One Approach to Automated Compiler Verification

Vyacheslav A. Bessonov
Department of Software and Computing Systems Mathematical Support
Perm State University
Perm, Russian Federation
E-mail: v.bessonov@hotmail.com

Scientific Advisor:
Lyudmila N. Lyadova
Department of Business Informatics
National Research University Higher School of Economics
Perm, Russian Federation
E-mail: LNLyadova@gmail.com

Abstract. **Most modern software is written in high level languages. The task of translating source code, written in high-level languages, into a representation, which can be executed on a computer system, solves by specialized programs called compilers. Errors in compilers lead to differences between the behavior of modules, resulting from the work of compilers, and behavior, defining the semantics of the original program. Such errors are very difficult to detect and correct, and their presence casts doubt on the quality of the programs generated by a compiler. Obviously, the correctness of the compiler is a strong prerequisite for reliable software created with its help [20]. This paper describes the concept of a system designed to automate the process of testing the major components of any compiler: syntax analyzer and context conditions analyzer (semantic analyzer).**

Keywords – compiler verification, automated testing, syntax analyzers testing, semantic analyzers testing

I. Introduction

All kinds of software verification methods can be divided into two large groups [8]:

1. Static verification methods, including formal methods, methods of static analysis and expertise. Using of such methods implies that the verification of software systems is done "statically", i.e. without execution on a computer system.
2. Dynamic methods that are used to verify the behavior of the program during execution.

The compiler of any language, having practical value, is such a complex system that static verification techniques can be used only for its individual small subsystems. Despite the fact that there are exceptions such as CompCert or πVC, common practice for compiler testing is dynamic verification [20], which involves the following tasks [14]:

1. Test generation (test writing).
2. A verdict on the results of test execution which is performed by the so-called test oracle, which is a procedure for determining the correctness of the system under this test.
3. Assessment of the tests quality which is performed with special test coverage metrics.

Currently, there are two common approaches used to solve these problems:

1. "White box" testing that used to identify all erroneous fragments of specific implementation.
2. "Black box" testing, designed to determine formal specification's degree of compliance.

Model-based testing is a compromise between these two methods. This approach combines the advantages and eliminates the disadvantages of the above methods [20]. The model can be described formally, that allows using it as input for test generation and evaluation of test coverage. At the same time, the model defines the requirements for implementation and therefore it can be used to test the correctness of a particular implementation.

But it is obvious that manual construction and maintenance of the test suite is extremely difficult task. To simplify this task, it is proposed to use one of the main advantages of the model-based testing – the ability to systematically and automatically generate test cases [2]. The existence of a formal description allows automating the process of tests construction, which significantly reduces labor costs, and the systematic nature of testing increases confidence in its results.

Thus, the described above problems of the dynamic compiler verification can be summarized to the following problems [20]:

1. Automation of test construction:
   a. Automation of the test data generation.
   b. Automation of the test results validation (the problem of constructing a test oracle).
2. Definition of the verification process's termination criterion.

This work is licensed under the Creative Commons Attribution License.

In [20] author proposed a verification scheme that designed to solve these problems. Its schematic representation is shown in Fig. 1.

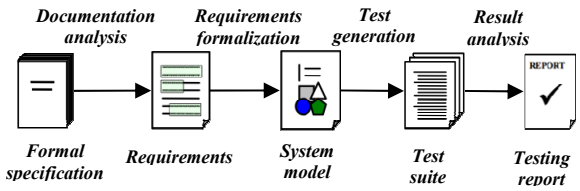

Fig. 1. Verification method scheme

The first stage of the scheme is process of requirements extraction from regulatory documents (e.g., specifications of the target programming language) and its classification. At the second stage a formal model is built via description of extracted requirements in some formal language. At the third stage test generation is performed on basis of the created model. It is often assumed that the user can optionally specify the desired size of the test suite, and/or test suite requirements in terms of some test coverage metrics. Depending on the task in addition to texts in the target programming language test suite may additionally contain an oracle for automatic verdict of the compiler correctness. At the last stage the created test suite is performed. After that reports on the entire process of testing are built. These reports contain information on how the compiler's observed behavior corresponds to the created formal model.

