old_balance = balance;
        System.err.println("Overall Precondition is "+OVERALLPRECONDITION);
        //end
        if(operation.equals("deposit")){
            balance= amount + balance;
            System.out.println("balance after deposit amount "+ amount +" is: "+balance);
        }else  if (operation.equals("withdraw")){
            if(balance<amount) throw new RuntimeException();
            balance = amount - balance;
            System.out.println("balance after withdraw amount "+amount+" is: "+ balance);
        }
        //Insert code:
        boolean OVERALLPOSTCONDITION =  balance == old_balance + amount ||  balance == old_balance - amount;
        System.err.println("Overall Postcondition is "+OVERALLPOSTCONDITION);
        //end

    }
    public static void main(String[] args) {
        Account SampleAccount = new  Account();
        SampleAccount.balance=1000;
        SampleAccount.test(50, "withdraw");
    }
```

Fig. 9. Code with pre-condition and post-condition checking for an if-then-else.

```
Tasks  Console
<terminated> Account [Java Application] D:\java\jdk1.8.0_65\bin\javaw.exe
Overall Precondition is true
Overall Postcondition is false
balance after withdraw amount 50 is: -950
```

Fig. 10. Result of pre-condition and post-condition checking for an if-then-else.

To locate the error, the programmer can click the option called “Generate subcontracts” as shown in Fig. 4. The system will then automatically generate the code for the dynamic checking of sub-design contracts based on the inference rule for if-then-else statements. This code along with related comments generated by the system is inserted into the original code. Figure 11 presents the original code and the inserted code for dynamic checking of the system-generated sub-design contracts. Due to the space limit, the inserted code for the dynamic checking of the programmer-provided design contract is removed from Fig. 11 in this paper. Figure 12 shows that actually the false branch of this if-then-else statement is executed for the dynamic checking of system-generated sub-design contract. As the result, its pre-condition is true and its post-condition is false. This means that the dynamic checking of the sub-design contract helps the programmer locate the error in the false branch of this if-then-else statement. The programmer needs to analyze the design contract and the false branch code to remove the defect. Figure 13 shows the expected correct result of the dynamic checking of this false branch after the programmer changes the line of the code “*balance* = *amount* – *balance*” to “*balance* = *balance* – *amount*”.

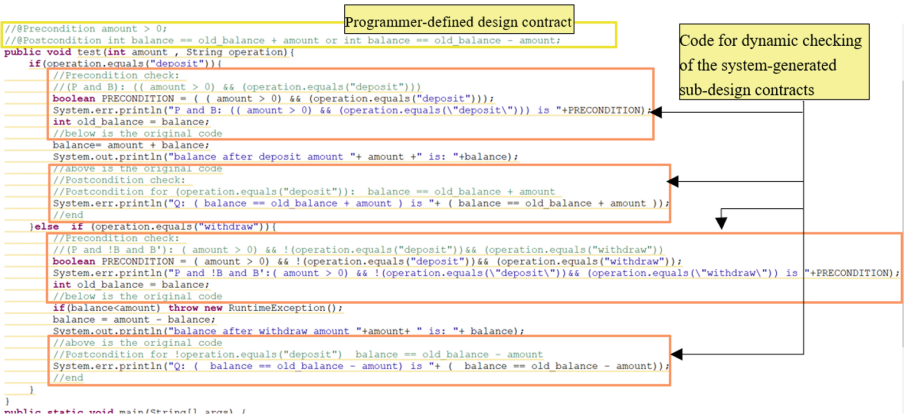


Fig. 11. An if-then-else statement with sub-design contracts.

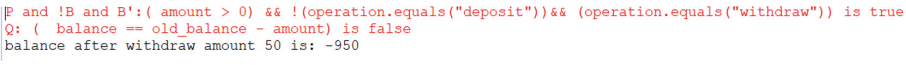


Fig. 12. Result of sub-design contract checking for an if-then-else.

Fig. 13. Result of sub-design contract checking for an if-then-else after the code is changed.

Example for a While-Loop Statement. An example of a while-loop statement is shown in Fig. 14. Following the system advice, the programmer provides the pre-condition, the post-condition, the loop-invariant for this while-loop statement.

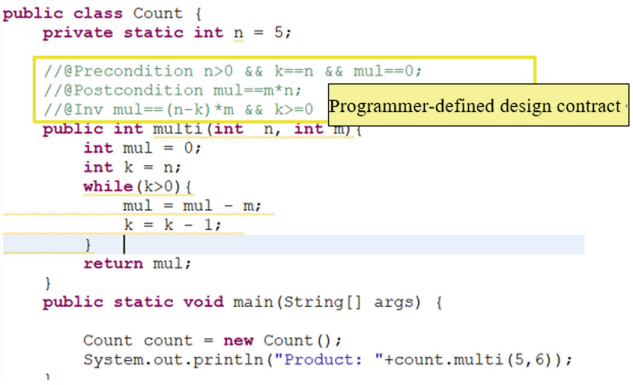


Fig. 14. Java source code with programmer-defined design contracts for a while-loop.

If the programmer clicks the option called "Overall precondition and postcondition check" as shown in Fig. 6, the system will automatically generate code for dynamical checking of the design contract. Similar to the above-given example for if-then-else statement, this code along with system-generated comments is inserted into the original code. When the programmer executes the augmented code in Eclipse, the result of the overall pre-condition and post-condition checking is given in Fig. 15. The dynamic checking result shows that the pre-condition is true, the post-condition is false, and the result of the program execution is -30 that is not the correct one. This means there is some error either in the design contract or in the code.

