ELSEVIER

Information and Software Technology 39 (1997) 617–625

INFORMATION AND SOFTWARE TECHNOLOGY

Compiler test case generation methods: a survey and assessment

A.S. Boujarwah, K. Saleh*

Kuwait University, Department of Electrical and Computer Engineering, P.O. Box 5969, Safat 13060, Kuwait

Received 12 November 1996; revised 3 March 1997; accepted 12 May 1997

Abstract

Software testing is an important and critical phase of the application software development life cycle. Testing is a time consuming and costly stage that requires a high degree of ingenuity. In the development stages of safety-critical and dependable computer software such as language compilers and real-time embedded software, testing activities consume about 50% of the project time. In this work we address the area of compiler testing. The aim of compiler testing is to verify that the compiler implementation conforms to its specifications, which is to generate an object code that faithfully corresponds to the language semantic and syntax as specified in the language documentation. A compiler should be carefully verified before its release, since it has to be used by many users. Finding an optimal and complete test suite that can be used in the testing process is often an exhaustive task. Various methods have been proposed for the generation of compiler test cases. Many papers have been published on testing compilers, most of which address classical programming languages. In this paper, we assess and compare various compiler testing techniques with respect to some selected criteria and also propose some new research directions in compiler testing of modem programming languages. © 1997 Elsevier Science B.V.

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

1. Introduction

Testing is the process of executing a program with the intent of finding errors. Testing cannot prove the absence of errors in a software program unless it is exhaustive. If for some test data the software behaves incorrectly, then that software contains errors. However, if the software behaves correctly it shows only that for the particular test data the software works. Program testing is done to establish some degree of confidence that a program does what it is supposed to do and does not do what it is not supposed to do. The basic element in testing is a test case. A test case must consist of two components, a description of the input data to the program and a precise description of the correct output of the program for that set of input data. Test cases must be written for invalid and unexpected, as well as valid and expected, input conditions. In general, testing of software systems is often a time consuming and labor intensive task. One of the main problems in software testing is the impossibility of ensuring a 100% coverage of the system under test, and the infiniteness of the possible test input data. Software testing is discussed extensively in [1,2].

A compiler is a computer program that accepts a source program and produces either compiler error messages or an object code corresponding to the valid source program. Due to the complexity of the compiler as a program, checking the conformity of a compiler to its specifications is a complex task. A compiler is intended to be highly usable and its correctness is critical for the creation and execution of other programs. Therefore, it should be verified carefully before it is released. The generation of an effective set of tests can be a substantial task involving the analysis of a thousand combinations of cases. A compiler test suite contains many test cases. In the context of compiler testing, a test case consists of (i) a test purpose or test case description, (ii) a test input consisting of a source program for which the behavior of the compiler under test is verified, and (iii) an expected output which may include a reference to an output file or error file [2]. In the rest of the paper, we refer to the program part of a compiler test case as simply a test case.

When test cases are executed, they should give clear and unambiguous results. Moreover, test cases should cover all the syntax and semantic rules of the language and generate the errors in all possible contexts. As a result of executing the test case, a test verdict must be obtained, i.e. either the test case passes or fails the test. A test case is said to have passed the testing process if after compiling and executing the test program using the compiler under test, the generated output matches the expected output file, and, if applicable, the error messages match the expected

* Corresponding author.

PII S0950-5849(97)00017-7

error messages as specified in the test case expected output details [3]. On the other hand, a test case is said to have failed if the generated output does not match the expected output. Test case generators can be used to generate test cases for checking the correctness and completeness of compilers. For large scale systems, the automation of the test case generation process is a feasible approach for obtaining an effective and representative test suite.

The rest of the paper is organized as follows. Section 2 presents some issues related to compiler testing theory and practice, Section 3 provides the criteria for the assessment of compiler test case generation techniques, and Section 4 provides a survey of some existing compiler testing techniques. In Section 5 the surveyed papers are assessed and compared on the basis of the given criteria. Finally, in Section 6, we conclude the paper with some remarks on future research directions.

2. Issues in compiler testing

Two test execution strategies are identified, namely dynamic testing and static testing. Dynamic testing consists of executing the program under test using a set of test data controlled by the test environment. The output of the program under test is compared with the expected output as specified in the test case. Whereas static testing consists of inspecting the source code of the program under test without running it. Testing compilers is typically done using a dynamic test execution strategy. The compiler is executed using a preferably certified compiler test suite. The generated output is then compared with the expected output. To generate test data for the dynamic test execution strategy, test generation techniques can be classified under two broad categories: white box-based test generation and black box-based test generation [2].

