Model-Based Testing of Optimizing Compilers

Sergey Zelenov and Sophia Zelenova

Institute for System Programming of Russian Academy of Sciences
{zelenov, sophia}@ispras.ru
http://www.unitesk.com

Abstract. We describe a test development method, named OTK[1], that is aimed at optimizing compiler testing. The OTK method is based on constructing a model of optimizer's input data. The method allows developing tests targeted to testing a chosen optimizer. A formal data model is constructed on the basis of an abstract informal description of an algorithm of the optimizer under test. In the paper, we consider in detail the process of analyzing an optimization algorithm and building a formal model. We also consider in outline the other part of the method, test selection and test running. The OTK method has been successfully applied in several case studies, including test development for several different optimizing compilers for modern architectures.

Keywords: model based testing, compiler testing, formalization of requirements, formal data model, test data generation.

1 Introduction

High level programming languages are the main instruments in software development. Translation of source text written in a high level programming language into executable form is performed by software that is traditionally called "compiler".

Compiler defects break execution of entities resulting from translation: their behavior differs from what is specified in the language specification. Defects in executable entities induced by erroneous compiler are hard to detect and find a workaround, thus correctness of executables obtained from an incorrect compiler is always a doubt. Validation and verification of a compiler is an important activity for dissemination of a compiler in industry.

Validation and verification of compilers is always a very complicated. The main source of difficulties is complexity of input and output: the input is a program with a furcated syntax structure and rich set of context constraints imposed by the language specification, the output is an executable in machine or intermediate language and possesses similar or even higher degree of complexity.

The usual way to cope with complications of compiler validation and verification is a decomposition the validation and verification task into several subtasks that in total cover whole functionality of the compiler.

[1] OTK stands for "Optimizer Testing Kit".

A. Petrenko et al. (Eds.): TestCom/FATES 2007, LNCS 4581, pp. 365–377, 2007.

Typical compiler includes the following set of functions:

1. analysis of syntax correctness and parsing of input text;
2. semantic check of input;
3. optimization of the internal representation;
4. generation of the output.

There are many papers concerning validation and verification of the first and second functions of compiler. Papers [7,11,18,24] describe various approaches to validation and verification of parsers. Papers [2,5] describe approaches to validation and verification of semantic checkers.

Nowadays the main function of a compiler is optimization, which allows prodicing faster executable programs. So, the main subtask of the compiler validation and verification is the validation and verification of optimizers.

Papers [6,21,22] describes theoretical studies that use various logical calculi for compiler verification.

Papers [12,8,15,13] contain ideas on creation of oracles that check preservation of program semantics during optimizations. The common shortcoming of these methods is that they do not offer any approach for selection of compiler input data.

Study [9] describes an approach to automation of code generator testing. Specifications developed in XASM language were used for automated filtering tests and obtaining reference results. But this approach to test selection is not systematic and very ineffective.

We use testing [3] based on formal specifications and models [17] as the primary tool for compiler validation and verification.

In this paper we present the OTK method of automated test generation for optimizing compiler testing. The method is based on constructing a model of input data of an optimizer under test. The OTK method allows constructing data models and developing generators of tests targeted to testing a chosen optimizer.

The OTK method consists of the following phases:

1. requirements elicitation;
2. formalization of requirements;
3. automated tests generation;
4. tests execution.

In this paper we zero in on the first and second phases. The third and fourth phases have been described in details in [10].

The remainder of the paper is organized as follows. In Section 2 we describe the OTK method. In Section 3 we present practical applications of the OTK method. In Section 4 the paper is concluded.

2 The OTK Method

The OTK method was developed during joint project of ISP RAS and Intel on testing a set of optimizer units of Intel C++/Fortran compiler in 2001–2003.

Most of compilers perform optimization on some internal representation that is built during parsing and semantics analysis. Straightforward approach to verification of the optimization is to build internal representation of some piece of source code and then optimize it.

The problem is that since internal representation is encapsulated in implementation part of a compiler, then it is very uncertain and therefore tests are difficult to build and are not portable even between different versions of the same compiler. Another problem is that test developers may not have an access to the interface of optimizer units, which are working with internal representation of program code[2].

More practical approach to verification is to use purposely built source code. This approach is easier to implement and is more generic. The OTK method implements this approach.

The OTK method is based on UniTESK approach [4,20] to model-based testing and consists of the following phases.

The first phase is requirements elicitation: analytics study an algorithm of the optimization under test, identify input data requirements and categorize them. The result of the phase is a requirements diagram that contains precisely formulated input data requirements, classified into several groups with established links between them. The diagram is used on the following phases.

The first phase is described in Subsection 2.1.

The second phase is formalization of requirements. Elicited input data requirements get specified using appropriate formal notation. Such specification is called formal data model.

The second phase is described in Subsection 2.2.

The third phase is automated tests generation from the formal data model.

The fourth phase is tests execution that results in test reports that contain information about observed compiler behavior.

The third and fourth phases are described in outline in Subsection 2.3. Details may be found in [10].

Reports analysis, defects identification and corrections is beyond the scope of validation and verification. These issues are not discussed here.

2.1 Process of Analyzing an Optimization Algorithm

The first phase of the OTK method is requirements elicitation. An input data requirements are elicited from an abstract description of the optimization algorithm.

