IMPACT: International Journal of Research in Engineering & Technology (IMPACT: IJRET)
ISSN(P): 2347-4599; ISSN(E): 2321-8843
Vol. 5, Issue 5, May 2017, 1-12
© Impact Journals

UTILIZE MODEL APPROACH FOR SELF-TEST CODE AND OPTIMIZING COMPILERS

FAHADGHALIBABDULKADIM[1] & SABAH KHUDAIRABBAS[2]

[1]Assistant Lecture, Faculty of Computer Science and Mathematics, University of Kufa, Iraq

[2]Assistant Lecture, Faculty of Computer Science and Mathematics, Imam Kadhim College, Iraq

ABSTRACT

The paper proposes the concept of automated construction of test suites and test oracles for testing optimizers. It uses an approach based on the generation of test patterns. The main ideas of the model approach are as follows: 1) modeling language implicitly breaks many programs target language into equivalence classes; 2) test coverage criterion is formulated in terms of the model of language; 3) in accordance with the selected set of test criteria is generated. In this paper, we describe a scheme for constructing a test oracle that checks the semantics preservation program after optimization.

KEYWORDS: Implicitly Breaks Many Programs, Terms of The Model of Language

INTRODUCTION

Compilers [1, 2, 3] - is the main tool for creating software, so their reliability is particularly important. Along with other verification and validation Techniques for compilers, testing still remains an important element in the family of these methods. The need to automate testing of compilers also seems obvious, since the real volume of good-quality test kits and the complexity of the analysis results is very high.

Approach Unit Tasks [4] is a methodology for building reliable and quality software based on the use of this software model. Unit Tasks model approach uses the following purposes.

- To Build the Correct Implementation of the Criteria for the Software,
- For Constructing Criteria for Completeness and Effectiveness of Audit Quality Assurance,
- For the Construction of Input Test Data and Analysis Procedures Based on the Results of the Target.

In a narrow sense Unit Tasks proposes to consider the model as a tool for building a test target system. The process of test development and testing itself is divided into the following phases:

- Constructing an abstract model or specification of the behavior of the target system.
- Removing the test oracle (i.e. the target system the result of the analysis procedure) of the specification.
- The decomposition of the input data to the target system domains,
- Design criteria for test coverage in terms of the abstract model.
- Integration of the generated and manually written test system components.

- Skipping tests, including
    - Analysis of the target system using the results of the oracles;
    - Measurement in terms of test coverage models / or specifications in terms of implementation.

The main advantages of this approach are as follows:

- Specifications and models are usually built on the basis of functional requirements for the system, and often BOM structure follows the structure of the requirements, which allows explicitly link the system requirements with the test results and test coverage metrics.
- Specifications and models, and therefore tests can be developed to complete the implementation of the target system, which reduces the overall time of software development.
- Specifications and models are usually more compact and easier to implement, making it easier to re-skid and maintenance of both the models and tests that are based on them.
- Achieving comprehensive coverage in accordance with the criteria set out in the specifications and models, as a rule, provides the implementation level of coverage comparable to the level achieved in the conventional testing, but due to significantly lower labor costs.

Unites approach has been tested in projects with testing of both existing and newly created software [5, 6, 7]. Task force on these projects belongs to different classes of Systems that provide a procedural interface:

- Operating system kernel,
- Telecommunication protocols,
- Servers,
- Run-time support for compilers and debuggers.

Following the general scheme Unites process described above, it was possible to fully automate the work phases 2, 3, 5, 6. Phase 1 is performed manually, phase 4 - semi-automatically.

Transfer of experience and tools for testing compilers Unit Tasks revealed a number of problems. In this article, we describe the application of this approach to testing modules in optimizing compilers.

The main problem here is that there is no effective method to create optimizer such specifications, from which one could extract the effective oracles. Therefore, in the proposed approach, we use only the following Units Tasks phase process.