As mentioned above, compilers for real programming languages are extremely complex software systems. Furthermore, there is an additional source of difficulty in verifying compiler. It is the complexity of input data structure and its internal links. Obvious solution to reduce the complexity of the compilers verification task is functional decomposition into separate subtasks that should together cover all the functionality of the compiler [20]. Additional incentive for it is that the compiler is usually represented as a set of functional modules that have strictly defined order of interactions between them.

However, in this paper it is considered verification of only the first three modules: lexical analyzer, syntax analyzer and semantic analyzer. It is worth noting that the development of the lexical analyzer often regarded not as a standalone module, but as an internal infrastructure for syntax analyzer. Under the semantic analyzer in the future will be understood analyzer of static semantics given by the set of so-called context conditions, as an example of which is the enforcement that all used variables should be declared in the program code.

Thus, in accordance to the aforesaid, task of compiler verification may be divided into the following subtasks:

1. Syntax analyzer verification.
2. Semantic analyzer verification.

In the case of automated testing, these tasks can be formulated as follows:

1. Syntax analyzer automated testing.
2. Semantic analyzer automated testing.

II. Syntax Analyzers Automated Testing

The syntax analyzer is one of the core modules of any compiler and its incorrectness makes futile testing the rest of the modules. Therefore, verification of the syntax analyzer is one of the most important tasks of compiler verification.

Positive tests generation

Since the 60's of the 20th century, many authors have investigated the grammar-based test generation for syntax analyzers.

One of the first works in the field was the work of Hanford [6], who proposed a method based on using “dynamic” grammar for generating test data for PL/1 compiler. Its drawbacks are the lack of any coverage metrics and non-deterministic nature of the method.

Purdom’s work [15] considered fundamental. It contains one of the first coverage criteria for positive test sets: in a whole variety of tests for each grammar rule there must be language sentence, which is used in the derivation of this rule. In addition, in the same paper, the author proposed an algorithm for constructing a minimal test set that would satisfy this criterion.

Lämmel [10] showed that the Purdom’s criterion is inadequate: tests that are constructed by this algorithm fail to detect the simplest errors. Stronger criterion proposed by Lämmel avoids this disadvantage and consisted in the fact that the test should cover each pair of rules, one of which can be applied directly after the other.

Many authors ([11], [12], [13]) proposed probabilistic methods of test generation. But in any case, this means that there is no guarantee that the algorithm has finished for the end time and thereby violates one of the basic principles that we have tried to follow, is consistency.

Negative tests generation

The above-described methods devoted exclusively to the generation of positive tests. At this time works, which would have offered methods for generating negative tests, are virtually absent.

A so-called “mutation testing” method is proposed in [7]. The basis of this method is the assumption that after the adding to the original grammar a number of changes (mutations) it can be used to generate potentially negative tests. However this approach entails the following problems:

1. Grammar-mutant can be equivalent to the original grammar.
2. Tests, generated on the basis of grammar-mutant, which is not equivalent to the source, may not be valid.

In [19] authors described methods for generating positive and negative tests and their coverage criteria. The authors embodied developed methods in the tool SynTESK. Using of this tool for testing industrial compilers confirmed the practical applicability of the developed approaches.

SynTESK main advantages are:

1. It is made under a unified methodology UniTESK, which formalizes the process of testing not only syntax analyzers, but also any other software.
2. Mechanisms of its work are based on the formal theory having a clear rationale.
3. It has open nature and is distributed with source code.
4. SynTESK allows to store together with tests their descriptive metadata (for example, the parse tree), which can be used for subsequent analysis.
5. Tool's functionality can be expanded through the development of specialized plugins.
6. The tool has real-world examples of successful application in practice.

