Context-Dependent Extensible Syntax-Oriented Verifier with Recursive Verification

Nhor Sok Lang
University of Tokushima
Course of Info Sci. & Intel. Syst.
Tokushima
JAPAN
soklang@is.tokushima-u.ac.jp

Takao Shimomura
University of Tokushima
Dept. of Info Sci. & Intel. Syst.
Tokushima
JAPAN
simomura@is.tokushima-u.ac.jp

Quan Liang Chen
University of Tokushima
Course of Info Sci. & Intel. Syst.
Tokushima
JAPAN
quan@is.tokushima-u.ac.jp

Kenji Ikeda
University of Tokushima
Dept. of Info Sci. & Intel. Syst.
Tokushima
JAPAN
ikeda@is.tokushima-u.ac.jp

Abstract: In order to develop Web applications of high quality, it is important to apply efficient frameworks to standardize the process of development in projects, or apply useful design patterns to produce the program code that can easily be enhanced. However, it is not enough. In addition to these efforts, we have to check the programs to see whether they keep various kinds of rules such as verification items for security which are common for all kinds of Web applications, and verification items for coding styles or code conventions which are pertain to each project. This paper proposes a verification method for the syntax-oriented verifier, and describes the implementation of its prototype system, SyntaxVerifier. SyntaxVerifier makes it possible to detect syntactical objects based on their syntactic contexts. It realizes a recursive verification which makes it easy to dynamically trace a syntax tree in a verification process itself.

Key–Words: Customaizable, Extensible, Recursive, Syntax analysis, Verification

1 Introduction

In order to develop Web applications of high quality, it is important to apply efficient frameworks to standardize the process of development in projects [1], [2], [3], [4], or apply useful design patterns to produce the program code that can easily be enhanced [5]. However, it is not enough. In addition to these efforts, we have to check the programs to see whether they keep various kinds of rules such as verification items for security which are common for all kinds of Web applications [6], [7], [8], and verification items for coding styles or code conventions which are pertain to each project [9]. Tools that assist these kinds of verification are often incorporated into a part of integrated development environments [10]. However, in most of the existing tools [11], [12], [13], their verification items are determined in advance, or even if verification items can be added, they only permit addition of the conditions of a part of program code to be verified. Moreover, it is difficult to add verification items that check a sequence of program code that appear at multiple locations of the programs. Web application programs consist of servlets, Java classes, JSP, HTML, JavaScript, and others. The authors developed a pattern-oriented verifier [14] to make it easy to verify JSP pages containing HTML, Java, and JavaScript code. On the other hand, servlets and Java classes are programs written in Java language. Verifying these programs only by using patterns has its limitations. To supplement the previous verifier and incorporate it, we have developed a new verifier which is syntax-oriented. This paper proposes a verification method for the syntax-oriented verifier, and describes the implementation of its prototype system, SyntaxVerifier. SyntaxVerifier makes it possible to detect syntactical objects based on their syntactic contexts. It realizes a recursive verification which makes it easy to dynamically trace a syntax tree in a verification process itself.

2 Verification method

2.1 Definition of verification items

2.1.1 Verification conditions

What items should be verified depends on programming languages, applications, and projects. As for verification items related to program syntax, for example, there are some such as (1) Variable names should start with lowercase letters; (2) Comparative operators such as $>=$ and $>$ should be replaced with $<=$ and $<$, respectively [9]; or (3) The number of methods included in a class should not exceed 30 [15]. When a source program is syntactically analyzed, an abstract syntax tree will be obtained. The abstract syntax tree is a tree consisting of non-terminal symbols (such as methodDef and if) and terminal symbols (such as Identifier and Literal). Syntax-oriented verification is made by traversing this abstract syntax tree. In the SyntaxVerifer, instead of a node of the syntax tree, the following verification condition is specified as a target object to be verified in the abstract syntax tree.

node node ... Terminal Terminal ... [with] Modifiers(*modifier modifier ...*)

Node is a non-terminal symbol in the abstract syntax tree, and *Terminal* is a terminal symbol. This verification condition makes it possible to verify terminal symbols in a specific context. For example, if "if cond Binary" is specified as a verification condition, only binary operators in the conditional parts of if statements can be verified. Verification condition "methodDef varDef Identifier" lets local variables in method definitions be target objects to be verified. A terminal symbol is specified as *Terminal* or *Terminal*(*value*). *Terminal*(*value*) lets a terminal symbol that has a specific *value* be a target object. For example, "Binary(GT) Binary(GE)" lets binary operators $>$ and $>=$ be target objects. "Modifiers(*modifier modifier ...*)" specifies the modifiers (such as private and static) that are attached to the terminal symbols. This lets only the terminal symbols that have those modifiers be target objects. For example, "Modifiers(static private)" allows only private and static variables or methods to be verified.

