Testing syntax and semantic coverage of Java language compilers

A.S. Boujarwah*, K. Saleh, J. Al-Dallal

Kuwait University, Department of Electrical and Computer Engineering, PO Box 5969, Safat 13060, Kuwait

Received 2 January 1998; received in revised form 11 May 1998; accepted 15 June 1998

Abstract

Software testing is a critical and important stage of the application software development life-cycle. Testing is a verification activity that affects the overall software quality. The verification of critical and dependable computer software such as real-time safety-critical software systems consumes about 50% of the project time. In this work, we consider testing compilers. Since a compiler is a highly usable software, an increased emphasis on reliability requires a careful testing before releasing the compiler. In compiler testing, the compiler implementation is verified to conform to the specified language syntax and semantic available in the standard language documentation. In this work, an algorithm is designed and a tool is developed to automate the generation of test cases to check the language syntax. In addition, we propose a heuristic approach to cover language semantics. Since Java is a relatively new programming language, we concentrate on testing the adherence of new Java compilers to the language syntax and semantics. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Compiler; Test case generation; Software testing; Java

1. Introduction

Testing is an important and critical activity in the software development process aiming at finding errors to increase the level of confidence in the developed software. In real-life application software development life-cycle, the testing stage is known to be time-consuming and costly since highly reliable products are targeted. The basic element in testing is a test case packaged with other test cases in a test suite. Each test case must contain a description of the input data to the program and the description of the correct output of the program corresponding to the set of input data. Test cases must cover the invalid and unexpected, as well as the valid and expected situations. An elaborate test case coverage increases the designers, testers and users confidence in the tested software product. However, ensuring a complete coverage of a system under test is impossible. Generating an effective set of tests for systems of realistic sizes and complexities can be a substantial task requiring the analysis of thousands of combinations of cases. Thus, automation of the test plan generation process is the most practical way to generate an effective set of tests for large-scale systems.

A compiler is a large and complex program that accepts any string of symbols from the input, i.e. the source code, and produces either compiler error messages or an object code corresponding to the valid source code. The compiler should be carefully verified before its release since it is distributed to many users, and is essential for the creation of other programs. Due to the increasing complexity of software applications, the need for more general and expressive high level languages is becoming greater. Thus, testing compilers for high level programming languages is becoming an increasingly complex task. By testing compilers we ensure that the syntax and semantics of the language are supported adequately by the validated compilers.

Java, developed by Sun Microsystems, is a general purpose object-oriented programming language [1]. Using Java, application developers are able to run their written programs anywhere on the Internet. Besides supporting animated and interactive Web pages, Java can support new kinds of applications, such as autonomous intelligent agents and dynamically extensible borrowers. Although Java is a relatively new programming language, many of its features are borrowed from a variety of older languages, such as Smalltalk and C++, to achieve the language designer’s goals. Moreover, most of the Java language syntax is borrowed from C/C++.

In Ref. [2], a survey and assessment of compiler test case generation methods is provided. In this survey, it was found that no semantic coverage of language compilers was addressed. The aim of our work is to generate test cases to test Java compilers for adherence to both syntax and
semantics as described by the official language specification documentation published by Sun Microsystems. These test cases are black-box-based tests since they are only based on the language specification and not any compiler source code [5]. To achieve our goal, we have extended Purdom’s algorithm [3] to automate the generation of statements that cover the whole language syntax. Moreover, we implemented an extended version and applied it to the Java syntax. Also, in order to test the implementation of the language semantics, we propose a heuristic approach. Finally, we applied the proposed approach to generate test cases for some basic Java semantics.

This paper is organized as follows. In Section 2, we discuss covering syntax in language compilers. Section 3 discusses the semantic coverage in language compilers. Section 4 discusses the test coverage for Java semantics. In Section 5, we describe the implemented tool that automates the generation of syntax and semantic test cases for Java compilers. Finally, in Section 6, we conclude the paper.

2. Syntax coverage in language compilers

There are several types of grammars that can be used to define the syntactic structure of a language specification. A context-free grammar (CFG) is a well-known grammar type allowing the description of a programming language using a number of production rules. The left-hand side of each production holds an abstract symbol called a non-terminal. The right-hand side contains a sequence of one or more terminal and non-terminal symbols. The terminal symbols are drawn from a specified grammar alphabet. Given a context-free grammar of a language, statements of that language can be derived by repeatedly replacing any non-terminal, in a production sequence, with a right-hand side of a production for which the non-terminal is the left-hand side.

In 1972, Purdom introduced a fast algorithm to automatically generate a small set of short sentences that obey the syntax described by a context-free grammar. The algorithm guarantees the coverage of each production in the grammar at least once.

In a context-free grammar, a syntax rule may have more than one subrule (production) used to describe the syntax rule. To use Purdom’s algorithm, the name of the first subrule, which is called the axiom, should be the same as the name of the rule. Each subrule may have more than one alternative. Finally, each alternative has some contents. Some of these contents are terminal and others are non-terminal.

Purdom’s algorithm defines the length of a string derived from a symbol to be the number of characters (terminal symbols) in the string plus the minimum number of steps needed in this derivation. Using Purdom’s algorithm, the sentences are generated starting from the axiom and by rewriting at each step the leftmost non-terminal symbol in the yield of the derivation, as described in the following algorithm.

