

# Survey of Compiler Testing Methods

A. S. Kossatchev and M. A. Posypkin

*Institute of System Programming, Russian Academy of Sciences,  
Bol'shaya Kommunisticheskaya ul. 25, Moscow, 109004 Russia  
e-mail: kos@ispras.ru, posypkin@ispras.ru*

Received September 12, 2004

**Abstract**—Compilers are used for creating executable modules for programs written in high-level languages; therefore, the presence of errors in a compiler is a serious danger for the quality of the software developed with the use of this compiler. As in the case of any other software, testing is one of the most important methods of quality control and error detection in compilers. The survey is devoted to methods for generating, running, and checking the quality of compiler test suites, which are based on formal specifications of the programming language syntax and semantics.

## 1. INTRODUCTION

High-level languages have long been a basic tool for the development of software. This explains why programs supporting this development process (in particular, compilers) are widely used.

Compilers translate programs from high-level languages into representations executable by the computer. If there are errors in the compiler, the original program is translated into an executable module the behavior of which is different from that determined by the semantics of the original program. Errors of this kind are difficult to reveal and correct, and their presence questions the quality of the components generated by the compiler. Undoubtedly, the compiler correctness is fundamental to the reliable operation of any software developed by means of this compiler, and the compiler correctness check is critically important for improving the software reliability.

As in the case of any other software, testing is one of the most important methods of quality control and error detection in compilers. Traditionally, testing methods are divided into two groups: “white” and “black” box methods. In the former case, the tests are created on the basis of information about the implementation, with the use of the source code of the product being tested. In the latter case, the test generation is based only on the functional description, which is also referred to as *specification*.

Both methods have their advantages and disadvantages. The advantage of the “white box” method is that it allows a complete check of the efficacy of the source program text to be performed. Such a check allows us to detect many mistakes in the implementation but does not guarantee that the desired functionality is implemented in the system. For the latter purpose, the “black box” methods are used, which are designed for checking the correspondence of the implementation to the requirements for the system being tested.

One of the disadvantages of the “black box” methods is that they require additional efforts for the development of the product specification. In the case of compilers, this disadvantage is not very important, since, for the programming languages, there exists, as a rule, informal syntax and semantics description (language standard) and a formal syntax description (grammar). In certain cases, a formal, complete or partial, semantics description can also be available. This makes the use of the “black box” methods for testing compilers attractive. It is these methods that are discussed in this survey.

The paper is organized as follows. In Section 2, basic notions of the theory of formal languages and compilation needed for the following discussion are briefly considered. Sections 3–7 are devoted to testing of various stages of the compilation process. Conclusions from this survey are summarized in Section 8.

## 2. BASIC NOTIONS OF THE COMPILATION THEORY

*Compilation*, in the most general sense, is the process of transformations of the source program written in an input language to a program in an output language. Traditionally, the input language is specified by means of a formal grammar.

A *grammar* is a tuple  $\langle N, T, s, P \rangle$ , where  $N$  is a finite set of nonterminal symbols (nonterminals);  $T$  is a set of terminal symbols (terminals), which does not intersect with  $N$ ;  $s$  is an initial symbol from  $N$ ; and  $P$  is a finite subset of the set  $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ . Pairs from the set  $P$  are called *grammar rules*. A rule  $(n, m)$  is usually written as  $n \rightarrow m$ .

A chain  $w$  is said to be *directly derivable* from a set  $u$  if  $u = abc$ ,  $w = adc$ , and  $b \rightarrow d \in P$ . The direct derivability is denoted as  $u \Rightarrow^p w$ , where  $p = b \rightarrow d$ .

The *derivability* relation is defined as a transitive and reflexive closure of the direct derivability relation and is denoted as  $\Rightarrow^*$ . The expression  $u \Rightarrow^* w$  is read as “ $w$  is derivable from  $u$ .”

The set of chains derivable from the starting symbol of the grammar  $G$  and containing only terminal symbols is called the *language generated by the grammar*  $G$  and is denoted as  $L(G)$ . Such chains are called *language phrases*.

A grammar  $\langle N, T, s, P \rangle$  is said to be *context-free* if any rule has the form  $n \rightarrow w$ , where  $n \in N$ ,  $w \in (N \cup T)^*$ .

Formal grammars are convenient means for specifying the language syntax. The mechanism of the context-free grammars is not sufficient for defining *semantics of the programming language*.

It is generally accepted to divide the semantics of the programming languages into *static* and *dynamic* semantics. Static semantics deals with the problem of the correct use of types in a program, scopes of the identifiers, and other properties of the program that can be determined without running the program.

The most often used means for describing static semantics are *attribute grammars*. In the attribute grammars, each symbol of the grammar is made to correspond to a finite set of attributes, and each grammar rule, to a set of rules for evaluating the attributes corresponding to the symbols of the rule. Using these rules, one can evaluate attributes of each node of the parsing tree.