2.1.2 Verification items

After a target symbol is detected from a program, whether it is appropriately written will be judged. This judgment is made by a verifier class. The verifier class judges whether the detected symbol is appropriately written in the program. Verification items are defined in verification-item definition files. Figure

```
Lowercase Variable Name
LowercaseVariableName
varDef Identifier
```

(a) Definition of verification item

```
package ver;
import javacProcess.ver.*;
public class LowercaseVariableName extends Verifier {
    public void verify(String terminalSymbol,
    String verifiedText, String[] modifiers, int lineNo,
    int columnNo) throws Exception {
        addMessage(verifiedText + " (Line " + lineNo +
        ", Column " + columnNo + ")");
        for (int i = 0; i < modifiers.length; i++) {
            if (modifiers[i].equals("final")) {
                return;
            }
        }
        char c = verifiedText.charAt(0);
        if (!Character.isLowerCase(c)) {
            addProblem();
        }
    }
}
```

(b) Verification class

Figure 1: Definition of a verification item

1 (a) shows a verification-item definition file for variable name verification. It contains a verification-item name, a verifier class name, and a verification condition. Because the verification condition is "varDef Identifier", all variable names in the program will be verified. When a variable is detected in a variable definition of the program, it will be passed to LowercaseVariableName verifier class to be verified.

2.2 Verification Processes

2.2.1 A flow of verification processes

The system syntactically analyzes a program, obtains an abstract syntax tree, detects target symbols, which are specified as a verification condition, and then invokes a verifier class. Figure 2 illustrate a flow of these verification processes. (1) The SyntaxVerifier system reads all verification-item definition files *.ver in the conf/ directory, and builds the menu items of the "Verifiers" menu. (2) When a user reads a source program and (3) they choose one of the menu items of the "Verifiers" menu, the system will display a verifier window that corresponds to the chosen verification item. (4) The system starts a Java compile. (5) The Java compiler reads again the verification-item definition file that corresponds to the chosen verification item, and loads the corresponding verifier class. (6) It analyzes the source program to obtain an abstract syntax tree, and then by traversing the tree in preorder, invokes the verifier's verify() method to pass detected symbols. (7) The abstract class Verifier, which is a parent class of the verifier, outputs verification results into files. (8) The SyntaxVerifier system reads these verification results from the files, and then displays

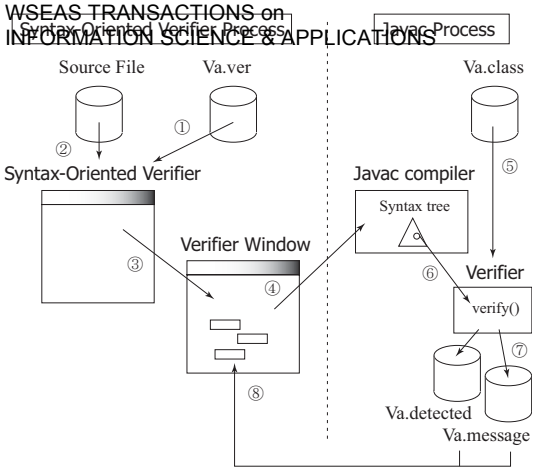


Figure 2: Verification method

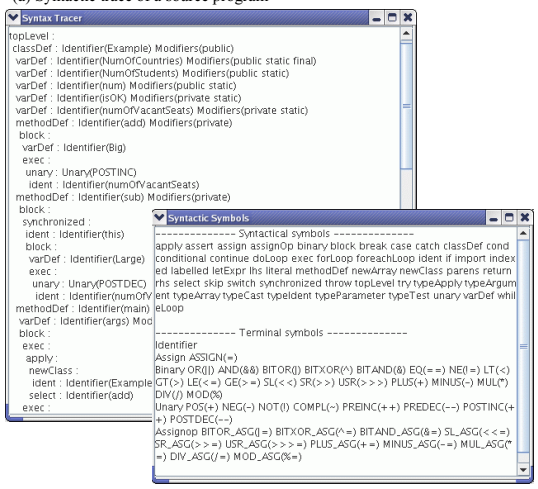


Figure 3: Syntax analysis of a source program

them in the verifier window.

2.2.2 Detection of target objects to be verified

While the Java compiler is traversing the abstract syntax tree, if it detects a target symbols, it will invoke the verifier's verify() method. To traverse the abstract syntax tree, the visitor design pattern [5] is used. In this system, a class named AST is defined to traverse child nodes for each node in the abstract syntax tree. AST class provides visit*Node*(*node*) method for each *node* type. This visit*Node*(*node*) method invokes *child*.accept(this) for a *child* node of the *node*. After an instance ast of AST class is created, when *node*.accept(ast) is invoked for a *node* in the abstract syntax tree, ast.visit*Node*(*node*) will be invoked. Therefore, when topLevel.accept(ast) is invoked for the top level node of the abstract syntax tree, all nodes of the tree can be traversed in turn.

topLevel.accept(ast)—
>ast.visitTopLevel(topLevel)—>....—
>*child*.accept(ast)—>ast.visit*Child*(*child*)—>....