PURDOM’s algorithm:
1. Insert the axiom S of the rule in a pushdown stack whose content is the yield.
2. repeat
   if the top of the stack is a terminal symbol then pop it and write it.
   else CHOOSE a subrule A → α rewriting the non-terminal symbol A and substitute α for A at the top of the stack
3. until stack is empty.

The choice of a particular subrule A → α is made using the following strategy.

CHOOSE algorithm:
if one or more subrules A → α exist which has not yet been used then choose the one that yields the longest string derivation from A.
else if a derivation A ⇒ α ⇒ γ1Bγ2 exists such that B is non-terminal symbol not in the stack, and a subrule B → β exists which has not yet been used then use the subrule A → α with the shortest right-hand side
else use the subrule which is the first step of the derivation having as its yield the shortest string derivation from A.

Purdom’s algorithm generates statements that cover productions of a single syntax rule. However, in the case of testing a language syntax, we have to cover all the syntax rules. Thus, in our work, we use Purdom’s algorithm for each syntax rule in the language. In addition, we extended the algorithm for the case of having a non-terminal symbol, defined in another syntax rule, in the right-hand side of a syntax production rule. In this case, we apply an algorithm to generate the minimal set of symbols needed by that non-terminal symbol, as described in the following.

EXTENDED PURDOM’s algorithm:
repeat
1. Empty the stack
2. Insert the axiom S of the rule in a pushdown stack whose content is the yield.
3. repeat
   if the top of the stack is a terminal symbol then pop it and write it.
   else if the top of the stack is a non-terminal symbol defined in the current rule
   then CHOOSE a subrule A → α rewriting the non-terminal symbol A and substitute α for A at the top of the stack
   else if the top of the stack is a non-terminal symbol defined in another rule then Generate_Minimum_Statement for the non-terminal symbol
else Error
until stack is empty.

until all syntax rules are covered

The generation of the minimum statement is made using the following algorithm.

Generate_Minimum_Statement algorithm:

1. Insert the axiom $S$ of the rule in a pushdown stack whose content is the yield.
2. repeat
   if the top of the stack is a terminal symbol then pop it and write it.
   else if the top of the stack is a non-terminal symbol defined in the current rule
     then choose a subrule $A \rightarrow \alpha$ that yields the shortest string derivation from $A$ rewriting the non-terminal symbol $A$ and substitute $\alpha$ for $A$ at the top of the stack
     else if the top of the stack is a non-terminal symbol defined in another rule then Generate_Minimum_Statement for the non-terminal symbol
   else Error
until stack is empty.

The generation of statements containing some language syntax errors can be done by eliminating some rule symbols, randomly, until all possible language syntax errors (messages) are covered. The elimination of symbols can be done to symbols either in the stack or in the rule itself, which leads to incompletely generated statements.

Since our aim is to generate complete test cases, we have to include some declarations to the generated statements containing headers and some type declarations. These declarations depend on the language for which the compiler under test is written. Thus, these declarations can be added at the right locations, according to the language, after the statements are generated.

3. Semantic coverage in language compilers

As indicated earlier, the syntax of any programming language can be presented in a formal way using grammars and production rules. The semantics of language constructs can be described using operational, denotational or axiomatic semantics. However, the available language reference manuals, which describe semantics of programming languages, often use English instead of using one of the three formal approaches. Thus, we are not able to find a general algorithm that fits our aim. Generally, we can note that the generation of test cases for some semantical information can be automated, while others have to be done manually. Even for the case in which the testing can be automated, it is very difficult (or impossible) to have a specific algorithm which is suitable for all semantics in a language. As a result, it is difficult (or impossible) to write a general test cases generation algorithm that suits all language semantics. However, the following heuristic approach seems to be useful for producing programs (test cases) that cover semantic rules.

Heuristic:

1. Highlight the semantic information from the official language definition manual.
2. Consider normal and abnormal semantic cases that are related to the highlighted semantics.
3. Decide whether the test cases can be automated or dealt with manually.
4. If test cases have to be generated manually then write them, otherwise
   1. List semantic information in a file to facilitate dealing with them.
   2. If needed, design the appropriate data structures needed to store the semantic information.
   3. Design and implement procedures that read the semantic information and automatically generate semantic test cases.

In Step 4 of the heuristic approach, all semantic information (highlighted in Step 1) in both normal and abnormal cases (listed in Step 2) have to be covered by the generated test cases. Moreover, the designer of the procedures required in Step 4.3 has to consider the sequence of needed variables, values, and used alternatives in order to generate the required statements correctly. These sequences restrict the generated statements to the point we aim to test. To generate the needed statements, we have to use a procedure that generates the minimum restricted statements since we are not testing the compiler for its adherence to the language syntax. Thus, we have to extend the Generate_Minimum_Statement algorithm to accept some restrictions for non-terminal symbols inserted in the stack. The suggested extension is outlined in the following.