```
Overall Precondition is true
Overall Postcondition is false
Product: -30
```

Fig. 15. Result of pre-condition and post-condition checking for a while-loop.

To locate the error, the programmer can click the option called "Generate subcontracts" as shown in Fig. 6. The system will then automatically generate the code for the dynamic checking of sub-design contracts based on the inference rule for while-loop statements. This code along with related comments generated by the system is inserted into the original code. Figure 16 presents the original code and the inserted code for dynamic checking of the system-generated sub-design contracts. Due to the space limit, the inserted code for the dynamic checking of the programmer-provided design contract is removed from Fig. 16. Figure 17 shows that $P \Rightarrow Inv$ is true, $\{Inv \wedge B\}S\{Inv\}$ is not satisfied for every time of executing the loop body, and $Inv \wedge (not\ B) \Rightarrow Q$ is true. This means the dynamic checking of the sub-design contract helps the programmer locate the error in this loop body. The programmer needs to analyze the design contract and the code for the loop body to remove the defect. Figure 18 shows the expected correct result of the dynamic checking of this loop statement after the programmer changes the line of the code "$mul = mul - m$" to "$mul = mul + m$".

Example for a Sequence of Assignments. Figure 19 shows an example of using the assignment axiom and the inference rule for the sequence of statements to automatically generate the weakest pre-condition of an assignment sequence from the given post-condition of the sequence. The backward method associated with the assignment axiom is used in the process. For this example, the generated weakest pre-condition is equivalent to the programmer-defined pre-condition. In general, there should be an implication relationship from the programmer-defined pre-condition to the system-generated weakest pre-condition. If this logic relationship is not satisfied, the programmer needs to analyze and then correct the pre-condition, post-condition, and/or code. This analysis and correction are performed statically.

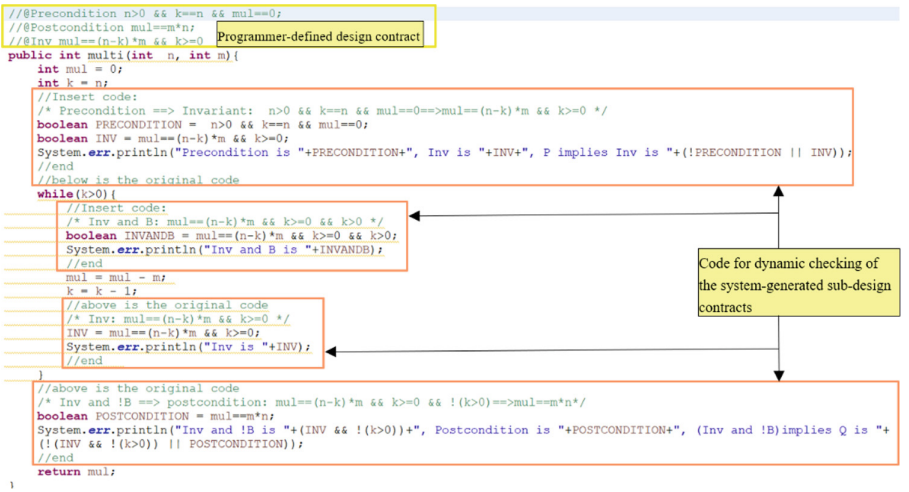


Fig. 16. A while-loop statement with sub-design contracts.

```
Precondition is true, Inv is true, P implies Inv is true
Inv and B is true
Inv is false
Inv and B is false
Inv is false
Inv and B is false
Inv is false
Inv and B is false
Inv is false
Inv and B is false
Inv is false
Inv and !B is false, Postcondition is false, (Inv and !B)implies Q is true
Product: -30
```

Fig. 17. Result of sub-design contract checking for a while-loop.