Figure 3 (a) shows an example of traced nodes, which are obtained when a program is analyzed. Figure 3 (b) shows the lists of non-terminal symbols and terminal symbols contained in the abstract syntax tree.

To detect target symbols that satisfy a verification condition while traversing an abstract syntax tree, the system uses two stacks, syntaxStack and verifyingStack. For a current node in the abstract syntax tree, syntaxStack records its ancestor nodes. When a sequence of nodes (such as "if cond binary") that matches the verification condition is found in syntaxStack, verifyingStack will records the location of the sequence. When verifyingStack is not empty, if the system detects a target symbol that meets the verification condition, it will invoke the verifier's verify() method. In fact, this invocation of verifiers also makes use of the visitor pattern [5]. If verifyingStack is not empty, whenever the system visits a node in the abstract syntax tree, it will invoke the target symbol classes. These target symbol classes will invoke the verifier's verify() method only when they receive a node that corresponds to their symbol types (See Fig. 13).

2.2.3 Verifier classes

When a target symbol is found, a verifier's verify() method is invoked. The following parameters are passed to the verify() method as shown in Fig. 1 (b).

1. a detected symbol
2. the value of the detected symbol
3. a list of modifiers attached to the symbol
4. the location (line and column numbers) of the symbol in the source program

The verifier class extends abstract class Verifier. A variety of methods Verifier provides can be used in the verifier class for verification. Table 1 shows an example of these methods. In addition, each node of the abstract syntax tree contains a lot of information (such as a name, a value, a line number, and a column number). The verifier class can use the methods these nodes provide to verify the detected symbols in more detail.

Methods	Functions
int numOfNodesInStack()	Return the number of nodes in the stack
String getSymbolName(int n)	Return the name of the n'th node in the stack
String[] getModifiers(int n)	Return the modifiers of the n'th node in the stack
void addProblem()	Mark this node as a problem
void addMessage(String message)	Add a message to be displayed in the Verifier main window
boolean analyze(String verification, String className)	Perform recursive verification by specifying a verification definition and its verifier class
Tree getNode(int n)	Return the n'th node in the stack

Table 1: Example of verification methods

The verifier, which is shown in Fig. 1 (b), checks whether a variable name starts with a lowercase letter. By using addMessage() method, it adds a message that consists of the name of a detected variable, a line number and a column number to the buffer of the Verifier. This message will be shown in the SyntaxVerifier main window when the "Verify" button is clicked in the variable name verifier window (See Fig. 6). Then, if modifier "final" is not attached to the variable and its name does not start with a lowercase letter, the verifier will invoke addProblem() method to inform Verifier that this variable has a problem. These problematic variables will be marked as problems in the source text of the variable name verifier window (See Fig. 6).

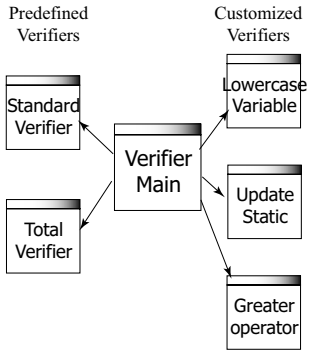


Figure 4: SyntaxVerifier windows

3 SyntaxVerifier

3.1 SyntaxVerifier windows

3.1.1 Window configuration

As shown in Fig. 4, the SyntaxVerifier system displays a verifier window for each verifier to show its verification results. When the source text is modified in one verifier window, this modification will be immediately reflected in the other verifier windows.

This system provides Standard Verifier and Total Verifier as predefined verifiers. In the Standard Verifier, programmers can enter a verification condition and see the verification results right away. When we construct a customized verifier class, we can also use this Standard Verifier to see whether its verification condition is valid or not in advance. Total Verifier shows the verification results of all the verifiers that are registered in the SyntaxVerifier system at a time. The displayed results may be complicated because the verification results of multiple verification items are displayed at a time. However, it is convenient when we check whether the verification of all verification items is completed or not after we finish checking a series of verification items.

3.1.2 Verification result display

A verifier window shows the source text to display verification results on it. As a result of verification, the parts of code that have been judged inappropriate will be marked as problems (See Fig. 6). To implement this function, the abstract class Verifier, which is a parent class of each verifier class, outputs the verification results into files as shown in Fig. 5 and Fig. 2. Figure 5 (a) shows the contents of a *.detected file that records the locations at which problems exist. It consists of a line number, a column number, a judgment, and the value of a detected symbol. The line number and the column number identify the location of a detected symbol. If a detect symbol has no problem, it will be marked light green in the source text of the verifier window. If a detect symbol has a problem, it will be marked light pink. The value of a detected symbol (such as "NumOfStudents") will be used to show it in the "Target" field of the verifier window (See Fig. 6).