White box testing techniques are based on the examination of the internal structure of the program to verify that it is executing as intended [1,2]. Using this strategy the tester derives test data after examining the program's logic in addition to some structural control and data flow based techniques. White box testing techniques are based on the availability of the source code for testing purposes. These techniques are used by compiler implementors to test the individual parts of the compiler, to guarantee that each part work correctly. However, black box based test suites are normally used for various testing activities such as performance testing and robustness testing, and for marketing and certification purposes. Test data are derived solely from the specifications. Test cases are not concerned with the program structure, since the focus is on the program's features and its external behavior. Black box testing techniques are used to check the conformity of the compiler functionalities to the language definition and to the user interface, which is the only available information. Certifying the conformance of a language compiler to the language standard definition is an increasingly important issue that can enhance the marketability of the certified compiler.

Exhaustive black box and white box testing are in general impossible, but a reasonable testing strategy may efficiently use elements of both the testing techniques. One can develop a reasonably rigorous test by using certain black box oriented test case design methodologies and then supplementing these test cases with white box oriented methods.

In summary, the different issues involved in compiler testing and in software testing in general are as follows.

- Test case generation strategy. Concerned with the systematic method for generating test cases, i.e. black box or white box test generation strategy. The different methods that come under white box test generation are logic coverage testing, statement coverage, decision coverage, etc., and the methods that come under black box test generation are equivalence partitioning, boundary value analysis, transaction, etc. [2].
- Test selection strategy. Concerned with the selection of a subset of the generated test cases. The selection should be made on the basis of obtaining the minimum and most significant and error revealing test cases from the automatically generated test cases.
- Test execution strategy. Concerned with the selection of a suitable strategy for executing the selected tests. The test set-up and configuration have to be explicitly defined to obtain a realistic test execution environment.
- Test specification language. Concerned with the use of a preferably standard language for the formal description of test cases. Formal description of test cases is necessary to facilitate test maintenance, understanding, execution and analysis.
- Test result analysis. The output obtained after the test execution and the expected output should be compared and analyzed and a test verdict can be obtained. Ideally the diagnosis and the error localization should be automated.
- Test coverage and metrics. The efficiency of the test cases depends on the extent to which the software functionalities be covered. Test cases should cover all the syntax and semantics of the programming language and generate compiling and run-time errors in all possible contexts.
- Test case correctness. Concerned with the verification of the validity of the test case design. Ideally, the test case generation strategy must guarantee the correctness of the generated test cases.

Fig. 1 shows the different stages involved in compiler testing. The first stage is the test case generation and selection. The grammar is used by the test program generator to generate the test programs and the expected output, i.e. the test cases. In the second stage, the test programs are executed in an appropriate test set-up. Here the test programs are given to the compiler under test and the compiler output is obtained. The last stage is the test result analysis. The output obtained from the compiler is compared against the expected output, and the results are analyzed to obtain a test verdict.

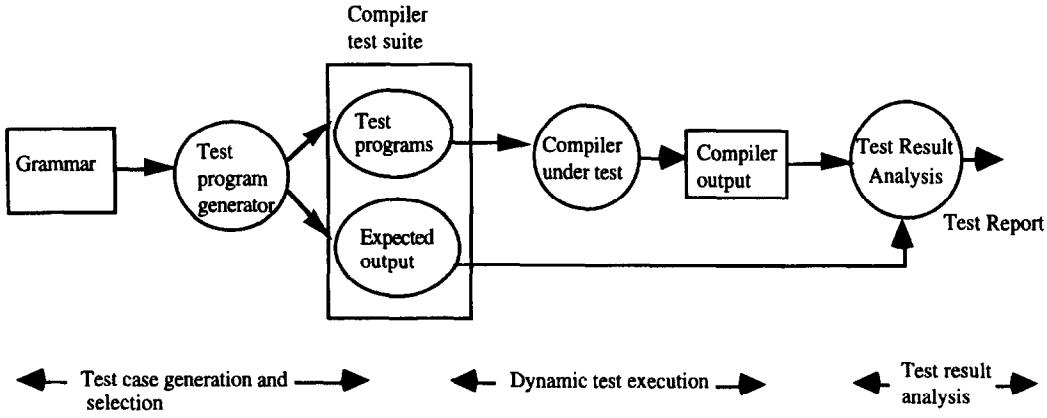

Fig. 1. Different stages of compiler testing.