- Constructing an abstract model of the input data of the target system.
- Design criteria for test coverage in terms of the abstract model.
- Integration of the generated and manually written test system components.
- Skipping tests, including

    - Analysis of the target system using the results of the oracles;
    - Measurement in terms of test coverage models / or specifications in terms of implementation.

Under the proposed approach, Oracle verifies only the preservation of the semantics of the program during optimization. To do this as a test actions are taken on the optimizer program such semantics are fully represented their route. This test allows the property to reduce the problem of semantics checking, saving compared to a reference track with the track.

Thus, the essence of the approach is as follows:

- Build Optimizer to test a representative set of test actions as follows:
    - To Construct an Abstract Model of the Input Data Optimizer;
    - In Terms of an Abstract Model to Formulate A Criterion to Cover These Inputs;
    - To Sort Out the Appropriate Test Actions;
- Optimizer Test as follows:
    - Skip Tests by the Compiler When the Test is Activated, the Optimizer;
    - Verdict on the Preservation of Semantics tests after Optimization.

In the following sections of this article we describe the details of the testing process optimizers in accordance with the proposed approach. At the end of the experimental data on the application of the methodology are discussed the range of applicability and limitations of this approach, and provides an overview of related work.

A preliminary version of this article was reported at the international seminar `` Understanding the programs' [20].

CONSTRUCTING AN ABSTRACT MODEL

The model is based on an abstract description of the optimization algorithm.

The optimization algorithm is formulated using the *terms* denoting the essence of a suitable abstract presentation of the program, such as the control flow graph, data flow graph, the symbol table and so on. The optimizer for its transformation, seeking a combination of entities abstract submission programs that satisfy certain *patterns* (e.g., the presence of program cycles, the presence in the body of the loop structures with specific properties in the presence of common subexpressions procedure, depending on the presence of data between instructions, etc. of some sort.). This may be considered a part of the essence of the terms. To build the model, we will consider only those terms that refer to entities that are involved in at least one template.

So, as a result of algorithm analysis highlighted terms and patterns used in the algorithm. Next, on the basis of this information describes a set of model *building blocks*.

- Each term corresponds to a kind of building block model;
- The building blocks can be linked together to be able to form structures that match the patterns.

Example

Weak-Zero SIV Subscripts analyzer. Consider the analyzer gathers information on any form of dependence of data for subsequent use of this information in different optimizers. Namely, consider the Weak-Zero SIV Subscripts analyzer (see. Eg, [3])

The term *subscript* is used to denote a pair of expressions that are used in a pair of hits in the loop in a single (possibly multi-dimensional) array, and standing in the same position in the index. Subscript called *of SIV* (the single index variable), if the corresponding indexed position is used exactly one index variable. SIV subscript, depending on the induction variable i, is called *weakly zero* (weak-zero), if it has the form $<ai + c_1, c_2>$ where a, c_1, c_2 - constants and $a \neq 0$.

The relationship between the two calls to the array exists if and only if the appeal to the common element enters the loop border. This happens only when the value of

$$i_0 = \frac{c_2 - c_1}{a}$$

It is an integer and $L \leq i_0 \leq U$, where L and U, respectively lower and upper limit cycle.

This algorithm uses the following terms: SIV subscript, is determined by three factors a, c_1 and c_2; cycle determined by its lower bound L and an upper limit of *the U*. The algorithm searches the following template:

$$L \leq \frac{c_2 - c_1}{a} \leq U.$$

Thus the model consists of the following building blocks:

- SIV subscript, contains three attributes that correspond to the values a, c_1 and c_2;
- The cycle containing two attributes that correspond to the values of L and *the U*, as well as many SIV subscripts.

For the special case optimizations, working with such an abstract representation that is close to the syntactic structure of the program, you can use the method of constructing a model based on the idea of reduction grammars (see. [8]).

Example

Control Flow Graph optimizer. Consider the optimizer, which performs the transformation to simplify the procedure of the control flow graph.

The term *linear section* (basic block) represents a sequence of instructions, which begins with the tags may end with a conditional or unconditional jump, and can contain a sequence of non-transition instruction. Linear area called *empty* if it does not contain non-transitional regulations.