Extended Generate_Minimum_Statement algorithm

1. Insert the axiom $S$ of the rule in a pushdown stack whose content is the yield.
2. repeat
   if the top of the stack is a terminal symbol then pop and write it.
   else if the top of the stack is a non-terminal symbol required to be restricted for a specified subrule $A \rightarrow \alpha$ or terminal symbol $B$ then rewrite the non-terminal symbol $A$ and substitute $\alpha$ or $B$ for $A$ at the top of the stack
   else if the top of the stack is a non-terminal symbol defined in the current rule then choose a subrule $A \rightarrow \alpha$ that yields the shortest string derivation from $A$ rewriting the non-terminal symbol $A$ and substitute $\alpha$ for $A$ at the top of the stack

The suggested extension is outlined in the following.
else if the top of the stack is a non-terminal symbol defined in another rule then Generate_Minimum_Statement for the non-terminal symbol

else Error

until stack is empty.

It can be noted from the heuristic approach that many steps have to be done manually and any written code has to be specific for testing the specific semantics of a single rule and cannot be used to test other rules. Actually, the only automation seen above is in generating test cases that meet the requirements (last step). However, in some rules there are variables to make the automation of test cases useful. In other words, the semantic information (in Step 1) does not have any variable that can be played with to generate various and meaningful test cases.

4. Test coverage for Java semantics

Since a language may have a large number of semantic rules, we did not put our effort in testing all Java semantics; instead, we chose only some of the basic semantics. Moreover, we tried to clarify our heuristic approach by selecting some semantics that can be tested automatically and others that have to be done manually. Although exception handling, multi-threading, and other advanced semantics concepts are important features of Java, we are planning to consider them in a future research.

In the following, we discuss the testing of some basic Java semantics extracted from two Java reference books by Gosling et al. [1] and Grand [4].

4.1. Looping

Testing the semantic implementation of looping means testing whether loops are looping as they are expected. A complete looping generated test cases should cover three situations: normal looping, abnormal looping (infinite looping), and nested loops. Moreover, three kinds of Java loop semantics must be tested: for, while, and do loops. For each of normal and nested loops, there are three variables. The first and second variables specify the initializaton value of the loop required to be generated and the number of times the generated loop should be executed, respectively. The third variable determines the number of required looping test cases, in the case of normal looping, and the number of required nested loops, in the case of nested looping. These three variables make the automation of looping test cases meaningful. The automation can be done by using these variables as restrictions on the generated loop statements.

The test cases that cover the above Java looping semantic rules are listed in Appendix B.

4.2. Switching

Testing the semantic implementation of switching means testing whether the switch statement transfers control to a labeled statement in its block depending on the evaluated value of an expression. The following are the extracted semantic rules that are related to a Java switch-case statement.

1. The type of `switch` expression and the case constant expression must be byte, char, short, or int, or a compile-time error will result.
2. The value of the `switch` expression and the compared case constant expression are converted to int when they are evaluated.
3. The body of the `switch` statement must be a block that begins with a statement labeled with one or more `case` or `default` labels.
4. If the `case` label that has the same value as the expression is found, all statements after the matching `case` label in the `switch` block, if any, are executed in sequence, unless the flow of control is altered.
5. If no `case` label is found with the same value as the expression but there is a `default` label, all statements after the `default` label in the `switch` block, if any, are executed in sequence, unless the flow of control is altered.
6. If no `case` matches and there is no `default` label, the statement after the `switch` statement is the next statement to be executed.
7. A compile-time error occurs if two or more case constant expressions associated with the same `switch` statement have the same value.
8. A compile-time error occurs if two or more `default` labels are associated with the same `switch` statement.

Since the generation of test cases that test Java compilers for adherence to switching requires meaningful variables, an automatic generation of test cases is more suitable.

4.3. Overloading

Testing a compiler for adherence to the semantic implementation of the overloading concept implies testing whether the compiler can distinguish methods of the same class and name, but with some differences as specified in the Java specification. The differences can be in the signatures (parameter names, types and their order), modifiers, throws, or return types of the overloaded methods. As indicated in the Java language specification, when a method is invoked, the number of actual arguments and the compile-time types of the arguments are used, at compile time, to determine the signature of the method that will be invoked. If more than one method is compatible with the given arguments, the method that matches the given parameters is selected. If no match is found, the compiler issues an error message. Similarly, if the methods have different throw clauses,
modifiers, or return types, the compiler will also issue an error message.

Thus, in order to test the semantics of the overloading concept in Java, we have to cover the following differences in the overloaded method:

1. number of actual arguments;
2. compile-time types of the used arguments;
3. order of the actual arguments;
4. close compile-time types of the used arguments (float and double);
5. return types;
6. throw clauses; and
7. modifiers.

Due to having limited cases, the loss of the needed variables that are useful in automating the generation of test cases, and the inefficient lengthy code required to be written, we prefer to generate the test cases that test Java compilers for adherence to overloading semantics manually.

The test cases that cover the Java overloading semantic rules are listed in Appendix B.