3. Criteria for assessment

The aim of this paper is to survey and assess the different test generation methods. Various methods for the generation of compiler test cases (i.e. test programs) were introduced in the literature and are discussed in the following section. The assessment of each method is based on some criteria, such as the type of grammar, the data definition coverage, the language syntax and semantic coverage, the ease of automation, the applicability to different types of language, the implementation and efficiency of the technique, the test case correctness and the consideration of advanced programming concepts like concurrency and exception handling.

In the following, we briefly discuss each of the criteria used in our survey to assess the different compiler test case generation methods.

- Type of grammar. The grammar is used to formally describe the language for which a compiler is being tested. A grammar is the main input to the test case generation process. An ideal method would be applicable to a wide variety of grammar types such as context-free grammars, regular grammars, etc.
- Data definition coverage. Since programs are composed of algorithms and data structures, a typical test generation technique should also cover data definitions. Typical programs in languages such as C and Cobol may include a large data definition part. Knowing also that a large source of errors in programs is data definition errors [1].
- Syntax coverage. The generated test cases should cover all the syntax of the programming language for which the compiler is written. For each syntactic definition or rule in the grammar of the language, the method should generate both syntactically correct and incorrect programs.
- Semantic coverage. The generated test cases should cover all the semantics of the programming language. For each semantic definition, the method should generate both semantically correct and incorrect programs. Semantic definitions include type checking, the meaning of looping, selection and other language constructs and features.
- Extent of automation. The method should be easily automated. In particular, testing compiler implementation is one area where automation is desirable.
- Type of language. The method should be applicable to the different types of programming languages, i.e. functional, procedural, etc.
- Implementation and efficiency. The given test generation method must be efficient as well as easily implementable.
- Test case correctness. The technique should be able to generate correct test cases. This is a very crucial requirement since a test case is considered as a reference point in the testing process. An incorrect test case cannot be tolerated. Therefore the method must guarantee that the generated test cases are correct with respect to the language syntax and semantic definitions.
- Concurrency/exception handling. The method should be able to generate test cases to test compilers for concurrent languages such as Ada, Concurrent C, etc. In addition, it should be able to generate test cases for testing exception handling facilities in compilers.

4. Survey of compiler testing

Automatic generation of test cases has been addressed extensively in the software evaluation and testing literature [1,2]. In this section, we briefly survey and assess the main methods introduced for automatic test case generation. The compiler test case generation methods are listed in chronological order below.

In 1970, Hanford [4] used a dynamic grammar to generate test data for a PL/1 compiler. A context-free grammar that can modify itself is called a dynamic grammar. This method produces meaningless but syntactically valid programs. For the compiler, the syntax machine can fully test the ability of

the input phase to accept syntactically valid input. It can be used to check the reliability of the compiler, related to problems such as getting into an infinite loop, abnormally ending or diagnosing non-existent syntax errors. In the syntax machine, the actual program consists of two parts, a grammar loader and a production algorithm. The loader stores the grammar in main storage in a form convenient for use by the production algorithm to create programs. An inexhaustible source of new test material can be obtained due to automation but the only problem is that they are not necessarily semantically correct. The syntax machine produces programs in a random or ad-hoc manner. Hanford's method concentrates on modular and procedural languages like FORTRAN and PL/1.

In 1972, Purdom [5] used a syntax directed method to generate test sentences for a parser. A fast algorithm is given to generate a small set of short sentences (i.e. programs) from a context-free grammar such that each production of the grammar is used at least once. To generate the sentences, two items of information are required for each non-terminal: (i) which production must be used to rewrite the symbol so that the shortest terminal string will result, and (ii) which production must be used to introduce the symbol into the derivation of the shortest sentence that can use the symbol in its derivation. Two algorithms are used to obtain this information. Using the values from the two algorithms the sentence generation algorithm is formed. Though this method may be applied for testing parsers, of both large and small programming languages, it is only complete in the case of a parser for smaller grammars. For some types of grammars, this method is incomplete. While testing the parsers of programming languages, it was found that the sentence generator did quite well in testing the various states, but only moderately well in testing the transitions. Some experimental results from using the sentence generator for some automatically generated simple LR(1) parsers are discussed in the paper. The sentences generated by the parser were quite useful for detecting errors in the grammar of programs.