The optimizer performs the following transformation:

- if some transition J_1 leads to the label L_1 of a blank of a linear plot that culminates unconditional transition J_2 on the label L_2, then J_1 is transformed into a direct link to the label L_2 ;

- if both branch conditional branch J are on the same label *of* L, then J is transformed into an unconditional jump to the label *of* L ;
- If the label L of a linear section B is not used in any transition, the B udalat.

This algorithm optimization uses the terms: linear plot, conditional jump, unconditional jump. The algorithm searches for the following patterns:

- Transition J_1 leads to the label L_1 of a blank of a linear plot that culminates unconditional transition J_2 on the label L_2;
- Both branches of the conditional jump J are on the same level *of* L;
- The label L is not used in any transition;

This algorithm uses a control flow graph as an abstract representation of the processed program. This idea is closely related to the syntactic structure of the program. Reduction of the grammar of the language allows you to get a model that consists of the following building blocks:

- The procedure comprising a sequence of linear segments;
- The linear portion comprising a label, the transition and the attribute `` empty '';
- The label that contains the attribute `` unused '';
- Unconditional branch containing a reference to a label;
- Conditional jump, containing references to the mark.

We call *the model the structure of* the graph, whose vertices - the building blocks and the edges - the connection between the building blocks.

The projection of the proposals in the original language model structures induces a partition of the source language sentences into equivalence classes. One equivalence class consists of proposals that have the same model representation, ie, that are indistinguishable to the optimization algorithm. This property allows us to put forward the hypothesis that in the equivalent proposals optimizer works equally. Therefore, the desired test set is not enough to have more than one representative from each equivalence class.

Since the set of model structures, ie, the set of equivalence classes, in general, is infinite, to create a test suite, we must choose a finite subset. The reason for this choice should serve as templates that were identified in the analysis of the optimization algorithm. Thus, the test coverage criteria stated in terms of an abstract pattern.

Example

Criterion test coverage analyzer Weak-Zero SIV Subscripts. Recall that the analyzer Weak-Zero SIV Subscripts searches the following template: *of* $L\ i \leq_0 \leq U$ and i_0 unit, where i_0 am determined by the relation

$$i_0 = \frac{c_2 - c_1}{a} \qquad (1)$$

Formulate an appropriate criterion test coverage in terms of the model, i.e., in terms of L, U, a, c_1 and c_2.

Fix L, U and c_1 - integers. Let *take values 1* and *2*. We want to i_0 is an integer of a set, as well as any non-integer values. Suppose that includes the plurality of integers which satisfy one of the following requirements:

- $i_0 < L$, for example, $i_0 = L\text{-}1$;
- $i_0 =$ *of* L ;
- i_0 is located close to L within the range defined by the boundaries of the cycle, for example, $i_0 = L + 1$;
- i_0 is located in the middle of the interval defined by the boundaries of the cycle, for example, $i_0 = \left[\frac{L+U}{2}\right]$;
- i_0 is situated close to the U within the range defined by the boundaries of the cycle, for example, $i_0 = U\text{-}1$;
- $i_0 =$ *the* U ;
- $i_0 > U$, for example, $i_0 = U + 1$.

To find the value of c_2 is sufficient to solve the equation (1) with respect to the values of a and i_0.

THE APPROACH TO THE PROBLEM OF VALIDATION CONSERVATION PROGRAM SEMANTICS DURING OPTIMIZER

When testing an important task it is to analyze the correct operation of the system under test. For the case of an optimizer such analysis consists of two parts:

- check that the semantics of the program has not changed since the optimizer;
- Verification that all were produced optimizes transformation.

In this paper we are not concerned with verifying that all the transformations were made. We consider here only the necessary part of the oracle, namely the preservation of semantics checking program after optimization.