Each rule can be associated with a *context condition*, which is a predicate depending on the attributes of this rule. A program is considered to be statically correct if, after the evaluation of all attributes, the context condition is violated in none of the nodes of the parsing tree; i.e., the corresponding predicate takes the *true* value.

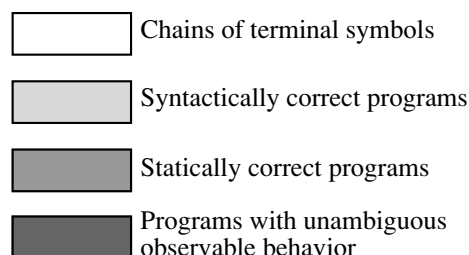
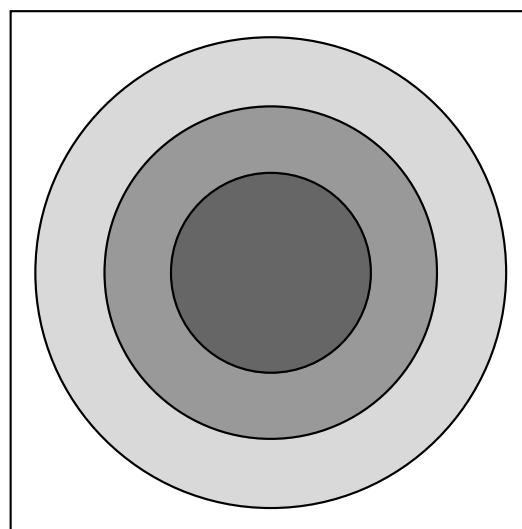
The attributes can be inherited, i.e., evaluated through the attributes of the parent node in the tree, or synthesized, i.e., be functions of the attributes of the descendant nodes. The attributes are made to correspond to various static properties of the program and are used for evaluating types of data and expressions, classes of memory, and values of constant expressions.

The dynamic semantics of a programming language defines the meaning of the execution of programs in this language. Unlike in the case of the syntax and static semantics, a unique formal, commonly accepted, description of the dynamic semantics does not exist.

Traditionally, the compilation process is divided into the following stages [1]:

1. Lexical and syntax analyses. At this stage, the source program text is analyzed, and the parsing tree is constructed.

2. Analysis of the static semantics. At this stage, the parsing tree is analyzed, attributes at all nodes of the tree are calculated, and the static correctness of the program is verified.



Structure of the compiler input data. Circles denote embedded subsets of the set of the compiler input data.

3. Optimizing transformations. At this stage, various transformations of the internal program representation aimed at improving the quality of the program in accordance with the selected criterion (code size, operation rate) are carried out.

4. Code generation. The code generation consists in creating an executable file by the given internal program representation.

Functional decomposition of the majority of the compilers, generally, corresponds to this scheme, although each particular implementation has its own specific features. For example, some compilation stages are decomposed in several finer substages. Some stages can be implemented in one, rather than several, component of the system.

Let us consider the structure of the compiler input data in more detail (see figure) and discuss how the subsets of this structure are usually used for testing various compiler stages.

The syntax of a language is specified by a grammar containing terminal symbols, nonterminal symbols, and productions. Chains of terminal symbols derivable from the start symbol of the grammar are called syntactically correct programs. The set of syntactically correct programs is a subset of the set of all chains of terminal symbols. They are used for checking of whether the syntax analysis stage recognizes a test as correct.

Such programs are usually called *positive test cases*. The chains that cannot be derived from the start symbol, the so-called *negative test cases*, are used for checking the compiler's ability to recognize a syntax error.

Static semantics is defined only for syntactically correct programs; it defines rules for computing program properties that can be determined without running the program. These properties include, in particular, types of variables and expressions. In addition to the computation rules, rules for checking static program correctness, or context conditions, are specified, which impose constraints on possible combinations of values of static program properties.

From the testing standpoint, both programs satisfying context conditions (positive test cases) and those violating them (negative test cases) are of interest. The programs satisfying context conditions are called statically correct programs. Statically correct programs constitute a subset of the set of syntactically correct programs. They are used for testing of whether the static analysis stage in the compiler correctly recognizes well-formed programs. The syntactically correct programs violating the context conditions are used for checking the compiler's ability to recognize static errors.

Dynamic semantics of a programming language defines the meaning of the execution of statically correct programs in this language. To test its implementation in the compiler, statically correct programs are used, which are compiled by the compiler being tested; the loadable modules obtained are executed, and their observable behavior is compared with the reference behavior determined by the dynamic language semantics. Clearly, if the program observable behavior is ambiguous, such a comparison is a very complicated task. Therefore, for testing implementations of the dynamic semantics in a compiler, statically correct programs with unambiguous observable behavior are used. Such programs constitute a subset of the set of statically correct programs.