In 1974, Seaman [6] discussed a program generator, which makes decisions on the coded source constructs on the basis of pseudo-random numbers. The generator keeps track of the values of variables declared in the program it is generating, and generates comparison operation between these variables and the constants that represent the values the variables ought to have. Another approach is running a test program against two compilers and then comparing the results. The program is first executed on the checker with the addition of WRITE statements so that all the variables in the program are written as records. The same program is then executed with the optimizer but with READ statements. After each READ, a comparison is made. An unsuccessful comparison indicates an error. Each decision in the generator is associated with a biasing factor. Examples of the decisions made are the values of constants, storage classes and lengths of strings. By changing biasing constants in the generator, it is possible to concentrate on checking a particular area.

In 1976, Wichmann and Jones [7] discussed subjecting the compiler to a large number of small programs all of which contain some unusual features to exercise rarely executed parts of the compiler code. Two types of tests are considered here. The first type is an exhaustive test that exercises every part of the compiler. The second class of tests is designed to ensure that any size limit is not excessive. Tests are included to check the depth of nested constructs like procedures, blocks, for-loops, etc. An additional test of a similar nature contains a complex expression in every position in the syntax where an expression is permitted. Tests are also included to examine the handling of the features of ALGOL 60. Due to the complications involving language definitions many features are difficult to handle. Moreover, any significant error in the syntax analysis part of a compiler will be detected. Whereas for semantic testing, the result is not satisfactory. The tests were run on four compilers and some characteristics of those compilers were noted.

In 1978, Duncan and Hutchison [8] presented an attributed grammar based method for generating semantically correct test data. The test cases are based on the specifications rather than the program code. The test cases are meant to test how the program handles certain input classes of data rather than to exercise some sections of the code. The attributes provide the context sensitive information necessary to generate semantically correct test cases and help to guide the selection of test cases according to various criteria. Including attributes in the grammar makes explicit how information is passed during the test case generation process. Here the approach is to capture the meaning of specifications in languages by constructing a formal mechanism for generating selected test cases, then it can be evaluated. To extend the power of the attributed grammar, values are allowed to control the choice of rules. Action routines are used to produce an output string corresponding to the generated input string. The creative part of using this system consists of finding appropriate ways of encoding various testing heuristics with algorithms for choosing productions or attributes.

In 1979, Bauer and Finger [9] developed a method that would generate complete test cases for finite state control programs by means of regular grammars. Regular grammar is used to generate test sentences each of which describes a sequence of stimuli to be applied to the system under test and responses required under test. The systems considered here are those which can be specified using an augmented finite state automaton (fsa) model. When a specification that can be modeled by an fsa is given, a set of test sequences is generated to test the system thoroughly. Each test sequence will be a sequence of stimuli and the associated responses. Here the test case generation scheme is implemented in a component called the test plan generator (TPG). Input to the TPG is an augmented fsa that describes the observable

behavior of the system to be tested. The augmented fsa is an fsa with an additional feature — associated with each arc is a condition that must be satisfied for that arc to be traversed. This fsa is generated from a component called a requirements language processor (RLP). The purpose of the TPG is to analyze the functional description produced by the RLP and to produce a set of executable test scripts. The test scripts are executed by another component which verifies that for each script the system responds in the desired manner. Factors such as the number of function states, number of stimulus types, and the cyclic nature of the system specification will affect the number of test sequences generated.

In 1980, Celentano et al. [10] discussed a system for assisting in the testing phase of compiler development. This method partially automates the task of compiler testing. The definition of the language to be compiled drives an automatic sentence generator. The language is described by an extended Backus–Naur Form (EBNF) grammar. The approach here is to check and possibly correct the test programs, so that they can be successfully compiled. The first step is to generate sentences that are correct with respect to the context-free grammar of the language. As a result of this step, the syntax analyzer of the compiler is verified. As a base for the generator discussed, the Purdom's algorithm is adopted to generate a minimal set of sentences covering all BNF rules. In the system discussed, several levels of grammars are allowed, each level refining some rules of a preceding grammar. The refinements are naturally expressed by the rewriting rules formalism, enriched with a parameter passing mechanism. The system has a multilayer structure that allows a gradual approach to language specification. The context dependent constraints are checked and corrected. By this, a syntactically correct program is converted to a compilable one. The whole process of grammar translation and generation is repeated for each successive level of refinement. The generator is augmented by inserting some actions into the context-free rules that check semantic constraints, and in case of violation, correct the partially generated program. As there is no limit to the complexity of actions, one could define any language construct in this way. However some data type as the user defined data types of Pascal, requires considerable effort. Some experience has been gained with a subset of Pascal.