The problem of semantics checking saving any program in the processing of its optimizer is equivalent to the problem of testing the equivalence of two programs. This problem is generally not solvable. However, for certain types of programs, such a problem can be solved. For example, programs which are fully functional semantics seem their route.

Recall that we consider being indistinguishable optimizer programs that correspond to the same model structure. Thus, it is possible as representatives of equivalence classes to select the program, which is fully functional semantics seems route. For such programs, the task of preserving the semantics checking during operation compared to the optimizer reduces the route-optimized software and a reference line. As such a standard we suggest to use the road, issued by non-optimized version of the same program.

Example

How to trace in the test analyzer Weak-Zero SIV Subscripts. Below is an example of the impact test analyzer Weak-Zero SIV Subscripts in the C programming language:

```
01: void f (int i, int * a, int * b, int * c)
02: {
03: for (i = -10; i <= 10; i ++) {
04: a [31] = b [i];
05: c [i] = a [2 + 22 * i];
06:}
07: print_int_1_array (a, 2, 43);
08: print_int_1_array (b, -10, 10);
09: print_int_1_array (c, -10, 10);
10: }
```

Lines 03-06 contain a code, built on the model structure. Lines 07-09 contain instructions for tracing.

To solve the problem of semantics checking saving Optimizer, you must have a lot of tests, and the oracle.

To build a test suite program will generate *the P*, having the following properties:

- numerous programs *P* is representative of the optimizer algorithm test, ie, this set corresponds to the selected criteria of test coverage;
- each *P* is compiled, i.e. syntactically and semantically correct;
- each *P* is completed correctly within a finite time;
- each *P* contains some computation in areas that should be subjected to optimization;
- Functional semantics of each *P* is output, depending on all the available computing program.

Oracle job is as follows:

- Each test is compiled twice - with and without optimization ;
- both compiled version launched for execution;
- produced tracks are compared;
- Semantics is recognized preserved if and only if the equivalent route.

Later in the article we consider in detail the processes of generation and running tests.

CREATING A TEST GENERATOR

We call *test the influence of* individual inputs to test the optimizer, ie program in the target language. The term *co-impact test* represents a single input to the result compilation of test action, ie, parameter values corresponding to run the compiled program. *Iterator to-test actions* - is the component that provides the necessary co-enumeration of test actions and the launch of the compiled test action. Thus, a single *test* includes a test and impact test iterator co-factors.

We need to develop an appropriate generator to produce a plurality of tests for the target optimizer. This generator should generate a representative set of test actions, together with the corresponding iterators to-test actions.

Create generator representative set of test actions starts with an analysis of the test algorithm optimizer build an abstract model and a criterion for test coverage, as described above. After that, the actual development of the generator going.

Test generator consists of two components. The first, called *an iterator* is responsible for the consistent generation of model structures. The second component, called a *mapper*, is responsible for displaying each model structure in the target language.

The iterator should create a set of model structures in accordance with the chosen test coverage criterion.

For a given model structure *S* mapper must construct the corresponding test with the following properties:

- test action, built on the model structure of *the S*, a model representation, which coincides with *the S* ;
- built test (i.e. test action and iterator to-test actions) is syntactically and semantically correct from the point of view of the target language;
- test action comprises computing at locations that should be subjected to optimization;
- test action contains instructions for tracing the final results of calculations;
- iterator to-test actions contain instructions for the formation of all necessary parameters, as well as instructions to activate the compiled test action (ie, the call of the procedure or several procedures).

If the model assumes some calculations, then to form a route mapper inserts the text of the final test action prints the values of all the variables involved in the calculation. Otherwise mapper further inserts the text of the impact test some calculations, and for the formation of the track inserts print the final results of these calculations.

At the end of the iterator and mapper of development they are going into the generator. Thereafter the desired generation test set.

RUNNING TESTS

To check the saving optimizer program semantics of each test is required to compile with optimization, compile and compare the results obtained with the standard route. Recall that we offer as a benchmark to use Tress unoptimized version of the corresponding test.