Depending on what stage is required to test, the object of the study is one or another subset of the structure shown in the figure. Further in this survey, for each of the above-listed subsets, we consider various approaches to solving the following testing problems:

- (1) test case generation (writing);
- (2) returning a verdict on whether the test was passed (test oracle);
- (3) estimation of the quality of the test suite (coverage criterion).

### 3. TESTING OF SYNTAX ANALYZER

The language syntax is, perhaps, the only aspect of the programming languages the need of a formal definition of which is commonly recognized. Grammars, owing to their nature, are an ideal tool for generating

program tests: they define a language as a set of phrases that can be generated by successively applying rules to an initial symbol. Thus, an algorithm for generating programs in a language can easily be constructed by applying and combining (in a random or regular way) grammar rules. In view of the wide use of grammars as means for describing languages and by virtue of their "generating" nature, it is the grammars that are basically used for generating test programs for compilers.

The fundamental work in this field is the paper [2]. Later, it was used as a departure point by many other researchers, who either modified the Purdom algorithm suggested in this work or compared it with their own algorithms.

The input data for the Purdom algorithm is a context-free grammar in which the starting symbol occurs only once, on the left-hand side of a rule. Such a constraint does not restrict the class of the languages under consideration, since any context-free grammar can be reduced to this form. The algorithm constructs a set of phrases derivable in this grammar in such a way that each grammar rule is applied not less than once. In doing so, the problem is posed to minimize this set.

The algorithm works as follows. First, for each non-terminal symbol, the minimal length of the derivation tree is calculated, which is required in order to obtain a string of terminal symbols. This information is used in the second part of the algorithm.

The second part of the algorithm uses a stack for storing the generated phrases. Initially, this stack contains the starting symbol of the grammar. Then, a loop is performed that either prints the symbol located at the top of the stack if this symbol is terminal or, if the symbol is not terminal, replaces it by the right-hand side of the rule the left-hand side of which contains this symbol. The rule to be used in the replacement is selected from those that have not been used yet (if this is impossible, from the remaining ones) as the rule with the minimal inference length. These actions are repeated until the stack is empty. The phrase generated is added to the test suite, the starting symbol is placed into the stack, and the second part of the generation algorithm is repeated. The loop terminates when each grammar rule has been used at least once each.

The Purdom algorithm actually generates compact test suites for grammars. Data regarding the application of this algorithm to generating tests for C and C++ grammars have been presented in [3]. For the C grammar, which contains 211 rules, 11 phrases have been generated; for the C++ grammar containing 824 rules, the test suite contained 81 phrases.

This algorithm allows one to generate phrases satisfying the *rule coverage criterion*, which requires that each rule be used not less than once when deducing phrases of the test suite. The fulfillment of this criterion is a necessary requirement for any test suite. Later on, Lämmel showed [4] that the test suites based on the rule coverage are not capable of revealing very simple errors



in the grammar rules and suggested a more powerful *context-dependent rule coverage criterion*.

To introduce this criterion, we need the following definition.

**Definition 1.** Let  $G = \langle N, T, s, P \rangle$  be a context-free grammar. A *direct occurrence* of a nonterminal symbol  $n$  in the grammar is any rule the right-hand side of which contains  $n$ .

The coverage criterion itself is defined as follows.

**Definition 2.** Let  $G = \langle N, T, s, P \rangle$  be a context-free grammar. A phrase  $w \in T^*$  is said to *cover the rule*  $p = n \rightarrow z$  for the *direct occurrence*  $q = m \rightarrow unv$  of a symbol  $n$  if there exists the derivation  $s \Rightarrow^* xmy \Rightarrow^q xunvy \Rightarrow^p xuzvy \Rightarrow^* w$ . A set of phrases  $W$  is said to satisfy the *criterion of the context-dependent rule coverage for the grammar*  $G$  if, for any rule  $p = n \rightarrow z$  and any direct occurrence  $q$  of the symbol  $n$ ,  $W$  contains a phrase that covers the rule  $p$  for the occurrence  $q$ .

Lämmel [4] showed that the context-dependent rule coverage criterion enables creation of test suites capable of revealing a wider class of errors compared to the rule coverage criterion.

Another coverage criterion that takes into account the context of the grammar rule application is suggested in [5]. The grammar notation makes use of the BNF variant, which admits grouping alternatives with the help of parentheses. Each subphrase and all nonterminal symbols are called *branching points*.

Each branching point is associated with a set of terminal symbols that belong to a phrase derivable from it. This set is called a *frontal set* for the given branching point. The set consisting of all pairs  $(n, t)$ , where  $n$  is a branching point and  $t$  is a terminal symbol from the frontal set for  $n$ , is called a *set of situations for the given grammar*. The coverage criterion is formulated as follows.