But SynTESK has the following disadvantages:

1. SynTESK allows using as a meta-language for the grammar formally describing only one certain type of EBNF. Users who use specialized tools to generate the syntax analyzer (Lex/Flex, SableCC, ANTLR, etc.) will be forced to perform translation from tool's meta-language to SynTESK meta-language.
2. It does not contain any specialized tools for managing sets of tests and their analysis. SynTESK provides no opportunities to work with the generated tests (e.g., edit or delete), and the user is forced to use for this a file system, which greatly complicates the tests processing. In addition, it is often necessary to analyze a set of generated tests (for example, to estimate the coverage metrics or determine the number of tests for a certain grammar rules, etc.), but SynTESK also provides no any special features for this and the user is forced to perform these operations manually.
3. SynTESK does not provide any special features to perform syntax analyzers profiling. For example, changing of string handling internal mechanisms in the syntax analyzer can strongly affect both the value of consumed memory and performance.
4. The tool interprets negative tests as a self-checking. However, apart from establishing the fact of error there must also ensure that the syntax analyzer correctly identifies the type of error and its location. Because application developers will use exactly this information when working with the compiler.

III. Semantic Analyzers Automated Testing

In their works Hanford [6] and Purdom [15] described the methods used to generate a positive tests for the syntax analyzers of procedural languages compilers, but these methods does not take into account any contextual conditions.

In [17] Wichmann and Jones proposed a method for constructing test sets, which would take into account some contextual conditions such as a correct processing of restrictions on the depth of nesting blocks, procedures blocks, cycles, etc. However, this method does not allow checking other simple rules of static semantics, for example, concerning the using of variable names.

Celentano et al in [3] described the practical application of approach, which allows partially automate the testing of Pascal compiler. They used Purdom's algorithm to generate positive tests. To generate the test programs, which correct from the standpoint of static semantics, they used a specialized module with a grammar, augmented with a code for converting syntactically correct programs to semantically correct. The authors noted that the description of the context conditions in this way requires considerable effort and it is unlikely that this approach would be viable for testing modern programming language analyzers.

In [5] authors offer to use attribute grammars as a formalism to describe contextual conditions. The resulting test suite, generated in accordance with the method proposed by Duncan and Hutchison, should contain only syntactically correct tests satisfying the context conditions. This is achieved by sequential scanning of all grammar production rules, which are executed only if it's permitted by contextual conditions. The tests generated by this method, should cover all grammar production rules and all described contextual conditions. However, this approach leads to a large number of empty runs of the generator, because of necessity to interrupt the process of generation due to unfulfilled contextual conditions. Furthermore, this approach leads to the construction of large numbers of semantically uninteresting tests [5].

In [16] Sirer and Bershad described language Lava. Grammar defined on Lava reminds EBNF-grammar augmented by Java code describing the contextual conditions. The authors used Lava to generate a small number of tests (approximately 6 tests) with large size (approximately 60,000 instructions). These tests allowed making some resistance checks of Java Virtual Machine. Unfortunately, the paper does not give any estimates of test coverage.

In [1] author provides a method for constructive description of static semantics, as well as the method of generating both positive and negative tests. In addition, the author proposed a set of coverage criteria. The SemaTESK tool is the practical embodiment of proposed approaches.

SemaTESK as a SynTESK was developed in accordance with the methodology UniTESK and therefore inherits many advantages of this tool. Its other advantages are:

1. The tool uses an algorithm of semantically controlled generation. This algorithm makes it possible to systematically generate test data.

1. The performance of this tool is significantly higher compared to the other instruments (both real and hypothetical) [2]. It is achieved through the use of constructive test generation techniques.

Many SynTESK disadvantages, listed above, are also present in the SemaTESK. Its other disadvantages are:

1. One of necessary steps when working with the tool is the stage of creating a TreeDL representation of AST. However, in the case of using of specialized tools for the generation of syntax analyzers, this representation may be generated by this tool. For example, ANTLR generates a similar representation together with generation of grammar listener or visitor.