4.4. Overriding

Testing the semantics of the overriding concept implies testing whether the compiler can distinguish methods of a subclass from methods of its superclass when the methods have the same names, but with some differences, as specified in the Java specification. The differences can be in the signatures, modifiers, throws, or return types of the overridden methods. As indicated in the Java specification, a method inherited from a superclass is said to be overridden if a method in the inheriting class has the same name, number of parameters, and types of parameters as the inherited method. If the overridden method returns void, the overridden method must also return void. Otherwise, the return type of the overriding method must be the same as the type of the overridden method. An overriding method can be more accessible than the overridden method, but it cannot be less accessible. If a method overrides another method, the overriding method cannot throw an exception that the overridden method does not throw. Also, a compile-time error occurs if an instance method overrides a static method. Finally, an overridden method can be accessed by using a method invocation expression that contains the keyword super.

Thus, in order to test the semantics of the overriding concept in Java, we have to cover the following differences in the overridden method:

1. number and order of actual arguments;
2. compile-time types of the used arguments;
3. return types;
4. throw clauses;
5. static and non-static; and
6. modifiers.

If the first two items were the same, the subclass and superclass methods are said to be the overriding and the overridden methods, respectively. Otherwise, they will not be considered as overridden methods and no compile-time error will be issued. However, the other four items, in some cases, may cause the compiler to issue error messages.

Since considering the semantics of Java overriding requires a complex work, and having no useful variables in automating the generation of test cases, we can say that the generation of test cases that test the semantics of overriding in Java compilers can be done manually.

4.5. Interfaces

Interfaces provide templates of behavior that other classes are expected to implement. The following are the extracted semantic rules that are related to Java interfaces.

1. An interface cannot provide implementations for its methods.
2. When a class declaration specifies that it implements an interface, the class inherits all the variables and methods declared in that interface.
3. The class declaration must provide implementations for all the methods declared in the interface, unless the class is declared as an abstract class.
4. A class can implement an unlimited number of interfaces.
5. An interface can extend only other super interfaces.
6. Interfaces have no constructors.
7. Interfaces have no static initializers.
8. An interface must contain an initializer that sets the value of the named constant.
9. The accessibility of an interface field is the same as the accessibility of the interface even if it is declared differently.
10. Overriding super interface method is allowed in any of its sub interfaces.
11. An interface field variable hides a super interface field variable if they have the same name.
12. It is an error to declare an interface field variable as transient or volatile.
13. It is an error to declare two interface field variables with the same name.
14. It is an error to declare an interface field variable with the same name of a method in the same interface or one of its super interfaces.
15. It is an error to declare an interface method with static, final, native, or synchronized.
16. It is an error to declare an interface method with static, final, native, or synchronized.
17. It is allowed to have overloading methods in an interface.
18. Interfaces cannot be instantiated.
19. A compile-time error occurs if the identifier naming an
20. A compile-time error occurs if the same modifier appears more than once in an interface declaration.
21. A compile-time error occurs if there is a circularity such that an interface directly or indirectly extends itself.
22. A compile-time error occurs if, in a class that inherits more than one field with the same name from different interfaces or classes, an attempt to refer to either field by its simple name exists.
23. A compile-time error occurs if, in an interface that inherits more than one field with the same name from different interfaces, an attempt to refer to either field by its simple name exists.
24. A compile-time error occurs if the same interface is mentioned two or more times in a single implements clause.

As for covering the overriding semantics, the generation of test cases that test Java compilers for adherence to interface semantics can be done manually, and will cover the test purposes listed above.

4.6. Inheritance

Inheritance is a way of organizing related classes in a class hierarchy so that they can share common code and state information. The following are the extracted semantic rules that are related to inheritance in Java.

1. When a class declaration specifies that it implements an interface, the class inherits all the variables and methods declared in that interface (tested in Section 4).
2. When a class declaration specifies that it extends a class in its package, the class inherits all non-private variables and methods declared in that extended class.
3. A subclass can inherit directly from only one class (i.e. simple inheritance).
4. There is no limit on the depth to which you can carry subclassing.
5. Abstract classes cannot be instantiated.
6. Final classes cannot be subclassed.
7. Overriding rules (tested in Section 4.4).
8. A compile-time error occurs on any attempt to refer to any ambiguously inherited field by its simple name.
9. It is allowed to inherit the same field declaration from an interface by multiple paths.
10. If a class inherits more than one method with the same signature then if one of the methods is static, a compile-time error occurs.
11. If a class inherits more than one method with the same signature then the method inherited from a class is considered to override, and therefore to implement, all other methods on behalf of the class that inherits it (overriding rules are discussed in Section 4.4).
12. A compile-time error occurs when a class inherits more than one method with the same signature from different interfaces, such that they have different return types.

Some of Java inheritance semantics have variables that can be meaningfully manipulated, e.g. depth of carried classing. However, the generation of these cases for the rest of inheritance semantics is either hardly or inefficiently automated. Thus, the generation of test cases that test Java compilers for adherence to inheritance semantics can be semi-automated.

4.7. Packages

Java programs are organized as sets of packages. Each package is a group of classes. The following are the extracted semantic rules that are related to Java packages.

1. If a class is not declared public, it can be referenced by other classes in the same package only.
2. A class or interface definition can refer to class and interface definition in a different package by qualifying the class or interface name with the package name and a period.
3. Import statement can make classes and interface available in another package.
4. A compile-time error occurs if a package contains a type declaration and subpackage of the same name.
5. A compile-time error occurs if in an important statement the named class or interface types does not exist.
6. A compile-time error occurs if two single-type-import declarations in the same compilation unit attempt to import types with the same simple name, unless the two types are the same types, in which case the duplication declaration is ignored.
7. A compile-time error occurs if an import statement imports a subpackage.
8. If an interface is not declared as public, it can be referenced by other classes and interfaces in the same package only.