**Definition 3.** A phrase  $w$  covers a situation  $(n, t)$  for the given grammar if there exists an inference  $s \Rightarrow^* m \Rightarrow^r m' \Rightarrow^* w$  such that  $t \in w$  and the right-hand side of a rule  $r$  contains the branching point  $n$ . A set of phrases  $W$  is said to satisfy the *situation coverage criterion* if, for each situation from the set of situations of the given grammar, there exists a phrase from  $W$  that covers it.

This coverage criterion is oriented to the LL parsers, for which the analysis of the branching points is an important component of the syntax analysis algorithm.

Other works devoted to the grammar-based test generation [6–10] do not introduce any coverage criteria for the test quality analysis. If a coverage criterion is not used, then it is not clear when the test generation should be completed. Since almost all grammars of real programming languages are recursive, a mechanism preventing infinite recursion is required. For this purpose, constraints on the number of rule applications or probabilistic approaches are introduced.

Guilmette [6] suggests an algorithm of phrase generation by a context-free grammar based on the phrase unfolding, which begins with the start symbol. First, the leftmost nonterminal symbol is unfolded. After this symbol has completely been unfolded, it is replaced by the string of symbols obtained, and the algorithm is applied to the next nonterminal symbol from the left. The probability that a rule is selected for the next unfolding reduces along the unfolding branch after each application of the rule. Since the application of the rule after which a nonterminal is unfolded into a sequence of nonterminal symbols stops the generation process, this method gradually reduces the probabilities of the applications of other rules in the course of the unfolding and guarantees the termination of the process in a finite time.

The alternative method based on *stochastic grammars* [11] was employed in works [7, 8, 12]. The difference between the stochastic grammars and the ordinary ones consists in that each rule in the former is characterized by a number, *weight of the rule*, which determines the relative frequency of the selection of this rule in the generation process. A stochastic grammar is said to be consistent if it defines a *stochastic language* with the following basic property: for any arbitrarily small number  $\epsilon$ , there exists a positive integer  $N$  such that the total probability that the length of a phrase in this language is greater than  $N$  is less than  $\epsilon$ . This property guarantees that the generation algorithm terminates in a finite time. The consistency property of the stochastic grammars is studied in [11].

The works mentioned above are devoted to the generation of phrases of a language, i.e., chains of terminal symbols satisfying the syntax requirements. The phrases are used for checking whether the syntax analyzer correctly recognizes well-formed programs. As is known from practice, the testing of whether the syntax analyzer correctly processes programs containing syntax errors is also very important. For this purpose, programs from the complement of the set of syntactically correct programs in the set of all possible chains of terminal symbols (negative tests for the syntax in the figure) are used. Automated generations of such programs and the coverage criteria for them are considered in [13].

The approach suggested in [13] is based on constructing, for each terminal symbol  $t$ , a set  $F_t$  of possible terminal symbols that can immediately follow  $t$  in some phrase of the language. The approach uses also the complement  $N_t$  of the set  $F_t$  in the set of all terminal symbols. This set contains symbols that cannot immediately follow  $t$  in the language phrases. The coverage criteria for positive tests suggested in this work are intended for covering all admissible sequences consisting of two successive terminal symbols in various grammar contexts. In contrast to this, for negative tests, possible inadmissible combinations are considered. The paper describes algorithms for generating sets  $F_t$  and  $N_t$ , thus proving the existence of such algorithms.

#### 4. TESTING OF THE STATIC SEMANTICS ANALYSIS STAGE

To test the static analysis stage in a compiler, statically correct programs, as well as programs from the complement of the set of statically correct programs in the set of syntactically correct programs (the so-called *negative tests for static semantics*) are used (see figure).

To specify static semantics, attribute grammars are most often used. An attribute grammar extends the context-free grammar through the incorporation of variables called *attributes*, semantic functions, and context conditions. Each grammar rule is associated with a set of semantic functions expressing values of a node attribute in terms of the attributes of the parent nodes and attributes of other descendants (inherited attributes). In addition, each rule is associated with a context condition that determines admissible values of the attributes of this node. A program is said to be *statically correct* if, after computing all attributes in the tree of the abstract syntax, all context conditions are true.

In [14], a technique for constructing tests for the language static semantics based on its VDM description is proposed. The authors suggest a criterion of the test suite completeness based on the coverage of the context conditions. The fulfillment of the criterion means that the test suite contains all possible combinations of the attribute values on which the context condition is true. For this purpose, the context condition is represented as a conjunction of disjunctions, and all combinations of the attribute values for which each disjunction in the formula takes the *true* value are selected.