In 1982, an automatic generator for compiler testing was presented by Bazzichi and Spadafora [11]. Compilable programs to evaluate features and performances of different compilers for the same language were generated and incorrect programs were also generated in a controlled way. The compiler is exercised by programs, automatically generated by a test generator. Since the system produces compilable programs, the following parts of the compiler are tested: the lexical analyzer, the syntax analyzer, the semantic analyzer, diagnostic and error handling capabilities, and limiting conditions. The main input to the compiler is the source grammar in the context-free parametric formalism. The characteristic feature of the context-free parametric grammar is that, variables (these are the names of languages, i.e. they denote languages) can be associated with non-terminal symbols; therefore the productions can be thought of as context-free productions, in which some non-terminal symbols have associated parameters, whose range is a language. An algorithm is introduced to generate syntactically correct strings to use all the productions of a context-free parametric grammar in the shortest way. The production that derives a terminal string of minimal length is found. A string is then generated using a stack. Moreover the generation of test programs can be controlled by some user defined directives.

In 1985, Mandl [12] discussed a new method for testing compilers. According to him, in many situations, testing all the combinations for a particular function is not necessary. All potential combinations can be divided into distinct classes by applying some criteria that places two combinations in the same class if testing one combination provides nothing new over the results of testing the other classes. The method proposed here is one that uses the properties of orthogonal Latin squares. For k variables each admitting n values, choose a set of $k-2$ orthogonal $n \times n$ Latin squares and implement that. Instead of the total number n^k of test cases, only n^2 combinations are needed. The method has been used in designing some of the tests in Ada compiler validation capability test suite.

In 1988, Homer and Schooler [13] discussed the testing of a large compiler whose modules communicate through complex graph structured intermediate representation. A test case generator called TGG is used. It accepts context-free grammar as input and builds a program that generates strings of the grammar. This compiler uses the LG (linear graph) system. LG defines a notation for describing such representations and provides tools to support compiler construction. In particular, a human-readable notation, called LGN, is supported by tools that translate between LGN and internal format. The test grammars produce the source language programs and corresponding LGN for the executable statements in the programs. LGN facilitates creation of test inputs and examination of phase outputs. By modifying the input grammar it is easy to generate new tests according to the specification. The main innovation in the TGG is the treatment of attributes. The approach adopted in the TGG is to allow the derivation tree to be used as the primary data structure. The TGG is implemented as a C language preprocessor. This method is said to be useful for producing large tests that stress certain language features.

In 1989, Wichmann and Davies [14] discussed a Pascal program generator. A validation suite is available in Pascal that ensures the syntax and semantics correctness. The PPG has been produced to validate the code generator. It generates self-checking correct Pascal programs when provided with a few parameters. The PPG generates only a subset of Pascal. A machine independent pseudo-random number generator is used to help in getting some degree of

repeatability. PPG works by recursive descent using random numbers to determine the options to be taken. Corresponding to an integer expression, there is a recursive procedure which generates it, which has as a parameter the value to generate. PPG generates code in a single pass fashion noting the value of variables set for subsequent use. The tool is designed to be used in a host–target environment. The host would be used to run the PPG and the tests would be transferred to the target either before or after compilation, depending upon the type of compiler being tested.

In 1990, Maurer [15] discussed data generator generators. One of the data generator that he considers very useful is the data generation language (DGL). It contains a grammar describing the tests and when executed it generates tests according to the grammar. The best way to create a test grammar according to Maurer, is to start with an existing test set and turn it into a grammar. To generate a test DGL looks at the grammar's start symbol. The first step in generating a test is to select an alternative from the test production. DGL scans this alternative from left to right, replacing the non terminals with data. A set of alternatives is chosen at random. Selection rules are provided by the DGL that specify how to choose alternatives and assigns weights to them. There is a probability assigned to each alternative and by changing the probability the selection can be done. Otherwise action routines are added to the alternatives. Systematic tests can also be generated. Moreover, it is easy to generate C structures but not sure about other languages. The style is dependent on the data structure implementation.