Figure 5 (b) shows the contents of a *.message file that records the messages a verifier class added by

```
2 28 - NumOfCountries
4 22 - NG NumOfStudents
5 22 - num
6 27 - isOK
7 23 - numOfVacantSeats
10 12 NG Big
15 16 NG Large
20 37 - args
```

(a) lowercaseVariableName.detected

```
1 34
NumOfCountries (Line 3, Column 29)
2 33
NumOfStudents (Line 5, Column 23)
3 23
num (Line 6, Column 23)
4 24
isOK (Line 7, Column 28)
5 36
numOfVacantSeats (Line 8, Column 24)
6 24
Big (Line 11, Column 13)
7 26
Large (Line 16, Column 17)
8 25
args (Line 21, Column 38)
```

(b) lowercaseVariableName.message

Figure 5: Display method of Verification results

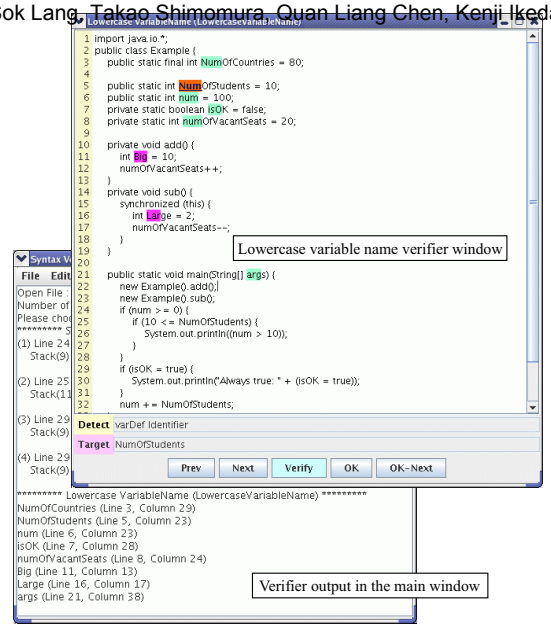


Figure 6: Lowercase variable verifier window

invoking the addMessage() method. This file records index, length and a message itself. Index indicates that this message is related to the index-th detected symbol in the abstract syntax tree. Length indicates the length of the following message. Indexes will be used for merging multiple *.message files when a recursive verification is performed (See Fig. 8).

3.2 Examples of verification

3.2.1 Lowercase variable verifier

An example of verification, where it is verified whether a variable name starts with a lowercase letter, is described below. We first create a verification-item definition file “LowercaseVariableName.ver” shown in Fig. 1 (a). The verification-item name is “Lowercase Variable Name”. This name will be registered as a menu item of the “Verifiers” menu in the SyntaxVerifier main window. The verifier class name is “LowercaseVariableName”. As a verification condition, “verDef Identifier” is specified. This makes the system detect variable names, which are identifiers in the variable definitions of the program.

Next, we create LowercaseVariableName verifier class. Figure 1 (b) shows the source program of this class. While the abstract syntax tree is being traversed, every time a variable name is found in a variable definition, the verify() method of the verifier class will be invoked. If modifier “final” is not attached to the detected variable and its name does not start with a lowercase letter, this verifier class will invoke the addProblem() method to inform Verifier that this variable has a problem.

Figure 6 shows a verifier window for this variable name verification. Variables that start with uppercase letters are marked light pink while variables that start with lowercase letters are marked light green. When the “Next” button is clicked, only the variables that start with uppercase letters can be traversed. When the “Verify” button is clicked, some messages will be shown in the SyntaxVerifier main window, which have been added by the addMessage() method invocation of the verifier class.

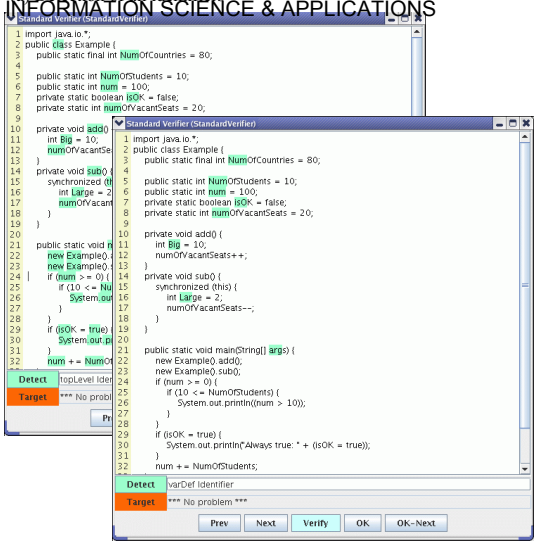
3.2.2 Standard verifier

Figure 7 shows some examples of verification performed by the Standard Verifier, which is a system-predefined verifier. In Fig. 7 (a), “topLevel Identifier” is entered as a verification condition. When the “Detect” button is clicked, all identifiers of the program will be detected. In Fig. 7 (b), “varDef Identifier” is entered as a verification condition. Only the identifiers included in variable definitions are detected.