In [15], an algorithm for the generation of test programs by a description of the language static semantics given in the form of an attribute grammar is suggested. For each grammar rule, a statically correct test program derived with the help of this rule is generated.

For each rule, the construction of the test begins with a list of strings containing one element, the right-hand side of this rule. At each iteration, from the list of strings constructed, the algorithm selects a string with the least estimated length of the inference required for transforming it to a string of terminal symbols. If this string is a program, i.e., a string of terminal symbols derivable from the start symbol, then the algorithm stops. Otherwise, the list is supplemented by a set of phrases obtained from the given phrase by one of the following methods:

(1) a set consisting of all strings that can be obtained by substituting the given string for one of the nonterminals in the grammar production rules is constructed;

(2) one of the nonterminals in the string is selected and is replaced by a set consisting of all strings obtained by substituting the right-hand sides of the production rules the left-hand side of which coincides with this terminal.

The method to be used in a particular case is selected heuristically with the objective of obtaining an incorrect string as soon as possible. The fact of the string incorrectness is determined by evaluating all attributes of the string and verifying the context conditions. The strings identified as incorrect are deleted from the list and do not take part in the subsequent generation.

The algorithm stops in the two following cases: (i) when a phrase in the language has been constructed or (ii) when the list is empty. The latter implies that the given production rule cannot be a part of the derivation of a correct program; i.e., there is an error in the description of the attribute grammar.

The algorithm considered generates suites of statically correct tests satisfying the rule coverage criterion, which was suggested by P.A. Purdom for the language syntax and does not take into account the language semantics. This disadvantage was noted by the author of the algorithm, and, in his later papers written jointly with R. Lämmel, new coverage criteria are suggested, which are based on a combination of the rule coverage criterion (context-dependent or context-independent) and that of the attribute values coverage. The latter is a partition (specified by the user) of the set of attribute values into domains.

The test generation algorithm consists in the successive generation of tests satisfying the syntax coverage criterion by an algorithm similar to Purdom's algorithm and the subsequent discarding of statically incorrect tests. The generation algorithm stops either when the coverage criterion is fulfilled or when the given constraint on the number of tests is exceeded.

The reachability of the coverage criterion introduced by J. Harm and R. Lämmel is an algorithmically unsolvable problem, and, hence, the question of whether the test generation algorithm stops is undecidable. Therefore, a constraint on the maximal number of generated tests is required.

It is noted in [16, 17] that it is advisable to have tests that violate the static semantics constraints. The authors do not formulate coverage criteria and do not suggest algorithms for generating such tests.

In [18], the use of attribute context-free grammars for generating test data is described. The authors allow the so-called *hybrid attributes* to be used for passing parameters up and down the parsing tree. The following example illustrates this:

$$E(K) ::= [? K = 0] \text{ "a" } | \\ [ [? K = 1] \#K := 0 ] \text{ "b" }$$

Here, the nonterminal  $E$  can be unfolded as  $a$  or  $b$  depending on the value of  $K$  (note that, in the latter case, this value will be modified). The expression in the square brackets beginning with '?' denotes the *guard* that must be a logical expression containing inherited node attributes or synthesized attributes of the nodes on the left of the terms. The expression in the square

brackets beginning with '#' denotes the evaluation of the attribute value.

The generation begins with the start symbol, which is unfolded successively in accordance with the grammar rules from left to right. The rules are selected in a random or regular way (which may be specified by the user). In the course of the test generation, rules of the evaluation of attributes and constraints are fulfilled. If a constraint for a rule is not satisfied, another unfolding rule is applied.

Attribute values can be specified either exactly or in terms of certain ranges. In the latter case, when selecting the rules, the values of the attribute from the range are searched through to ensure its optimal coverage. This may be an exhaustive search of all values from the range or selection of boundary and average values.

Methods of automated test generation and coverage criteria for negative tests for the language semantics by its formal description are suggested in [19]. In this paper, the so-called *constraint coverage criterion* is proposed, which is based on the analysis of the causes of violation of the context condition. In the majority of cases, the context condition can be formulated as simultaneous fulfillment of several semantic conditions, the so-called basic conditions. The cause of the violation of the context condition may be the violation of any basic condition. A coverage of the constraints is achieved on a suite of negative tests if, for any such a cause, there is a test that violates the semantics in accordance with this cause. The test generation algorithm consists in constructing syntactically correct tests and subsequently filtering them: the test suite is supplemented by programs that violate the context conditions and enhance the coverage. The attainability of this coverage criterion for the specifications not containing redundant basic conditions is proved in the paper.

In this section, we have discussed some approaches to the test generation and coverage criteria for the static semantics. The test oracles for the static semantics usually do not check the degree of correctness of the static information and only verify whether the correct tests are recognized correctly, whether the compiler generates codes for them, and whether the incorrect tests lead to the error diagnosis.