2. Users of the tool must create a specialized Java code intended for translation TreeDL representation into the text.

Common SynTESK and SemaTESK problem is that for the user they look like two completely different programs, each of which has its own characteristics and specific sequence of actions. For example, SynTESK user only has to run the program, passing to the input a formal description of the grammar and generation parameters. In the case of tool SemaTESK sequence of actions is much more difficult. In addition to formal description of context conditions user should also create TreeDL representation and develop Java code that performs mapping from TreeDL representation into the text. In the first case, a tester without any programming skills could handle the task of generating. In the second case, the requirements for the qualification of the tool's user are significantly higher.

IV. Another Compiler Testing Suite

Our goal is to develop a system that would combine the advantages of the above-described tools and thus would be deprived of their disadvantages. First of all, the system must meet the following requirements:

1. Unified approach to test generation for syntax and semantic analyzers.

2. Presence of specialized tools designed to manage test sets and to analyze them.

3. Ability for integration with existing development tools using to automate the development process of syntax and semantic analyzers.

The system was called ACTS (Another Compiler Testing Suite) and its schematic representation is shown in Fig. 2.

Fig. 2. Automated testing system scheme

Components of this system are:

1. *Test generator* is the main component of the system. It is designed to automate the process of developing test sets.

2. *Test warehouse* is storage for test suites and their metadata. This component contains special tools for analyzing the warehouse content.

3. *Test runner* is a component, the main purpose of which is to automatically run test suites and collect the results of testing.

A. Test Generator

Test generator should use a unified approach to the generation of tests for both the syntax and semantic analyzers. To implement this requirement, we suggest the following:

1. To use for grammar formal description a meta-language used in some of the most popular tools for generating syntax analyzers (for example, ANTLR).

2. To eliminate the need for intermediate TreeDL representation and use as a representation for the parse tree grammar classes generated by ANTLR tool. This, in turn, saves us from having to write additional code that performs the mapping from TreeDL representation into the text.

Schematic representation of the test generator is shown in Fig. 3.

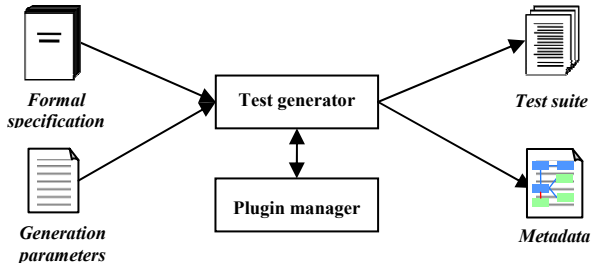

Fig. 3. Test generator scheme

Input data for the test generator are the formal specification of interesting language constructs and user-defined generation parameters. User may specify tests kind (syntax/semantic, positive/negative), test generation method kind, coverage metric, etc.

Currently there are many different methods for generating test data for syntax and semantic analyzers. Many of them are interesting from a practical point of view. That is why the test generator must provide the ability to use different methods of generation.

To implement this requirement, it is proposed to use the plugin-based architecture. Plugin is an abstraction of a method for generating tests and describes a generalized software interface that is used by the generator. Any particular method of generation may be implemented as a separate plugin.

To control the individual plugins it is proposed to use specific module, called “plugin manager”. It allows viewing a list of available plugins, adding new or deleting an existing one. Test generator has access to a specific plugin only through the plugin manager. To select a specific plugin, the user must specify the appropriate information in the list of parameters passed to the input of the generator.

Schematic representation of the plugin manager is shown in Fig. 4.