As for covering overriding and interface semantics, the generation of test cases that test Java compilers for adherence to package semantics can be done manually.

4.8. Modifiers

Methods are prefixes that can be applied in various combinations to the methods and variables within a class and, some, to the class itself. The following are the extracted semantic rules that are related to Java access modifiers. These rules are organizing according to their application to classes and interfaces, methods, and field variables.

4.8.1. Classes and interfaces modifiers

1. If a class is not declared as public, it can be referenced by other classes in the same package only (tested in Section 4.7).
2. Abstract classes cannot be instantiated (tested in Section 4.6).
3. Final classes cannot be subclassed (tested in Section 4.6).
4. If a class contains an abstract method, the class must be also declared abstract.
5. Classes not declared abstract must override any abstract methods they inherit with methods that are not abstract (tested in Section 4.5).
6. Classes that implement an interface and are not declared abstract must contain or inherit methods that are not abstract that have the same name, number of parameters, and corresponding parameter types as the methods declared in the interfaces that the class implements (tested in Section 4.5).
7. If an interface is not declared as public, it can be referenced by other classes and interfaces in the same package only (by default) (tested in Section 4.7).

4.8.2. Methods and field variables modifiers

1. Public methods and field variables are accessible from any class (some cases tested in Section 4.7).
2. Protected methods and field variables are accessible in any class that is part of the same package as the class in which the methods and field variables are declared. They are also accessible to any subclass of the class, in which they are declared, in any package.
3. Private methods and field variables are only accessible in the class in which they are declared.
4. Methods and field variables that are not declared with any of the access modifiers are accessible from any class in the package in which they are declared only.
5. Static methods cannot be overridden (tested in Section 4.4).
6. Final methods cannot be overridden (tested in Section 4.4).
7. If a method is declared with the synchronized modifier, a thread must obtain a lock before it can invoke the method. If the method is not declared static, the thread must obtain a lock associated with the object used to access the method. Otherwise, the thread must obtain a lock associated with the class in which the method is declared (will be tested in multi-threading).
8. Rules of interface method modifiers (tested in Section 4.5).
9. There is exactly one copy of each static variable associated with the class. Thus setting the value of a class variable changes the value of the variable for all objects that are instances of the class or any of its subclasses.
10. There is a distinct copy of each non-static variable associated with every instance of the class. Thus, setting the value of an instance variable in one object does not effect the value of the instance variable in any other object.
11. Any assignment to a final variable other than the one in its declaration is an error.
12. If a class contains a final variable, it is allowed to declare a variable with the same name in a subclass of the class without causing an error.
13. The transient modifier is used to indicate that a field variable is not part of the persistent state of an object.
14. The volatile modifier is used to tell the compiler that the field variable will be modified asynchronously by methods that are running in different threads (will be tested in multi-threading).

As for covering overriding, interface, and package semantics, the generation of test cases that test Java compilers for adherence to modifier semantics can be done manually covering the above listed test purposes.

4.9. Typing

A data type defines the set of values that an expression can produce or a variable can contain. There are two types of data types in Java programs: primitive types and reference types. Primitive types are self-contained values that can be contained in a variable. The reference types contain references to objects of classes, which contain collections of variables and methods that are described by the classes. Arithmetic types, which contain integer types and floating types, and boolean types are primitive types. Class, interface, and array types are reference types.

To test the type compatibility in a Java compiler, we have to test whether castings and assignments are working as specified in the Java reference manual. The following are the extracted semantic rules that are related to Java typing.

1. Table 1 shows whether an assignment from a particular integer type to another integer type can be done directly or requires a type cast.
2. A float value can be assigned to a double variable.
without using a type cast, but assigning a double value to a float variable requires a cast.

3. No other type can be cast to or from boolean.

4. It is not allowed to cast reference types to primitive data types or primitive data types to reference types.

5. It is not allowed for a reference variable declared as a class type to contain a reference to a superclass of the declared class without a casting. However, the reverse is true.

6. It is not allowed for a value to be assigned to a variable declared using an interface type unless the object referenced by the value implements the specified interface.

7. A run-time error occurs if an object is cast to an interface type which the object being cast does not implement.

8. Any reference can be assigned to a variable that is declared of type Object.

9. A reference to an array can be assigned to an array variable if both array types contain elements of the same type or both array types contain object references and the type of reference contained in the elements of the array reference contained in the elements of the array reference can be assigned to the type of reference contained in the elements of the variable.

10. A value of any data type can be cast to its own type.

11. A value of any primitive data type cannot be cast to a reference data type, nor can a reference be cast to any primitive data type.

12. A reference to the class Object can be cast to an array type if the reference actually refers to an array object of the specified type.

13. A reference to an interface type can be cast to a class type if the reference actually refers to an instance of the specified class or any of its subclasses.