In 1991, Denney [16] presented a meta interpreter (an interpreter for a language written in the language itself) for generating test cases from Prolog specifications. It handles problems that arise by using Prolog-based specifications. Obstructions like recursion, evaluable predicates, etc., confront the person who wants to use Prolog. The meta interpreter controls recursion by using a deterministic automaton to monitor its progress through the specification. The automaton is also used to define which paths through the specification are to be used as test cases. So the meta interpreter can compare where it is with where it needs to go. The meta interpreter generates the specification automaton dynamically as it executes the specification by translating Prolog's goal reduction states to automaton states. Equivalence classes provide a basis for good test coverage and avoid wasting time on tests that are essentially equivalent. As test cases are being generated the interpreter checks to see whether their equivalence class has already been used.

In 1992, Liyuan and Guangjun [17] presented an automatic system of generating test cases for a compiler. In the automatic system of generating test cases based on Purdom's algorithm [5], only context-free grammars can be dealt with. Such a limitation severely limits the usefulness of that algorithm. So, in this paper a system that eliminates the above mentioned limitation is presented. The input to the system is a Jovial grammar, and its output is a set of Jovial programs that can be used to test Jovial compiler. The programs are generated in a top-down, step-by-step refinement way. The concept of multi-level is introduced into the system and a series of functional modules is set up to solve the context-sensitive problem in program generating processes. The generating algorithm guarantees that the output of system is a smallest set of shortest Jovial programs. The system provides a function of automatically modifying grammar, so that a set of Jovial programs that have typical syntax errors can be generated. These programs are necessary for the complete testing of Jovial compiler.

In 1993, Kawata et al. [18] discussed a test program generator called TPGEN. It generates executable programs with self-checking code. TPGEN selects production rules randomly or if specified accordingly and generate source text. If a test program is generated by selecting production rules completely at random, variables or functions defined in the declaration portion will not coincide with those used in the execution portion. In TPGEN, system functions are available that make it easy to store and retrieve information concerning declared variables. Test programs generated by TPGEN have self-checking code and if a test program is executed correctly such a program is discarded and only the information about the type of functional test done is stored in the database. Although TPGEN understands the values of all variables, the user must specify the position, where and how the self-checking code should be inserted.

5. Overall assessment

In this section, we provide an overall assessment of the test case generation methods we surveyed with respect to the features listed earlier in Section 3.

- A variety of grammar types were used in the surveyed methods. Hanford [4] used a dynamic grammar, which is a context-free grammar that can modify itself. Context-free grammars were also used by Purdom [5] and Homer and Schooler [13]. Duncan and Hutchison [8] and Kawata et al. [18] used attributed grammar. The attributes provided necessary information to generate semantically correct test programs. Regular grammars derived from the functional specification are also used by Bauer and Finger [9]. Celantano et al. [10] used an extended BNF grammar that can be augmented by actions to ensure contextual congruence. Bazzichi and Spadafora [11] used context-free parametric grammar and Maurer [15] used enhanced context-free grammar.
- The methods by Hanford [4], Bazzichi and Spadafora [11], Purdom [5] and Wichmann and Davies [14] guarantee the complete coverage of syntax. In the paper by Wichman and Jones [7] errors in the syntax analysis part will be detected. However, in the method discussed by Celentano et al. [10], syntactical correctness can be devised in the test programs.
- In Ref. [8], Duncan and Hutchison used attributed grammar to generate semantically correct programs. In Ref.

Table 1

Research paper	Hanford [4]	Purdom [5]	Seaman [6]	Wichman and Jones [7]
Type of grammar	Dynamic grammar	Context-free grammar		
Data definition	Possible	N/A		Not possible
Complete syntax coverage	Guaranteed	Guaranteed	Possible	Guaranteed
Complete semantic coverage	Not guaranteed	Not guaranteed	Possible	Not satisfactory
Extent of automation	Fully automated	Fully automated	Fully automated	Not automated
Application: functional vs. procedural	Procedural	Functional	Functional	Functional
Implementation/efficiency	Implementable, Succesfully used on a number of products	Implementable, more complete for parsers for small grammars	Tests about 20% of the code in a thorough manner	Not so efficient
Test case correctness	Test cases could be used as diagnostic test cases	Test cases havebeen very usefull indetecting errors ingrammar	Test cases generated are self checking	Test cases are not so effective
Concurrency	Not possible	Not possible	Not possible	Not possible