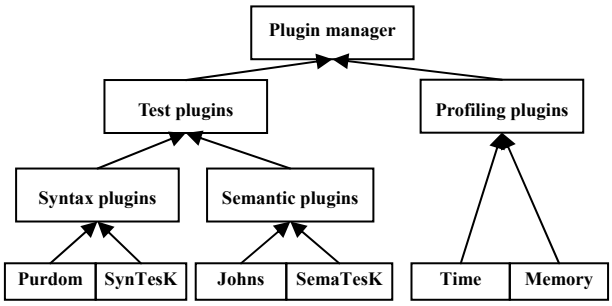

Fig. 4. Test generator plugin manager scheme

In addition to testing compliance of a developed analyzer to a formal specification, ACTS can be used to test analyzers efficiency and productivity. To do this, for example, ACTS can use specialized plugins designed for generation of tests with a very large number of instructions. These tests can be used for analyzers load testing. It is worth noting that these plugins do not have to be a stand-alone product and can use existing plugins for test generation.

The results of the test generator are test suite, which is a set of programs for a particular programming language, and set of metadata representing a formalized description of the test suite.

Such metadata can be extremely diverse. For example, such metadata can be a subset of the Dublin Core properties or the information of the tests structure.

B. Test Warehouse

Test suite and its metadata are placed in test warehouse. Testing reports are also stored in warehouse. Its schematic representation is shown in Fig. 5.

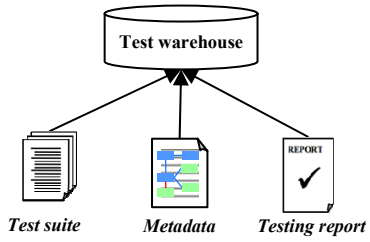

Fig. 5. Test warehouse scheme

In addition to direct physical storage warehouse should provide to the user with a convenient tools to control and analyze its content:

1. Test warehouse should provide a special opportunity to examine the contents of test suites and its metadata. For example, the user may need information on statistical information of existing tests: the number of positive/negative tests, the number of tests for a certain grammar rules, etc.
2. Warehouse must provide the ability to retrieve tests that meet certain criteria (for example, tests that verify the correctness of the implementation of a compiler module).
3. User should be able to view statistical information on the test results: the total number of uncorrected errors, common errors, etc.
4. Test warehouse may need also functions of version control system. At the case of new language development old tests can be an important historical material, showing the path of language development.

To implement this requirement, we propose to use warehouse’s structure, schematically depicted in Fig. 6.

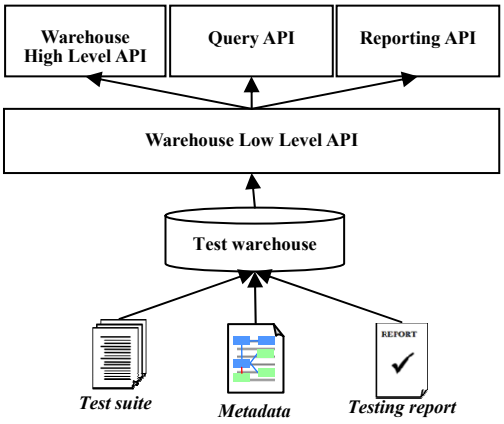

Fig. 6. Test warehouse extended scheme

Warehouse High Level API is a high-level programming interface for managing warehouse content (adding new test or test suite, changing, or deleting an existing one) and for managing its different versions. The main purpose of this programming interface is abstracting from low-level operations like creating new repository, adding new file to repository, committing changes, etc., which would assumed working with specific version control system.

All low-level operations are performed by *Warehouse Low Level API*, which delegates the execution of these operations to a particular version control system. For example, Maven SCM API or specialized software interfaces used in different IDE (e.g., NetBeans VCS API).

Query API is a high-level programming interface for executing queries that retrieve various information from the warehouse (for example, statistical information mentioned above).

Reporting API is a specialized programming interface for reporting. For example, this report is in addition to the

standard information on the number of tests performed successfully or unsuccessfully, may also contain information extracted from the version control system (for example, information about what changes were made in the analyzer source code for a certain time period, by whom they were made and when).