14. A reference to an array object can be cast to the class type Object.

15. A reference to an array object can be cast to another array type if the elements of the referenced array and the elements of the specified array type are the same primitive type or the elements of the referenced array are of a type that can be cast to the type of the elements of the specified array type.

Since the generation of test cases that test Java compilers for adherence to typing compatibility requires meaningful variables and type combinations, an automatic generation of test cases is more suitable.

5. Java test cases generation tool

The Java test cases generator tool we developed generates test cases to test Java compilers for adherence to the language syntax and semantics. The tool is written in Turbo Pascal (under DOS).

To automate the generation of test cases that cover testing the whole syntax of a compiler, we follow the following steps:

1. List all programming language syntax rules represented by the context-free grammar in a file.

2. Design and write a subroutine that reads the context-free grammar stored in the input file and fills it in a data structure.

3. Compute the length of each alternative in each syntax subrule of each syntax rule, according to Purdom’s algorithm.

4. Use the implemented Extended Purdom’s to generate the minimal set of sentences covering all alternatives in all context-free grammar rules.

5. Write a template for each test case considering the compiler under test and the language rules.

In our work, we developed a software tool that applies the above steps to generate test cases for Java compilers. The input for this tool is, as indicated earlier, a file containing all Java syntax rules presented by the context-free grammar. The tool stores the Java syntax rules in linked lists. There are four types of nodes used to represent the syntax rules. These types are: syntax rule, sub-syntax rule, alternatives, and content nodes. The attributes of these nodes and the way of their linkage are shown in Fig. 1.

In the tool, we implemented two algorithms: one that computes the length of each alternative in each syntax subrule in each syntax rule, and one that generates the minimal set of sentences covering all alternatives in all context-free grammar rules (extended Purdom’s algorithm). The Java templates containing the headers and the required type declarations can be either written within the code to generate complete test cases, which is our choice, or written...
manually, after executing the tool, for each generated statement.

The output of the syntax part of the tool, which can be stored in a file, contains complete test cases that are ready to be executed by the Java compiler under test.

The tool is powerful enough to be used to generate test cases to test the adherence of a compiler for the syntax of any language, by only changing the programming language templates.

In order to test the compiler for syntax errors, the tool automates the generation of statements with syntax errors by eliminating some content nodes for some stored rules. This elimination forces the generation of syntactically incorrect statements.

As can be noted from the heuristic approach listed in Section 3, two input files are required to generate test cases for a semantic rule under testing: the file that contains the CFG of the language, which holds the syntax of the generated rule, and a file containing a list of semantic information written in a formal way.

As discussed in Section 4, in our work, we decided to automate the generation of test cases for Java looping, switching, and typing semantics.

5.1. Looping

The semantic information for normal looping contains the initialization values for the generated loop statements and the number of times the loops are required to be executed. The number of required test cases for each looping rule can be also determined in the semantic information file (SIF). For nested looping, the tool user can specify the number of nested loops, the initialization loop values, and the number of times each of the nested loops is required to be executed.

In SIF, the keywords used in the looping information list are for_statement, while_statement, do_statement, initial, # of loops, and nested loops. For each of the looping rules: for, while, and do, the user has to write the rule name at the beginning in a separate line. In the next line the initial keyword has to be written, followed by a line containing the initial values of the loop statements required to be generated, separated by white spaces. The number of specified initial values determines the number of needed test cases. The next line holds the # of loops keyword, followed by a line containing the number of loops for each loop statement required to be generated, separated by white spaces. The number of values in this line should be equal to the number of values in the initial values line. If the tool user needs to generate nested loops for the looping rule then he has to type nested loops keyword in the next line. The nested loops line should be followed by four lines that contain the initial and number of loops information, as described earlier. However, the number of initial values for nested loops determines the number of nested loops. Thus, only one nested loop test case can be generated for each loop rule. In general, the SIF has to be terminated by %%%.

The tool stores the looping information in linked lists reached through an array of pointers. The attributes of the used array and nodes and the way of their linkage are shown in Fig. 2.

The four cells of the array point to the link lists that hold the values of loop initializations, number of loops, nested loop initializations, and number of times each of the nested loops required to be executed, respectively. Each node contains a field holding the alternative value and a pointer to the next alternative node. The number of nodes linked together to the first array cell determines the number of needed test cases. The number of nodes linked together to the third array cell determines the number of required nested loops.

The semantic information stored in the data structure is used to force the Extended Generate Minimum Statement routine to substitute some non-terminal symbols in the CFG by certain terminal rules, or to use certain CFG rule alternatives to generate the required statements.

Appendix B holds an SIF looping example and the generated test cases produced by the tool. The tool is powerful enough to be used to generate test cases to test the looping semantics of any compiler for any language, by only changing the programming language templates and considering the order of the used variables and values.

5.2. Switching

The semantic information used to test the Java switching contains the type of switch and case constant expressions and the number of case statements contained in the switch statement block. The case statements that contain break statements and the availability of a default statement can be also determined in the SIF.