```
Precondition is true, Inv is true, P implies Inv is true
Inv and B is true
Inv is true
Inv and B is true
Inv is true
Inv and B is true
Inv is true
Inv and B is true
Inv is true
Inv and B is true
Inv is true
Inv and !B is true, Postcondition is true, (Inv and !B)implies Q is true
Product: 30
```

Fig. 18. Result of sub-design contract checking for a while-loop after the code is changed.

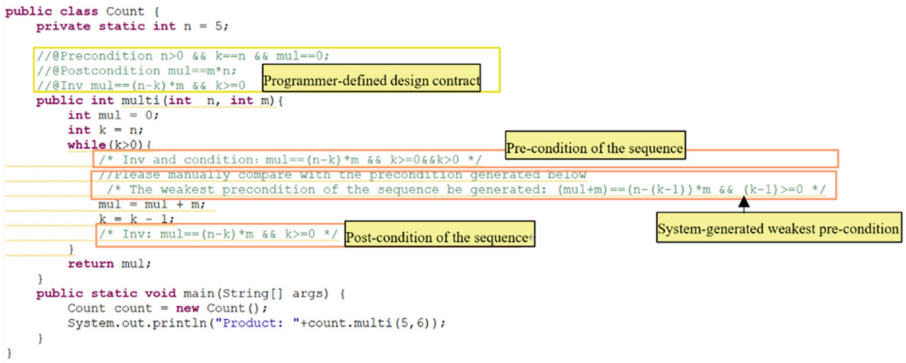


Fig. 19. Result of using assignment axiom and inference rule of sequence statements for a sequence of assignments.

5 Conclusion and Future Work

The enhanced Secure Coding Assistant supports early detection of secure coding rule violations as defined by SET CERT, and the early detection of code defects based on a combination of Design by Contract and Programming Logic. It can help programmers improve software security by removing potential security vulnerabilities during coding. The future work will focus on the nested code structures, such as nested while-loop statements and if-then-else statements. The challenge is to repeatedly or recursively generate sub-design contracts for nested code structures based on inference rules.

References

1. Aldausari, N., Zhang, C., Dai. J.: Combining design by contract and inference rules of programming logic towards software reliability. In: Proceedings of SECRYPT 2018 (2018)
2. ASTRewrite. https://www.ibm.com/support/knowledgecenter/ko/SSZHNR_1.0.0/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/ListRewrite.html
3. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass — Java with assertions. Electron. Notes Theor. Comput. Sci. **55**(2), 103–117 (2001)
4. Cybercrime Facts and Statistics. https://1c7fab3im83f5gqiow2qqs2k-wpengine.netdna-ssl.com/wp-content/uploads/2021/01/Cyberwarfare-2021-Report.pdf
5. Kramer, R.: iContract - the Java(tm) design by contract(tm) tool. In: Proceedings of the Technology of Object-Oriented Languages and Systems (1998)
6. Li, C., Dai. J., Zhang, C.: Enhancing secure coding assistant with error correction and contract programming. In: Proceedings of the National Cyber Summit, 6–8 June 2017 (2017)
7. Le, N.M.: Cofoja github page. http://github.com/nhatminhle/cofoja
8. Liang, W.: Combining design by contract and programming logic to enhance secure coding assistant system. MS Project Report, California State University, Sacramento, May 2021
9. Meyer, B.: Eiffel: a language for software engineering. Technical Report TR-CS-85-19 University of California, Santa Barbara (1985)
10. Meyer, B.: Applying 'design by contract.' Computer **25**(10), 40–51 (1992). https://doi.org/10.1109/2.161279

11. Meyer, B.: Introduction to the Theory of Programming Languages. Prentice Hall, Hoboken (1990)
12. Melnik, V., Dai, J., Zhang, C., White, B.: Enforcing secure coding rules for the C programming language using the eclipse development environment. In: Choo, K.-K.R., Morris, T.H., Peterson, G.L. (eds.) NCS 2019. AISC, vol. 1055, pp. 140–152. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-31239-8_12
13. OpenJML. http://www.openjml.org/
14. Slonneger, K., Kurtz, B.L.: Formal Syntax and Semantics of Programming Languages. Addison Wesley, Boston (1995)
15. SEI CERT Coding Standards. https://wiki.sei.cmu.edu/confluence/display/seccode
16. SEI CERT C Coding Standard. https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard
17. SEI CERT Oracle Coding Standard for Java. https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java
18. The Hidden Costs of Cybercrime. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-hidden-costs-of-cybercrime.pdf
19. White, B., Dai. J., Zhang, C.: Secure coding assistant: enforcing secure coding practices using the eclipse development environment. National Cyber Summit (2016)
20. White, B., Dai, J., Zhang, C.: An early detection tool in eclipse to support secure coding practices. Int. J. Inf. Priv. Secur. Integr. **3**(4), 284–309 (2018)