C. Test Runner

Fig. 7 shows a schematic representation of the module running test suites. It is also based on the abstract program interface describing the runner, which can be used to run the tests in any programming language that are stored in the test warehouse.

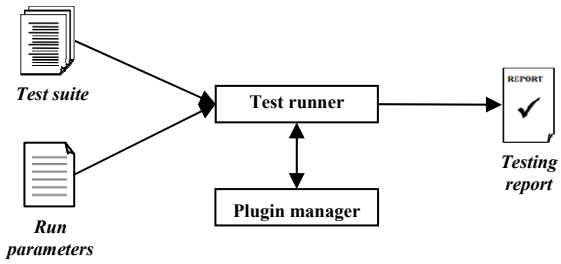

Fig. 7. Test runner scheme

Required possibility of extension, as in the case of the test generator, achieved through the use of plugin-based architecture, where modules designed to run tests on a particular programming language acts as a plugins.

To work with plugins as well as in test generator test runner uses a specialized plugin manager, schematic representation of which is shown in Fig. 8.

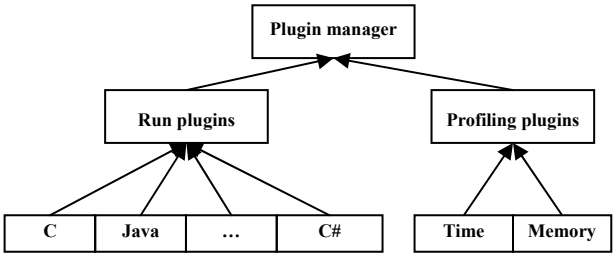

Fig. 8. Test plugin manager

In addition to plugins designed for running test suites and recording the results, ACTS must contain specialized plugins designed to perform profiling analyzers (for example, to determine the number of used RAM or to measure the total execution time).

The result of the test runner is the test report, which contains information on which of the tests have been passed, and which are not, as well as any other information that may be needed for further analysis.

VIII. Integration with Development Instruments

As noted above, currently there are many tools designed to automate the development process of syntax and semantic analyzers: Lex\Flex, Yacc\Bison, SableCC, ANTLR, GOLD Parsing System, etc.

Pretty interesting scenario is the integration of tools that automates the creation of separate compiler modules and tools that automate the process of testing them. In this case, the resulting instrument would almost completely automate the entire process of developing a compiler or its individual modules and greatly facilitate the work of both developers and testers.

For example, in practice, it is not a rare case when one developed language is similar in many ways to others. “Language” at the same time may not necessarily mean a programming language (although in this case there are many examples of similarity of different languages, for example, C# and Java), but the description languages of different data structures, protocols, etc., or DSL languages. For example, the syntax grammar of the new DSL language may be based on the grammar of existing language, which has already been added to the warehouse. Thus the developer can create a new grammar, which includes existing rules and also the tests checking these rules. So with the help of a minimum set of actions developer can build not only a working analyzer, but also a set of tests that can be used to check how well the implementation meets the requirements.

For example, the ease of warehouse integration with different development environments provide a specialized abstraction level *Warehouse High Level API* which allows you to use any version control APIs that exists in modern IDEs (for example, Maven SCM API, NetBeans VCS API, etc.).

Using ANTLR in test generator should ensure ACTS easy integration in such a development environment like ANTLR Works or any ANTLR plugins, existing for other IDEs (IntelliJ IDEA, Eclipse and Visual Studio).

IX. Conclusion

In this paper it is introduced the concept of a system designed to automate the testing of syntax and semantic analyzers. The main advantage of this system compared to existing competing solutions:

1. Unified approach to test generation for syntax and semantic analyzers.
2. Presence of specialized tools designed to manage test sets and to analyze them.
3. Ability for integration with existing development tools using to automate the development process of syntax and semantic analyzers.