4 Recursive verification

4.1 A flow of recursive verification

Let’s consider a little more complicated verification, where the abstract syntax tree needs to be traversed twice. For private static variables, we here verify whether their updates will be made synchronously



(a) Detection of all identifiers

(b) Detection of identifiers in variable definitions

Figure 7: Example of Standard verifiers

when they are updated inside instance methods. (1) First, we need to detect all private static variables in a program. (2) Then, for each of these variables, if it is updated inside an instance method, we need to verify whether the update is made by a synchronized method or inside a synchronized statement.

We can detect those variables with a verification condition "varDef Identifier with Modifiers(private static)". To find where these variables are updated in the program and check whether they are updated synchronously, a verifier class might be able to traverse the abstract syntax tree by itself. However, this way of verification will make the size of the verifier class larger, and make its processes more complicated. Therefore, this system has made it possible to dynamically perform the second verification inside the first verifier class. The first verifier class can perform a recursive verification by invoking the analyze() method with a new verification condition and a new verifier class (See Table 1).

Figure 8 illustrates a flow of the recursive verification. This figure follows Fig. 2. (7) When a verify() method invokes an analyze() method in a verifier class, the Java compiler is started. (8) It reads the verification-item definition file for a new verification item, and then loads a new verifier class. (9) It analyzes the source program to obtain an abstract syntax tree, and then by traversing the tree in preorder, invokes the verifier's verify() method to pass detected symbols. (10) The abstract class Verifier, which is a parent class of the verifier, outputs verification results into files.

In this recursive verification, a verifier class at each level outputs its verification results into files. Verifier class Va at level 0 invokes a verifier class at level 1 each time a target symbol is detected, and finally, outputs its verification results, which are a Va.detected file and a Va.message file. If the verifier class at level 1 is invoked when the index-th symbol is detected in the verifier class at level 0, it will output a Va.index.detected file and a Va.index.message file.

The SyntaxVerifier system merges all of these files (Va.*.detected and Va.*.message) to show the verification results in the verifier window at level 0. Va.*.detected files will be sorted in the order of line and column numbers. Va.*.message files will be merged so that the messages will construct a hierarchical structure with respect to their levels and indexes. For example, let the contents of Va.message, Va.1.message and Va.2.message files be as follows:

Va.message
1 length1
message1
2 length2
message2
Va.1.message
3 length3
message1.3
Va.2.message
4 length4
message2.4

If these Va.*.message files are merged, the result will be as follows:

message1
<TAB>message1.3
message2
<TAB>message2.4

4.2 An example of recursive verification

Figure 9 shows a verification item for detecting private static variables. Private static variables can be detected with a verification condition "varDef Identifier with Modifiers(private static)" (See Fig. 9 (a)). For each detected private static variable *verifiedText*, verifier class UpdatePrivateStatic specifies a new verification condition "methodDef Identifier(*verifiedText*)" and a new verifier class SynchronizedUpdate, and then invokes analyze() method to ask a new verifier to perform a recursive verification (See Fig. 9 (b)). If detected private static variables *verifiedText* are referred

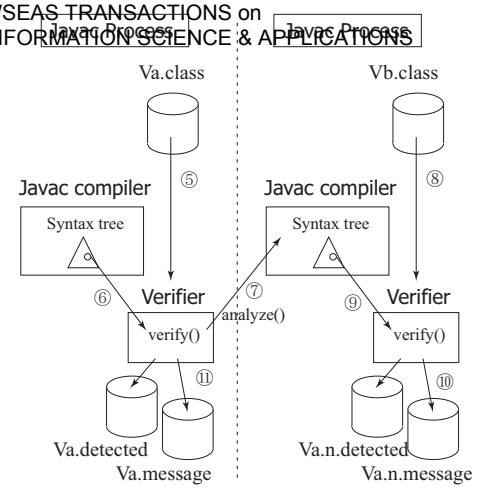


Figure 8: Recursive verification method

```
Update private static variables
UpdatePrivateStatic
varDef Identifier with Modifiers(private static)
```

(a) Definition for UpdatePrivateStatic

```
package ver;
import javacProcess.ver.*;
public class UpdatePrivateStatic extends Verifier {
    public void verify(String terminalSymbol,
    String verifiedText, String[] modifiers,
    int lineNo, int columnNo) throws Exception {
        String verification =
        "methodDef Identifier(" + verifiedText + ")";
        String fullClassName = "ver.SynchronizedUpdate";
        boolean isProblem =
        analyze(verification, fullClassName);
        if (isProblem) {
            addProblem();
        }
    }
}
```

(b) UpdatePrivateStatic class

Figure 9: Verifier class definition 1 for recursive verification

to in method definitions, the new verifier class SynchronizedUpdate will further verify them. If a variable is judged to be inappropriate in the recursive verification, verifier class UpdatePrivateStatic will finally judge it to be wrong.