Thus, the process of running tests and analysis consists of the following steps:

- **Testing SEO:**
    - Compilation of tests enabled the optimization target.
- **By Testing:**
    - launch the compilation of results for performance;
    - Saving the resulting tracks (test track).
- **Reference:**
    - Test compilation with optimization off target;
    - launch the compilation of the results;
    - Saving the resulting tracks (track reference).
- **Running Oracle:**
    - Comparison of the test runs with the standard;
    - Verdict to preserve the semantics during optimization test.

The test is recognized optimizer preserves the semantics of the programs in accordance with the chosen test coverage criterion if and only if Oracle issued a positive verdict for all tests.

The Practical Application of the Approach

Using this approach was built and tested a series of tests in several compilers optimizers for modern architectures: GCC, Open64, Intel C / FORTRAN compiler.

Generators for the following optimizers have been developed:

- Control Flow Graph optimization;
- Common Sub expression Elimination;
- Induction Variable optimization;
- Loop Fusion optimization;
- Loop Data Dependence analysis.

Matching sets of tests generated for C and FORTRAN programming languages.

THE AREA OF APPLICABILITY OF THE APPROACH

The proposed approach is applied to test the imperative programming language compilers. The composition of such compilers tested modules on optimizations and / or analysis.

We believe that the approach is also applicable to inter procedural optimizations, as well as to a wider class of languages, such as functional languages. Adapting the approach to be used in these areas - it is the task of the near future.

RELATED WORK

Formal methods are used to build compilers and theoretical proof of the correctness of their behavior. Verify project [9] contains a theoretical development schemes of constructing reliable compilers based on the application of a series of intermediate languages. Furthermore, there are various attempts to implement reliable compilers using logical calculations [10, 11, 12]. It is also a purely theoretical work. Typically, it offers a simple model compiler, built for him a logical calculus and shows a method of proving the correctness of the proposed compiler.

In [13], proposed the idea of building specifications optimizing transformations using graph redrawing system (Graph Rewrite Systems). The author claims that this technology is applicable to many specifications of the optimization algorithms and program analysis. This approach to specifying transformation, of course, is very interesting and can be used for the construction of oracles. However, the practical use of the system redraw graphs requires great technical support, the creation of which is a separate complex task.

There are also approaches to testing compilers that do not use formal methods. In [14, 15, 16] contains the idea of the implementation of the oracle that checks preserving the semantics of the program during the transformation, in the absence of any specification performed optimizations. It should be noted that a common shortcoming of these approaches is the lack of a method for selecting the input test data. In addition, the approach described in [14] requires intervention in the work of the compiler, which is unacceptable for testing industrial commercial products.

The paper [17] describes the simulation based on the ideas of methodology of automatic construction of tests for the parser of formal languages. As a description of the model used in the BNF-grammar of the source language. In [18] provides a methodology manual creation of test for semantic analyzer model description language programs.

In [19] describes an approach to test automation of semantic analysis and code generation. Specifications for vector and multi-language expressions were developed XASM language. The specifications used for filtering programs generated by relatively simple iterator, and to obtain reference results. Also based on these criteria are generated test coverage tools, and its assessment.

So, testing compilers is a very promising direction in this area there are many interesting studies. However, so far not offered any solution suitable for widespread use in industrial processes testing optimizing compilers.

CONCLUSIONS

To automatically obtain the representative test sets were constructed on the basis of generators abstract description of the optimization algorithm. In the construction of the generators used tools and libraries, which greatly facilitates the process modeling and generator components. Some stages of creation of the generator at the same time have been fully automated.

The advantages of using the model approach are as follows:

- much less labor input than with writing tests manually;
- systematic testing;
- easy maintainability received test sets;

- The possibility of reuse of individual components of the generator.

These benefits are confirmed by the results of a number of projects to test the commercial compilers.