Together with instruments designed to automate the creation of separate compiler modules the system could almost completely automate the entire process of developing

a compiler or its individual modules and greatly facilitate the work of both developers and testers.

A deep integration of testing tools and development tools can provide the high quality of the final product.

REFERENCES

[1] Arkhipova M.V. “Avtomaticheskaya generatsiya testov dlya semanticheskikh analizatorov translyatorov”, Dissertatsiya na soiskanie stepeni kandidata fiziko-matematicheskikh nauk. Moscow. 2006. ISP RAS (in Russian).

[2] Arkhipova M.V. “Generatsiya testov dlya semanticheskikh analizatorov”, Vychislitel'nye metody i programmirovanie, Vol. 7, 2006. pp. 55-70 (in Russian).

[3] Celentano A., Reghezzi C.S., Della V.P., Granata G., and Savoretti F., "Compiler Testing using a Sentence Generator," Software - Practice and Experience, Vol. 10, No. 11, 1980. pp. 897-913.

[4] CMMI for Systems Engineering/Software Engineering, Version 1.02 (CMMI-SE/SW, V1.02) CMU/SEI-2000-TR-018 ESC-TR-2000-018. 2000. pp. 598.

[5] Duncan A.G., Hutchinson J.S. Using Attributed Grammars to Test Designs and Implementation // In Proceedings of the 5th international conference on Software engineering. Piscataway, NJ, USA. 1981. pp. 170-178.

[6] Hanford K.V., "Automatic generation of test cases," IBM Systems Journal, Vol. 9, No. 4, 1970. pp. 242 - 257.

[7] Harm J., Lammel R., "Two-dimensional Approximation Coverage," Informatica Journal, Vol. 2029, 2000. pp. 201-216.

[8] Kulyamin V.V., "Integratsiya metodov verifikatsii programmnykh sistem," Programmirovanie, 2009.

[9] Lämmel R., Verhoef C., "Cracking the 500-Language Problem," IEEE Software, Vol. 18, No. 6, 2001. pp. 78-88.

[10] Lämmel R. Grammar Testing // Fundamental Approaches to Software Engineering. 2001. pp. 201-216.

[11] Maurer P.M., "Generating test data with enhanced context-free grammars," IEEE *Software*, Vol. 7, No. 4, 1990. pp. 50 - 55.

[12] Maurer P.M., "The design and implementation of a grammar-based data generator," Software: Practice and Experience, Vol. 22, No. 3, 1992. pp. 223–244.

[13] McKeeman W., "Differential testing for software," Digital Technical Journal, Vol. 10, No. 1, 1998. pp. 101-107.

[14] Posypkin M.A. “Primenenie formal'nykh metodov dlya testirovaniya kompilyatorov”, Trudy Instituta sistemnogo programmirovaniya RAN, 2004 (in Russian).

[15] Purdom P., "A sentence generator for testing parsers," BIT Numerical Mathematics, 1972. pp. 366-375.

[16] Sirer E., Bershad B.N. Using production grammars in software testing // In Proceedings 2nd conference on Domain-specific languages. New York, NY, USA. 1999. pp. 1-13.

[17] Wichmann B.A., Jones B., "Testing ALGOL 60 compilers," Software - Practice and experience, Vol. 6, No. 2, 1976. pp. 261-270.

[18] Yang X., Chen Y., Eide E., and Regehr J. Finding and understanding bugs in C compilers // Proceeding PLDI '11 Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011. pp. 283-294.

[19] Zelenov S.V., Zelenova S.A. “Avtomaticheskaya generatsiya pozitivnykh i negativnykh testov dlya testirovaniya fazy sintaksicheskogo analiza”, Trudy Instituta sistemnogo programmirovaniya RAN, 2004, Vol. 8 (in Russian).

[20] Zelenov S.V., Pakulin N.V. “Verifikatsiya kompilyatorov – sistematicheskij podkhod”, Trudy Instituta sistemnogo programmirovaniya RAN, 2007 (in Russian).