Table 2

Research paper	Bauer and Finger [9]	Celentano et al. [10]	Duncan and Hutchison [8]	Bazzichi and Spadafora [11]
Type of grammar	Regular grammar	Extended BNF grammar	Attributed grammar	Context-free parametric grammar
Data definition	Not possible	Not possible	Possible	
Complete syntax coverage		Guaranteed	Possible	Guaranteed
Complete semantic coverage		Not possible	Guaranteed	Not possible
Extent of automation	Fully automated	Partially automated	Fully automated	Fully automated
Application: functional vs. procedural	Procedural	Claim to be applicable to any language construct	Procedural	Procedural
Implementation/efficiency	Implementable, not tried on a production system	Implementable	Implementable with the help of a good parser	Implementable and found useful in testing some compilers
Test case correctness	The test cases generated cannot be used as the only system tests	Test cases are compilable and complete	Semantically meaningful test cases are generated	Covers all grammatical constructs for source grammar
Concurrency	Not possible	Not possible	Possible	Not possible

Table 3

Research paper	Pesh et al. [19]	Homer and Schooler [13]	Wichmann and Davies [14]	Maurer [15]
Type of grammar		Context-free grammar	Context-free grammar	Enhanced context-free grammar
Data definition	Possible	Possible	Possible	Not possible
Complete syntax coverage		Guaranteed	Possible	
Complete semantic coverage		Guaranteed	Possible	
Extent of automation	Fully automated	Fully automated	Fully automated	Fully automated
Application: functional vs. procedural	Functional	Procedural	Procedural	Functional
Implementation/efficiency	Found to be very useful for testing, maintanace and validation	Implementable and efficient	The program is not complete but will be usefull for compiler writers	Implementable and found to be an effective tool
Test case correctness		Helps in testing during development phase	Self checking programs are generated	
Concurrency	Not possible	Not possible	Possible	Not possible

Table 4

Research paper	Denney [16]	Liyuan and Guangjun [17]	Kawata et al. [18]
Type of grammar		Extended context-free Jovial grammar	Attributed grammar
Data definition	Possible		Possible
Complete syntax coverage	Possible	Possible	Random selection of production rules
Complete semantic coverage			Possible
Extent of automation	Fully automated	Fully automated	Fully automated
Application: functional vs. procedural	Functional	Procedural	Both
Implementation/efficiency	Implementable	Implementable	Implementable assures a specific testing coverage
Test case correctness	N/A	N/A	Test cases have better bug detection characteristics
Concurrency	Not possible	Not possible	Not possible

[14], Wichmann and Davies used a Pascal program generator which has got a validation suite that ensures its semantic correctness. The introduction of attributes helped to deal effectively with the semantics of a test case in the methods by Kawata et al. [18] and Homer and Schooler [13].

- The methods by Hanford [4], Bazzichi and Spadafora [11], Homer and Schooler [13], Kawata et al. [18] and Wichmann and Davies [14] generate test cases for languages in which data definition parts are quite significant.
- Most of the methods are fully automated, but Celentano et al.'s [10] method is partially automated and the method by Wichman and Jones [7] is not automated.
- Regarding the applicability to functional and procedural languages, Celantano et al. [10] and Kawata et al. [18] claim that their methods are applicable to both types of languages. In Celentano et al.'s method, some data types require considerable effort. In the method discussed by Kawata et al. [18], the test programs generated have been used for several language processors.
- Hanford's [4] method was successfully used on a number of products. Purdom's [5] method is complete for parsers generating small grammars. In the method discussed by Wichmann and Jones [7], only a small part of the method is tested thoroughly. The method by Bauer and Finger [9] is not tried on a production system. Duncan and Hutchison's [8] method is implementable with the help of a good parser. Though the method discussed by Wichmann and Davies [14] is not complete, it can be considered useful for compiler writers.
- Correctness of the generated test cases is not mentioned specifically in any of the papers. Test cases can be produced in a way according to the particular requirements of the compiler implementers in the method by Bazzichi and Spadafora [11]. Testing quality of programs generated by Kawata et al.'s method [18] is close to that of those made manually. In Bauer and Finger's method [9], only the input production part of testing is generated. However, in Hanford's method [4], though the generated programs are syntactically valid, they are meaningless.
- The method by Duncan and Hutchison [8] and Mandl [12] was applied to a number of example programs, including programs in concurrent languages and consideration has been given to a concurrent language (i.e. Ada) in the method by Wichmann and Davies [14].