## 5. TEST ORACLES FOR THE OPTIMIZING TRANSFORMATION AND CODE GENERATION STAGES

The optimization and code generation stages should transform the original program in such a way that the *observable behavior* of the program obtained would correspond to the semantics of the original program. Under the observable behavior, we mean the effect of the interaction of the program with the environment, for example, output of information on the display, information transfer through a network, and the like.

The approach to the program testing based on the analysis of the observable behavior of the system being tested is described in [20]. In accordance with this approach, an implementation is considered to conform to the specification if, for each test, the observable behavior of the implementation corresponds to the observable behavior of the specification on the same test. The test is a system that is external with respect to the specification and implementation and capable of interacting with them and fixing their behaviors. The testing process consists in comparing observable behaviors of the specification and implementation on a set of tests.

To implement this approach, statements for printing values of variables [12, 19, 21–23] or expressions [9] are inserted into the test program used for testing the compiler. For the evaluation of reference results, an attribute grammar [21], or a partial specification of the dynamic semantics [9], or a different implementation of the compiler [12] are used. In [19, 22, 23], to obtain the reference observable behavior, an interpreter of the dynamic semantics is used. Usually, a simplified model, in which the program has no input data, is considered. Under such an approach, it is sufficient to evaluate the reference results only once and then use them when running the tests.

The testing process consists in the test compilation and execution of the load module followed by the comparison of the output flow with the reference one for the given test. If the results are different, the verdict of an error is returned.

The comparison of the observable behaviors can easily be done if the program behavior is unambiguous; otherwise, this is simply impossible. Thus, one of the requirements imposed on a test for the dynamic semantics is the uniqueness of its observable behavior. The fulfillment of this requirement must be guaranteed by the test generation algorithm.

## 6. TESTING OF THE OPTIMIZING TRANSFORMATIONS IN A COMPILER

To test the optimizing transformations, tests are generated in such a way that the constructs to which the optimizing transformations are applied occur in different combinations. The algorithms designed for the generation of general-purpose tests, which are aimed at testing general syntax and semantics of a language are not appropriate, since they will either fail to generate the required tests or generate an insufficient number of the tests.

A test generator for testing optimizing transformations in FORTRAN compilers is described in [21]. The input data for the generator are an attribute grammar and generator adjustments, which allow the user to control the test generation. The adjustments include:



(1) quantitative constraints (maximum nesting level of loops and branching statements, maximal number of statements in the loop body, and so on);

(2) types of data and variables used in the test programs;

(3) relative probabilities of the grammar rule applications.

The generation algorithm consists in the successive unfolding of nonterminals beginning with the start symbol. The unfolding rules are selected randomly, with the probabilities being specified by the user. The inherited attributes allow a grammar rule to be selected only when the semantic constraint is true, thus restricting the number of the generated tests that violate the semantics.

For the generated programs, the attributes are completely evaluated, and the verdict of their static correctness is returned. The incorrect programs are discarded. The verification of the dynamic correctness reduces to checking whether there are overflows and divisions by zero in the expressions; this is carried out at the code generation stage. In other words, the language subset for which the tests are generated is subjected to constraints that permit algorithms that generate only correct programs, the results of the operation of which can be evaluated statically. One of such constraints is, for example, the condition that all loops perform exactly one iteration.

The authors do not suggest any coverage criteria for the test selection, explaining this by the fact that the metrics based on the rule coverage are not necessarily adequate for estimating the quality of a test suite. The paper also cites results of practical experiments that substantiate the efficiency of the suggested approach.

The automated generation of programs for testing optimizing compilers is studied in [24, 25]. For each type of the optimizing transformations, a *model language* is constructed, which defines the language subset that contains only the constructs to which this type of transformations can be applied.

The use of the model language allows one to abstract from insignificant (from the standpoint of the given optimizing transformation) possibilities and generate the tests that test only this transformation. An appropriate coverage criterion for the optimizing transformation being tested is defined in terms of the model language.

The test generator consists of two components: an *iterator*, which successively generates programs in the model language on the basis of the coverage criterion, and the so-called *mapper*, which maps programs in the model language into programs in the target language. The verification of the correctness of the compiler operation on a particular test consists in the comparison of the observable program behaviors with the optimization turned on and off.

## 7. TESTS FOR CHECKING DYNAMIC SEMANTICS

The tests based on the dynamic semantics of the language are used for checking the correctness of the code generated by the compiler or correctness of the interpreter operation. For this purpose, statically correct programs with uniquely defined observable behavior are used (see figure).

The generation of dynamically correct programs is a more complicated task compared to the generation of statically correct ones. This is explained, first of all, by difficulties associated with the formalization of the dynamic semantics of the programming language. Unlike the syntax and static semantics, which are successfully described by means of formal grammars, the dynamic semantics has no unique formal, commonly accepted, description. The existing formal descriptions of the dynamic semantics of real programming languages are seldom used in practice either by the compiler designers or by the experts in language standardization.