In the SIF, the keywords used in the switching information list are switch, type, # of cases, break cases, and default. For the switch statements intended to be generated, the user has to write the switch keyword at the beginning in a
In the next line, the type keyword has to be written, followed by a line containing the type of the switch expression and the case constant expressions. The next line holds the # of cases keyword, followed by a line containing the number of case statements required to be contained in the switch statement block. If there are case statements intended to contain break statements, then the following line should contain the break keyword followed by a line containing the case statement numbers that contain the break statements, separated by white spaces. Finally, if the switch statement to generate contains a default statement, then the last line must contain the default keyword. If more than one switch statement required to be specified, then their specifications have to be separated by %.

The tool stores the switching information in a linked list, such that each of its nodes holds the specifications of a switch statement. Each node holds the type of the used expression, the number of case statements, a break pointer, a default flag, and a pointer to the next switch node. The break pointer points to a break linked list. Each of the break linked list nodes contain the number of the case statement that includes a break statement and a pointer to the next break node. The attributes of the used nodes and their linkage are shown in Fig. 3.

The switching semantic information stored in the data structure is used to force the Extended Generate_Mini_Statement routine to substitute some non-terminal symbols in the CFG by certain terminal rules, or to use certain CFG alternative rules to generate the required statements.

In order to cover the switching semantic rules listed in Section 4, the tool generates switch statements in which the values of the switch expression match the default statement and the first, the last, and a random ordered case statement. For each generated switch statement, a variable is declared as int is assigned to zero and incremented in each case and default statement. The tool is intelligent enough to compute the correct value of that variable, according to the Java language specification, when the switch statement terminates. In the generated test cases, the generated code compares the correct value with the actual value obtained by the compiler under test.

Finally, to cover the invalid test input, the tool generates three switch statements, the first one with more than one default statements, the second with more than one case statement with identical constant expressions, and the last one with an empty block.

5.3. Typing

The semantic information used to test the Java type compatibility contains primitive and reference type information. Required primitive type information contains the name of the type and a variable name of that type. Reference type information contains type names, object names, and classing and interfacing inheritance information.

In SIF, the keywords used in the typing information list are int, byte, char, short, long, boolean, array, class, extend, implement, object, and interface. For the primitive type case, the user has to write the type name at the beginning in a separate line followed by a variable name of that type. There are three reference types. For the class reference type, the user has to write the class keyword at the beginning. If the class extends another class, the user has to write the word extend followed by a line containing the name of the extended class. Next, if the class implements interfaces, the user has to write the interface keyword in a separate line. Each of the following lines should hold an interface name implemented by the class. Finally, the object keyword has to be written followed by a line containing the name of the class object. If another class needed to be declared, another class information has to be listed next as described earlier.

The class declaration part must be ended by a * symbol. The way of listing interface information is similar to the class one, except for the extension part. The extended interface names should be listed under the implement keyword.

The tool stores the primitive typing information in an 8-cell string array. The reference type information is listed in linked lists and an array of pointers. The attributes of the used reference type array and nodes and the way of their linkage are shown in Fig. 4.

The first and second cells of the pointer array are for classes and interfaces information, respectively. The reference node holds the name of the class or interface, the name of the extended class (for classes), a pointer to the names of implemented or extended interfaces, the name of the class or interface object, and a pointer to the next reference node. As shown in Fig. 5, the array reference type information can be stored in a linked list. Each array node holds the array type name, the array name, and a pointer to another array node.

The typing semantic information stored in the data structure is used to force the Extended Generate_Mini_Statement routine to substitute some non-terminal
symbols in the CFG by certain terminal rules, or to use certain CFG rule alternatives to generate the required statements.

Before generating test cases, the tool checks for repeated primitive and reference type names to prevent any naming compile-time errors. The tool also checks for the existence of extended class and implemented interface declarations and the existence of types referenced by the specified arrays. The generated correct and incorrect assignments and castings are generated according to a stored table similar to Table 1.

6. Summary and conclusions

In this paper, we proposed a procedure for the automation of syntax test cases generation for language compilers. In order to fit our purpose, we extended Purdom’s algorithm and implemented it in a test case generation tool. However, it is difficult to find an implemented algorithm to cover all the semantics of a language since semantics is language-specific. Therefore, we proposed a heuristic approach in which we attempt to test the semantic implementation of a language. This heuristic approach takes into consideration the nature of each semantic rule. The Java programming language was selected to apply our work, and the Java compiler version 1.1.4 for Windows 95 was partially tested and all test cases have passed successfully.

Table 2 summarizes the extent of test automation and the Java features that were covered in this paper.

In our future research, we plan to generate test cases that cover complex and advanced Java semantics such as exception handling and multi-threading. Then, we will perform a complete evaluation of existing Java language compilers such as Microsoft’s Visual J++ compiler and Symantec’s Visual Cafe compiler.

Acknowledgements

The authors would like to acknowledge the support of this work by Kuwait University Research Grant EE066. Also, we are thankful to the referee comments that helped improve an earlier version of this paper.