REFERENCES

1. Turing, A. (1950). “Computing machinery and intelligence”, Mind LIX (236): 433-60.
2. Naur, P. and Randell, B. (eds) (1969). “Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE”, Garmisch, Germany, 7th to 11th October 1968, Scientific Affairs Division, NATO.
3. Randell, B. and Buxton, J. (eds) (1970). “Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee”, Rome, Italy, 27-31 Oct. 1969, scientific Affairs Division, NATO.
4. Royce, W. W. (1970). « Managing the development of large software systems: Concepts and techniques”, 1970 WESCON Technical Papers, Vol. 14, Western electronic Show and Convention, Los Angeles, pp. 1-9. Reprinted in Proceedings of the Ninth International Conference on Software Engineering, Pittsburgh, PA, USA, ACM Press, 1989, pp.328-338.
5. Boehm, B. W. (1988). “A spiral model of software development and enhancement”, IEEE Computer 21(5): 61-72.
6. Robillard, P. N. and Robillard, M. P. (1998) “Improving Academic Software Engineering Projects: A Comparative Study of academic and Industry Projects”, Annals of Software Engineering, 6:343-363.
7. Humphrey, W. S. (1998). “Why don’t they practice what we preach?” Annals of Software Engineering, 1(4): 201-222.
8. Cook, S., Ji, H., and Harrison R. (2000), “Software evolution and software evolvability”, Working paper, University of Reading, UK.
9. Denning, P., Comer, D., Gries, D., Mulder, M., Tucker, A., Turner A., and Young, P. (1989), “Computing as a Discipline”, Communications of the ACM, 32(1)J. Hannan, F. Pfenning. Compiler Verification in LF. *Proc. The IEEE Symposium Annual 7Th on Logic's in the Computer Science* (1992) 407-418
10. M. Wand, Zh. Wang. Conditional Lambda-Theories and the Verification of Static Properties of Programs. *Proc. The IEEE Symposium on 5th Logic's in the Computer Science* (1990) 321-332
11. M. Wand. Compiler Correctness for Parallel Languages. *On Functional Programming Conference the Languages and the Computer Architecture (FPCA)* (1995) 120-134
12. U. Assmann. Graph Rewrite Systems for Program Optimization. *ACM Trans. the Languages and Programming on Systems' (TOPLAS)*, **22**, No. 4 (2000) 583-637
13. C. Jaramillo, R. Gupta, ML Soffa. Comparison Checking: An Approach to Avoid Debugging of Optimized Code. *7Th Symposium SIGSOFT the ACM on Foundations of Software Engineering and by European Software Engineering Conference*, LNCS, **1687** (1999) 268-284
14. G. Necula. Translation Validation for an Optimizing Compiler. *Proc. SIGPLAN Conference on the ACM Programming the Language the Design and Implementation* (2000) 83-95

15. TS McNerney. Verifying the Correctness of Compiler Transformations on Basic Blocks using Abstract Interpretation. With In *Symposium on the Partial Evaluation Download now and Semantics-Based Program Manipulation* (1991) 106-115

16. AV Demakov, SA Zelenova, SV Green. Testing parsers texts on formal languages // `` *Software systems and tools: Thematic collection of the Faculty of Computational Mathematics and Cybernetics, Moscow State University ",* Moscow, 2001, Vol. 2, 150-156

17. AK Petrenko and others. Test compiler based on a formal model of language // *Preprint of the Institute of Applied Mathematics. MV Keldysh*, # 45, 1992

18. A Kalinov, A. Kossatchev, M. Posypkin, V. Shishkov. Using ASM Specification for automatic test suite generation for mpC parallel programming language compiler. *Proc. Workshop on International, Fourth the Action the Semantic, AS'2002*, the BRICS note Note the NS-series 02-8 (2002) 99-109

19. AS Kossatchev, AK Petrenko, SV Zelenov, SA Zelenova. Application of Model-Based Approach for Automated Testing of Optimizing Compilers. With In *the Proceedings of the Workshop on International, Program Understanding* (Novosibirsk - Altai Mauntains, Russia), July 14-16, 2003, 81-88 http://www.iis.nsk.su/psi03/workshop/