We begin with the approaches that do not assume an automated test generation. In works [26–28], test generation methods based on the tables and situation diagrams constructed by a text description of the language are considered. Each situation contains a description of input data (programs in the programming language) and the effect. The latter may be a situation of the correct termination of the compilation, or error diagnosis in the cases of tests for the static semantics, or the result of the program operation in the case of tests intended for the dynamic semantics.

Tests are divided into the following three categories: correct tests, which are correct programs in C++; statically correct tests, which are designed for the verification of whether the translator can recognize complex language constructs; and test programs, codes containing one or more semantic errors. The last category is intended for testing the compiler's ability to find static semantics errors.

The work [6] is devoted to the *TGGS* system, which makes it possible to generate programs by the grammar in which each rule is associated with a guard and a semantic action. When selecting a current rule, the semantic action is performed and the guard is computed. If the value of the guard is "false," then another rule is selected.

This method of test generation can be used for creating statically correct programs with an unambiguous observable behavior. The author illustrates this by an example of a simple language consisting of programs that can assign certain values to registers and print their contents. The generator is supplemented by a semantic action, which makes it possible to know what registers were assigned values, and a guard, which ensures that only the content of the initialized register is printed. It is not quite clear from the paper whether the approach suggested could successfully be applied to real lan-





cessively through the states  $st_1, \dots, st_n$ , and, in the state  $st_i$ , rules from  $T_i$  are satisfied for all  $i, 1 \leq i \leq n$ .

**Definition 2** (coverage criterion by  $n$ -paths). A test suite satisfies the coverage criterion by  $n$ -paths if, for any feasible  $n$ -path, there exists a program from the suite the execution of the dynamic semantics of which passes successively through the states  $st_1, \dots, st_n$ , and, in the state  $st_i$ , rules from  $T_i$  are satisfied for all  $i, 1 \leq i \leq n$ .

In that paper, tools for automated coverage measuring and a test generation scheme that uses these tools for filtering coverage-driven tests are considered. The question of whether the criteria introduced are reachable is open; the authors do not suggest any algorithms that could automatically solve this problem for the suggested coverage criteria.

## 8. CONCLUSIONS

As follows from the above discussion, there currently exist many test generation methods for all compilation stages. The test generation by a context-free grammar is the most examined among them. For this case, the coverage criteria, which are proved to be reachable, and the algorithms of the test suite generation satisfying these criteria have been suggested.

For the tests intended for the verification of implementations of the static and dynamic language semantics in a compiler, a number of coverage criteria have been suggested. However, in contrast to the syntax-oriented tests, the reachability of these tests has not been proved, and no automatic algorithms for determining this have been suggested.

From the practical standpoint, it can be stated that the algorithms for generating syntax-oriented test suites are time-tested ones. Test suites for complete grammars of some languages, such as COBOL [4], C, and Java [13], have been generated. There remains a question of whether the use of automated methods for generating semantics-oriented tests by the complete description of the static and dynamic semantics of a real programming language is justified from the practical standpoint, since the approaches discussed have been tested on subsets of real languages or on model programming languages.

## ACKNOWLEDGMENTS

We are grateful to S. Zelenov and M. Naprasnikova, research scientists at the Institute of System Programming, Russian Academy of Sciences, for their valuable advice on the content of this survey. We also thank R. Guilmette for his consultations on the test generation algorithm.