Figure 10 shows the verifier class SynchronizedUpdate that verifies whether a detected variable will be updated by a synchronized method or inside a synchronized statement if it is updated inside an instance method. This verifier checks nodes contained in the syntaxStack in turn to judge whether the detected variable is updated, whether it is included in a synchronized statement, whether it is included in a synchronized method definition, or whether it is included in a static method definition.

Figure 11 shows an example of this recursive verification. Variables isOK and numOfVacantSeats have

```
package ver;
import javacProcess.ver.*;
public class SynchronizedUpdate extends Verifier {
    public void verify(String terminalSymbol,
    String verifiedText, String[] modifiers,
    int lineNo, int columnNo) throws Exception {
        boolean isUpdated = false;
        boolean synchronizedStatement = false;
        boolean synchronizedMethod = false;
        boolean instanceMethod = true;
        int numOfNodesInStack = numOfNodesInStack();
        for (int level = 0; level < numOfNodesInStack;
        level++) {
String symbolName = getSymbolName(level);
if (symbolName.equals("unary") ||
symbolName.equals("lhs")) {
    isUpdated = true;
} else if (symbolName.equals("synchronized")) {
    synchronizedStatement = true;
} else if (symbolName.equals("methodDef")) {
    String[] methodDefModifiers = getModifiers(level);
    for (int i = 0; i < methodDefModifiers.length; i++) {
        if (methodDefModifiers[i].equals("synchronized")) {
            synchronizedMethod = true;
        } else if (methodDefModifiers[i].equals("static")) {
            instanceMethod = false;
        }
    }
}
        }
        if (isUpdated && instanceMethod &&
        !(synchronizedStatement || synchronizedMethod)) {
            addProblem();
        }
    }
}
```

Figure 10: Verifier class definition 2 for recursive verification

been detected as private static variables. Although variable isOK is detected as a private static variable, it is marked light green because it has no problem. Variable numOfVacantSeats is marked light pink because it is against this verification item. The marking on lines 7 and 8 in variable definitions is a result of the verification made by UpdatePrivateStatic verifier. On the other hand, the marking on lines 12, 17, 29 and 30 in method definitions is a result of the verification made by SynchronizedUpdateverifier.

5 Extension of verified syntax

5.1 Extension of non-terminal symbols

An abstract syntax tree is created by the Java compiler. Therefore, some non-terminal symbols might not be explicitly included in the tree. The SyntaxVerifier system has enabled any type of non-terminal symbols to be included in the tree if they are needed for verification. Figure 12 (a) illustrates a part of syntax tree created by the Java compiler. This syntax tree does not contain a node that represents the conditional part of an if statement. Therefore, we cannot specify “if cond Binary” as a verification condition to verify only the binary operators included in the conditional part of if statement. Figure 12 (c) illustrates a syntax tree to which some non-terminal symbols have been added for verification.

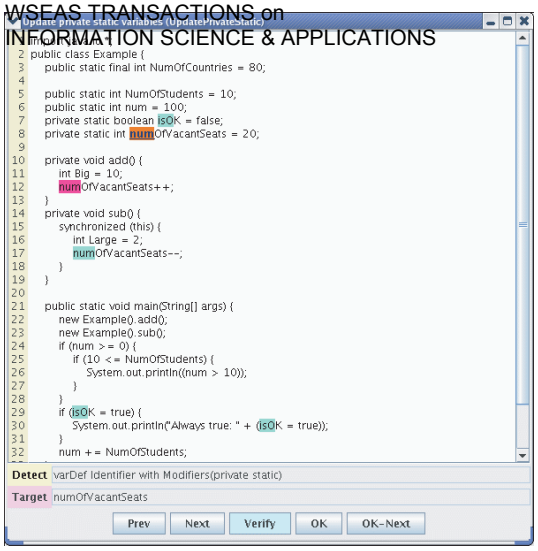


Figure 11: Example of recursive verification

In the system, a process of traversing child nodes for each node is written in a class named AST. Because the visitor pattern [5] is used for traversing an abstract syntax tree, if child.accept(this) is invoked for a child node of a node, all nodes in the sub-tree with the child node as a root can be automatically traversed (See Section 2.2.2). Figure 12 (b) shows a way of adding a new non-terminal symbol cond, which represents the conditional part of if statement. Some code needs to be inserted in the visitIf() method that processes if node. Immediately before traversing the conditional part of if statement, a new cond node is created, and startNode() method is invoked to push the created cond node to the syntaxStack. Then, after traversing the conditional part of if statement, stopNode() method is invoked to pop up the cond node from the stack.