Appendix A. Example of Java syntax test cases

A.1. Looping

CFG for looping stored in the input file

<while_statement> ::= while (<expression>) (<statement>) %
<do_statement> ::= do <statement> while (<expression>); %
<for_statement> ::= for (<initialization> (<condition> <operation>)) <statement> <initialization> ::=;
|<forInitialization>;
<forInitialization> ::= <localVariableDeclaration> <condition> ::=;
|<expression>;
<operation> ::= empty |
|<forIncrement>;
|<topLevelExpression> <forIncrement> %
|<topLevelExpression>

Looping test case generated automatically
import java.awt.*;
public class testLoopingSyntax {
  public static void main(String args[]) {
    int x;
    do;while(x==1);
    while(x==1);
    for(x=1;x==1;++x, + + x);
    for(;;);
  }
}

Appendix B. Examples of Java semantic test cases

B.1. Looping

Contents of SIF for looping
for_statement
initial
0 1 8
# of loops
0 1 3
nested loops
initial
0 1 10
# of loops
1 2 10
while_statement
initial
0 1 8
# of loops
0 1 3
nested loops
initial
0 1
# of loops
1 2
do_statement
initial
0 1 8
# of loops
1 3 10
nested loops
initial
0 1
# of loops
1 2
%
%

The Java looping test cases generated by the tool automatically according to the above inputs:

```java
import java.awt.*;
public class testLoopingSemantic {
    void test_for () {
        int x, counter;
        counter = 0;
        for(x = 0; x < 10; x++) counter++;
        if ((counter == 0) and and (x == (0 + 0)));
            else System.out.print(‘‘the compiler failed in dealing with for statement semantically’’);
    counter = 0;
    for(x = 1; x < 2; x++) counter++;
    if ((counter == 1) and and (x == (1 + 1)));
        else System.out.print(‘‘the compiler failed in dealing with for statement semantically’’);
    counter = 0;
    for(x = 8; x < 11; x++) counter++;
    if ((counter == 3) and and (x == (8 + 3)));
        else System.out.print(‘‘the compiler failed in dealing with for statement semantically’’);
    //test for nested loops
    int a, b, c;
    counter = 0;
    for(a = 0; a < 1; a++)
        for(b = 1; b < 3; b++)
            for(c = 0; c < 20; c++)
                counter++;
    if ((counter == 20));
        else System.out.print(‘‘the compiler failed in dealing with for statement semantically’’);
    // test for infinite looping
    for(;;);
}
    void test_while () {
        int x, counter;
        counter = 0;
        x = 0;
        while(x < 0){x++; counter++;}
    if ((counter == 0) and and (x == (0 + 0)));
            else System.out.print(‘‘the compiler failed in dealing with while statement semantically’’);
    counter = 0;
    x = 1;
    while(x < 2){x++; counter++;}
    if ((counter == 1) and and (x == (1 + 1)));
        else System.out.print(‘‘the compiler failed in dealing with while statement semantically’’);
    counter = 0;
    x = 8;
    while(x < 11){x++; counter++;}
    if ((counter == 3) and and (x == (8 + 3)));
        else System.out.print(‘‘the compiler failed in dealing with while statement semantically’’);
    //test for nested loops
    int a = 0;
    int b = 1;
    counter = 0;
    while(a < 1){while(b < 3){counter++; b++;} a++;}
    if ((counter == 2));
        else System.out.print(‘‘the compiler failed in dealing with while statement semantically’’);
    //test for infinite looping
    while(x == x);
}
```
void test_do() {
    int x, counter;
    counter = 0;
    x = 0;
    do {x++; counter++;} while (x < 1);
    if ((counter == 1) and and (x == (0 + 1)))
        else System.out.println(‘‘the compiler failed in dealing with do statement semantically’‘);
}

B.2. Overloading

The following is a complete Java program (test case) to test the overloading concept in Java semantically. This code is generated manually.

import java.io.IOException;
import java.awt.*;

class testOverloadingSemantically {
    int name1(int x, int y, int z) {return 1;}
    int name1(int x, int y, int z, int w) {return 2;}
    int name1(int x, int y, float z) {return 3;}
    int name1(int x, int y, double z) {return 4;}
    int name1(int x, float y, int z) {return 5;}
    public static void main(String args[]){
        int c = 0;
        testOverloadingSemantically TOV = new testOverloadingSemantically();
        //testing first deference
        c = TOV.name1(4, 4, 4);
        if (c != 1) System.out.println(‘‘The compiler failed in dealing with Java overloading concept semantically’’);
        c = TOV.name1(4, 4, 4, 4);
        if (c != 2) System.out.println(‘‘The compiler failed in dealing with Java overloading concept semantically’’);
        //testing second and fourth deference
        c = TOV.name1(4, 4, 4, 4, 4);
        if (c != 3) System.out.println(‘‘The compiler failed in dealing with Java overloading concept semantically’’);
        c = TOV.name1(4, 4, 4, 2);
        if (c != 3) System.out.println(‘‘The compiler failed in dealing with Java overloading concept semantically’’);
    }
}

The following test cases should result compile-time errors:
import java.awt.*;
class testOverloadingSemantically {
    void name(int x, int y, int z) {}
    boolean name(int x, int y, int z) {return true;}
}
import java.io.IOException;
import java.awt.*;
class testOverloadingSemantically {
    void name (int x, float y, int z) {}
    void name (int x, float y, int z) throws IOException {}
}
import java.awt.*;
class testOverloadingSemantically {
    private void name (float x, int y, int z) {}
    public void name (float x, int y, int z) {}
}

References