In general, most of the surveyed methods have considered real-life programming languages such as ADA, Fortran and Pascal. They mostly concentrate on the syntactic features and constructs of the language. We feel that none of them have clearly addressed the testing of the most critical, advanced features of modern languages. Also, most methods do not consider whether the compiler under test deals with the data declaration parts of programs properly. Therefore, the surveyed methods are more appropriate for the simplest syntax testing and they need to be extended to deal with complex semantics of languages.

Tables 1–4 summarize the main features of the methods we surveyed and assessed in this paper.

6. Conclusions and future directions

In this paper, we surveyed and assessed various compiler testing methods. The features of these methods were discussed according to some assessment criteria. The availability of development tools, the production time of compilers has been cut considerably. Ideally a compiler must generate correct object code according to the language specification. The development of compilers must rely on a systematic and formal method of testing. We feel that the discussed compiler testing methods are not complete in terms of their test coverage and applicability to complex programming language features and paradigms. A complete test suite covering all the syntactic and semantic aspects of

the language must be generated. In our opinion, more systematic and formal way of covering the language syntax and semantic should be developed. Concurrent programming is becoming increasingly important. Therefore, considerations must be given for testing the concurrency features of concurrent language compilers. Only Duncan and Mandl have considered applying their methods to a number of example programs, including programs in Ada; however, it was not clear whether their approach can handle the above mentioned paradigms. Currently, we are developing a comprehensive compiler test generation method which formalizes, in a systematic and automatic way, the test programs generation process with the assessment criteria in mind.

Acknowledgements

The authors would like to acknowledge the support of this work by the Kuwait University Research Grant EE066.

References

[1] B. Beizer, Software testing techniques, 2nd ed., Van Nostrand Reinhold, New York, 1990.
[2] G. Myers, The Art of Software Testing, Wiley, New York, 1979.
[3] A. Boujarwah, K. Saleh, Compiler test suite: evaluation and use in an automated environment, Information and Software Technology 36 (10) (1994) 607–614.
[4] K.V. Hanford, Automatic generation of test cases, IBM System Journal, (1970) 242–258.
[5] P. Purdom, A Sentence generator for testing parsers, BIT 12 (July) (1972) 366–375.
[6] R.P. Seaman, Testing compilers of high level programming languages, IEEE Comput. Sys and Technol., (1974) 366–375.
[7] B.A. Wichmann, B. Jones, Testing ALGOL 60 compilers, Software-Practice and Experience 6 (1976) 261–270.
[8] A.G. Duncan, J.S. Hutchison, Using attributed grammers to test design and implementations, IEEE Trans. Soft. Eng., (1981) 170–179.
[9] A.J. Bauer, A.B. Finger, Test plan generation using formal grammers, IEEE Trans. Soft. Eng., May (1979) 425–432.
[10] A. Celentano, et al., Compiler testing using a sentence generator, Software-Practice and Experience, 10 (June 1980) 897–918.
[11] F. Bazzicchi, I. Spadafora, An automatic generator for compiler testing, IEEE Trans. Soft. Eng., 8 (4) (1982) 343–353.
[12] R. Mandl, Orthogonal latin squares: an application of experiment design to compiler testing, Communications of the ACM, 28 (10) (1985) 1054–1058.
[13] W. Homer, R. Schooler, Independent testing of compiler phases using a test case generator, Software-Practice and Experience 19 (1) (1988) 53–62.
[14] B.A. Wichmann, M. Davies, Experience with a compiler testing tool, NPL Report DITC, 138/89 (March 1989) 3–22.
[15] M. Maurer, Generating test data with enhanced context-free grammars, IEEE Trans. Soft. Eng., (July 1990) 50–55.
[16] R. Denney, Test-case generation from Prolog based specifications, IEEE Trans. Soft. Eng., (March 1991) 49–57.
[17] J. Liyuan, H. Guangjun, An automatic system of generating test cases for compiler, Journal of Northwestern Polytechnical University 10 (2) (1992) 158–164.
[18] H. Kawata, H. Saijo, C. Shioya, A practical test program generator, FUJITSU Sci. Tech. J. 29 (2) (1993) 128–136.
[19] H. Pesh, P. Schnupp, H. Schaller, A.P. Spirk, Test case generation using Prolog, IEEE Software, (1985) 252–259.