## REFERENCES

1. Serebryakov, V.A., *Leksii po konstruirovaniyu kompilyatorov* (Lectures on Compiler Design), Moscow, 1993.
2. Purdom, P.A., A Sentence Generator for Testing Parsers, *BIT*, 1972, vol. 2, pp. 336–375.
3. Malloy, B.A. and Power, J.F., An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases, *Proc. of ICIS'01*, 2001.
4. Lämmel, R., Grammar Testing, *Lecture Notes in Computer Science* (Proc. of Fundamental Approaches to Software Eng. (FASE), 2001), Berlin: Springer, 2001, vol. 2029, pp. 201–216.
5. Zelenova, S.V., Demakov, A.V., and Zelenov, S.A., Testing of Text Parsers in Formal Languages, *Programnyye sistemy instrumenty*, 2001, no. 2.
6. Guilmette, R.F., *TGGS: A Flexible System for Generating Efficient Test Case Generators*, 1999.
7. Maurer, P.M., Generating Test Data with Enhanced Context-Free Grammars, *IEEE Software*, 1990, pp. 50–55.
8. Maurer, P.M., The Design and Implementation of a Grammar-Based Data Generator, *Software Practice Experience*, 1992, vol. 22, no. 3, pp. 223–244.
9. Sirer, E. and Bershad, B., Using Production Grammars in Software Testing, *Proc. of the Second Conf. on Domain-Specific Languages*, 1999.
10. Automated Syntax Testing Using JSynTest™, <http://software.qip.us/jsyntest.htm>.
11. ter Doest, H., Stochastic Languages and Stochastic Grammars, *Proc. of the 1994 Groningen Student Conf. on Comput. Sci.* (Groenbaum, H.M., Kleijn-Hesselink, H.W., and Lankhast, M.M., Eds.), 1994, pp. 51–59.
12. McKeeman, W., Differential Testing for Software, *Digital Tech. J.*, 1998, vol. 10, no. 1, pp. 100–107.
13. Zelenov, S.V. and Zelenova, S.A., Automatic Generation of Positive and Negative Tests for Testing Syntax Analysis Stage, *Programmirovaniye* (in press).
14. Petrenko, A.K., Chatskina, T.A., Borisova, M.V., and Morozova, T.A., *Testirovaniye kompilyatorov na osnove formal'noi modeli yazyka* (Compiler Testing Based on Formal Model of Language), Moscow: Institut prikladnoi matematiki im. M.V. Keldysha Ross. Akad. Nauk, 1992.
15. Harm, J., Automatic Test Program Generation from Formal Language Specification, *Rostocker Informatik-Berichte*, 1997, vol. 20, pp. 33–56.
16. Harm, J. and Lämmel, R., Two-Dimensional Approximation Coverage, *Informatika*, 2000, vol. 24, no. 3.
17. Harm, J. and Lämmel, R., Testing Attribute Grammars, *Proc. of the Third Workshop on Attribute Grammars and Their Applications*, 2000, pp. 79–98.
18. Duncan, A.G. and Hutchinson, J.S., Using Attributed Grammars to Test Design and Implementations, *Proc. of the 5th Int. Conf. on Software Eng.*, 1981, pp. 170–178.
19. Kalinov, A., Kossatchev, A., Petrenko, A., Posypkin, M., and Shishkov, V., Coverage-Driven Automated Compiler Test Suite Generation, *Proc. of LDTA'2003*, Elsevier, 2003, vol. 82.
20. Tretmans, J., *Testing Techniques*, 2002.
21. Burgess, C.J. and Saidi, M., The Automatic Generation of Test Cases for Optimizing Fortran Compilers, *Information Software Technol.*, 1996, vol. 38, pp. 111–119.
22. Kalinov, A., Kossatchev, A., Posypkin, M., and Shishkov, V., Using ASM Specification for Automatic Test Suite Generation for mpC Parallel Programming Language Compiler, *Proc. of the Fourth Int. Workshop on Action Semantics AS 2002*, 2002, pp. 96–106.

23. Kossatchev, A.S., Kutter, P., and Posypkin, M.A., Automated Generation of Strictly Conforming Tests Based on Formal Specification of Dynamic Semantics of the Programming Language, *Programmirovaniye*, 2004, no. 4, pp. 52–67.
24. Zelenov, S.V., Zelenova, S.A., Kossatchev, A.S., and Petrenko, A.K., Generation of Tests for Compilers and other Text Processors, *Programmirovaniye*, 2003, no. 2, pp. 59–69.
25. Kossatchev, A.S., Petrenko, A.K., Zelenov, S.V., and Zelenova, S.A., Application of Model-Based Approach for Automated Testing of Optimizing Compilers, *Proc. of the Int. Workshop on Program Understanding*, 2004, pp. 81–88.
26. Kaufman, V.Sh., Standardization and Control of Translators, in *Razlichnye aspekty sistemnogo programmirovaniya* (Various Aspects of System Programming), 1984, pp. 47–85.
27. Sukhomlin, V.A., Triple Standard 3C++ Programming System, *Int. Conf. "Open System Development and Application"*, 1997, pp. 37–47.
28. Baskakov, Yu.V., Principles of Construction of Test Suites for Testing Compiler Conformity with Programming Language Standards, *Teoreticheskie i prikladnye problemy informatsionnykh tekhnologii* (Theoretical and Applied Problems of Information Technologies), Sukhomlin, V.A., Ed., 2001.
29. eXtensible Abstract State Machines, <http://www.xasm.org>.
30. Kutter, P. and Pierantonio, A., Montages: Specifications of Realistic Programming Languages, *J. Universal Comput. Sci.*, 1997, vol. 3, no. 5, pp. 416–442.
31. Lastovetsky, A., mpC—A Multi-Paradigm Programming Language for Massively Parallel Computers, *ASM SIGPLAN Notices*, 1996, vol. 31, no. 2, pp. 13–20.
32. Gurevich, Y., *Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods*, Börger, E., Ed., Oxford Univ. Press, 1995, pp. 9–36.