5.2 Extension of terminal symbols

The SyntaxVerifier system deals with not only non-terminal symbols but also terminal symbols as a target symbol for verification. In the syntax tree created by the Java compiler, however, some terminal symbols might be attached to non-terminal symbol nodes as auxiliary information. Therefore, some terminal symbols do not appear as a node of the syntax tree. The SyntaxVerifier system has enabled any type of terminal symbols to be easily detected even if they are not included in the syntax tree if those terminal symbols are needed for verification.

Figure 13 illustrates a way of detecting a new terminal symbol Binary, for example, which can be used to specify such a verification condition as “if cond Binary(GE)”. To detect a new terminal symbol, we have only to define a new terminal symbol class that corresponds to it. For example, for binary operators “Binary” such as >, >=, <, <=, &&, ||, +, - , *, / , %, we have only to define a “BinaryTS” class (See Fig. 13 (a)). While an abstract syntax tree is being traversed, if a sequence of nodes (such as “if cond”) that matches the verification condition is found in syntaxStack, visit*Node*(*node*) method of BinaryTS class, which corresponds to the target symbol (Binary(GE)) specified in the verification condition, will be invoked for each *node* since then (See Fig. 13 (b)). When an if node is visited, visitIf(ifNode) method of BinaryTS class will be invoked. However, visitIf(ifNode) method is not defined in BinaryTS class. As a result, visitIf(ifNode) method of the abstract class TerminalSymbol, which is a parent of BinaryTS class, will be invoked, and nothing will be done. When a binary node is visited, visitBinary(binaryNode) method of BinaryTS class will be invoked. If the value of the Binary terminal symbol equals GE, this method will further invoke the verify() method of the verifier (See Fig. 13 (c)).

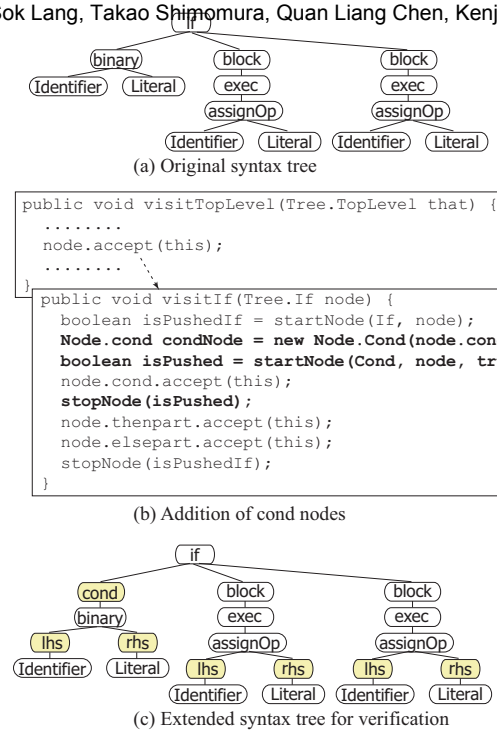


Figure 12: Extension of non-terminal symbols

6 Observation

Table 2 shows some verification conditions and their verifiers’ program sizes. Program sizes indicate the

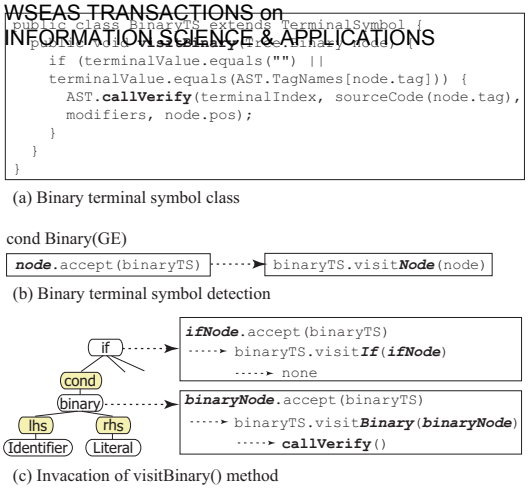


Figure 13: Extension of terminal symbols

number of lines of instructions which are written in the body of a verify() method. The fourth verification item has two verifier classes to perform a recursive verification. We were able to easily define these verification classes in a small number of lines of code. The reason we do not need to write much code for a verifier class might be because we can specify a syntactic context and some modifiers as a verification condition. In addition, even for more complicated verification items, we can divide those verification items into simpler ones, and easily write verifier classes by making use of a recursive verification.

Code Conventions [9] proposes the standard code conventions for Java Programming. The first and third verification items in Table 2 are examples of Code Conventions. Some systems that verify the quality of programs include IntelliJ IDEA [16], CheckStyle [15] and Eclipse [10]. IntelliJ IDEA is an intelligent Java IDE intensely focused on developer productivity that provides a robust combination of enhanced development tools. By using regular expressions, it searches some pieces of target code for a source program, and corrects the matched text by replacing it with a replacement string. It performs about the same verification as the pattern-oriented verifier [14]. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It does not have a verification-item definition file, and specifies a target node inside a verifier class. A lot of methods are provided for verification, and verifier classes can be easily defined. However, it cannot specify a syntatic context on which a target node is detected in a syntax tree. It does not have the concept of recursive verification. Because verification results are output as logs in Checkstyle, they are not marked on the source text.

7 Conclusion

This paper has proposed the syntax-oriented verifier SyntaxVerifier that makes it easy to verify the quality of programs. With the SyntaxVerifier system, we can define a target object to be verified in an abstract syntax tree that appears in a specific syntactic context, and easily add a new verifier class that verifies the detected target objects in detail. In addition, this paper has proposed a recursive verification by which a verifier class can dynamically repeat its more detailed verification. This system starts a Java compiler each time it performs a new verification. When the source text is modified in one verifier window, this modification can be also reflected in the other verifier windows. However, this decreases the performance of the system. In the future, instead of repeatedly starting the Java compiler, we are going to enhance the system so that it will output an abstract syntax tree to a file and share this syntax tree with multiple verifiers. We will also incorporate this syntax-oriented verifier into another pattern-oriented verifier.

References:

[1] *The Apache Software Foundation : Struts*. http://jakarta.apache.org/struts/, 2004.

[2] A. S. Christensen, A. Moller, and M. I. Schwartzbach. Extending java for high-level web service construction. *ACM TOPLAS*, Vol. 25, No. 6, pp. 814–875, 2003.

[3] Takao Shimomura. Visual design and programming for web applications. *Journal of Visual Languages and Computing*, Vol. 16, No. 3, pp. 213–230, 6 2005.

[4] A. Leff and J.T. Rayfield. Web-application development using the model/view/controller design pattern. In *Fifth International Enterprise Distributed Object Computing Conference*, pp. 118–127, 9 2001.

[5] Steven John Metsker and William C. Wake. *Design Patterns in Java*. Addison-Wesley, 4 2006.

[6] Takao Shimomura, Kenji Ikeda, Quan Liang Chen, Nhor Sok Lang, and Muneo Takahashi. Rich component generation for web applications using custom tags. In *Proc. of WSEAS International Conference on Computer Engineering and Applications*, pp. 390–395, 1 2007.

[7] Takao Shimomura, Kenji Ikeda, Quan Liang Chen, Nhor Sok Lang, and Takahashi Muneo.

No.	Verification item	Definition	LOC	Description
1	Lowercase variable	varDef Identifier	7	Variable names should start with a lowercase letter.
2	Public static variable	varDef Identifier Modifiers(public static)	4	Public static variables should be final.
3	Greater operators	binary Binary(GT) Binary(GE)	1	Greater than and equal operators should be replaced with less than and equal operators.
4	Private static variable update	varDef Identifier Modifiers(private static)	5	Recursively verify the syntax tree with detected private static variables.
		methodDef Identifier(*detectedVariable*)	20	Update of detected variables should be synchronized in instance methods.

Table 2: Lines of code of several verifier classes

Visual programming for web applications that use html frame facilities. In *Proc. of WSEAS International Conference on Computer Engineering and Applications*, pp. 384–389, 1 2007.

[8] Takao Shimomura, Quan Liang Chen, Nhor Sok Lang, and Kenji Ikeda. Integrated laboratory network management system. In *Proc. of the 7th WSEAS International Conference on Applied Informatics and Communications*, pp. 188–193, 8 2007.

[9] *Sun Microsystems, Inc. : Code Conventions*. http://java.sun.com/docs/codeconv/, 2006.

[10] *The Eclipse Foundation : Eclipse*. http://www.eclipse.org/, 2006.

[11] D. Castelluccia, M. Mongiello, M. Ruta, and R. Totaro. Waver: A model checking-based tool to verify web application design. *Electronic Notes in Theoretical Computer Science*, Vol. 157, No. 1, pp. 61–76, 5 2006.

[12] E. Di Sciascio, F.M. Donini, M. Mongiello, and G. Piscitelli. Anweb: a system for automatic support to web application verification. In *Proc. of SEKE*, pp. 609–616, 7 2002.

[13] Marco Pistore, Marco Roveri, and Paolo Busetta. Requirements-driven verification of web services. *Electronic Notes in Theoretical Computer Science*, Vol. 105, No. 10, pp. 95–108, 12 2004.

[14] Takao Shimomura, Kenji Ikeda, Chen Liang Quan, Lang Sok Nhor, and Takahashi Muneo. Customizable verifiers for web applications and their implementation. In *Proc. of WSEAS International Conference on Computer Engineering and Applications*, pp. 396–401, 1 2007.

[15] *Oliver Burn : Checkstyle*. http://checkstyle.sourceforge.net/, 2007.

[16] *JetBrains : IntelliJ IDEA*. http://www.jetbrains.com/idea/, 2007.