An optimization algorithm is formulated using *entities* of some appropriate abstract representation of an input data, for example, control flow graph, data flow graph, symbol table, etc. In order to perform transformations, an optimizer searches for combinations of entities that match some *patterns*, for example, presence of loops in a routine, presence of some specific statements in the loop, presence of common subexpressions, presence of some specific data dependences between statements. Patterns contains entities significant for the algorithm of the optimization. The goal of this phase is to build a UML-like diagram of these entities.

[2] In the project of ISP RAS and Intel we have no access to the interface of optimizers under test due to Intel security policy. The only information available was that an optimization algorithm operates similar to the one described in certain section of the Muchnick's book [14].

Here we proceed with step-by-step detailed description of the process of analyzing an optimization algorithm.

First, one should represent the text of the algorithm under consideration in "if–then" form.

Next, one should mark all branch conditions in this text, i.e. all parts of the text that are located between "if" and "then" words. These branch conditions are *patterns* that the algorithm deals with.

Next, one should mark all entities in all patterns.

Example: Induction-Variable Optimizations Algorithm. Let us consider the induction-variable (IV) optimizations (see [14]). An *induction variable* is a variable whose successive values form an arithmetic progression over some part of the program, usually a loop. There are three important transformations that apply to induction variables:

- *strength reduction* that replaces expensive operations, such as multiplications and divisions, by less expensive ones, such as additions and subtractions;
- *induction-variable removal*, when we may remove an induction variable that serve no useful purpose in the program;
- *linear-function test replacement*, when a variable is used only in the loop-closing test and may be replaced by another induction variable in that context.

For simplicity we consider only the principal part of the algorithm, identifying induction variables. Fig. 7 in Appendix presents the "if–then" form of this algorithm. Patterns are printed in italic. Entities in the patterns are underlined. ▷

Next, one should write out a list of all marked entities. Besides, one should add to this list a principal entity that is a common context where the algorithm is applied. For each entity in the list, one should create some unique identifier.

Example: List of IV-related Entities. A principal entity for the algorithm presented in Fig. 7 is a *loop body*. The list of all entities with corresponding identifiers is shown in Table 1. ▷

Next, one should write out a list of all marked patterns. For each pattern, one should create its graphical representation (a diagram of the pattern) as follows.

- The diagram should contain all entities that the pattern has.
- An entity in the diagram is presented in the form of boxed identifier that corresponds to the entity.
- If an entity in the pattern has some properties, then these properties should be reflected in the diagram under the box of the entity by the label of the form "`<property_identifier> : <value>`".
- If two entities in the pattern are related to each other in some way, then this relation should be reflected in the diagram as an arrow link between corresponding boxes. An arrow should be labeled by the identifier of the corresponding relation. All links fall into two categories:

Table 1. List of IV-related entities

Entity	Identifier
variable	Var
instruction of the form $i = i + c$ or $i = c + i$	Inc
loop constant	Const
subexpression	Expr
induction variable	IndVar
basic IV	BIV
dependent IV	DIV
temporary dependent IV	TIV
assignment	Asgn
loop body	Loop

  - *aggregation* that means that one entity contains another;
  - *reference* that means that entities are related in some another way.

  Any arrow that corresponds to aggregation is marked by a bullet point in the beginning of the arrow.
- Any relation between two entities has cardinality that is reflected by the following labels near the end of the corresponding arrow:
  - without label – "beginning" entity relates to exactly one "end" entity;
  - "`0..1`" – "beginning" entity relates to 0 or 1 "end" entity;
  - "`0..n`" – "beginning" entity relates to 0 or more "end" entities;
  - "`1..n`" – "beginning" entity relates to 1 or more "end" entities.

Example: Diagrams of IV-related Patterns. The algorithm presented in Fig. 7 provides us with the following list of patterns:

1. a variable i is modified by exactly one instruction of the form $i = i + c$ or $i = c + i$, where c is a loop constant, and the instruction is unconditionally executable;
2. a variable i is modified by two or more instructions of the form $i = i + c_n$ or $i = c_n + i$, where all c_n are loop constants, and all the instructions are unconditionally executable;
3.1. a subexpression has any of the forms $\{i * c, c * i, i + c, c + i, i - c, c - i, -i\}$, where i is a basic IV, c is a loop constant;
3.2.1. a subexpression has any of the forms $\{i * c, c * i, i + c, c + i, i - c, c - i, -i\}$, where i is a dependent IV, c is a loop constant, and the subexpression is located after modification of i;
3.2.2. a subexpression has any of the forms $\{i * c, c * i, i + c, c + i, i - c, c - i, -i\}$, where i is a temporary dependent IV[3], c is a loop constant;
4. a subexpression described in the patterns 3.1, 3.2.1, 3.2.2 is assigned to a variable k, and all assignments to k are unconditionally executable;

[3] Any temporary variable is always defined before use.

5. there are two or more cases described in the pattern 4 of modification of one variable k.

The corresponding diagrams are presented in Fig. 1.

The property `uncond` reflects that corresponding instruction is unconditionally executable, the property `kind` keeps the information about form of a subexpression, the property `afterIV` reflects that a subexpression is located after modification of used induction variable.

Links `iv` in the patterns 3.1 and 3.2.1 are references since one induction variable (basic or dependent) may be used in several different subexpressions. The other links in the patterns are aggregations. ▷

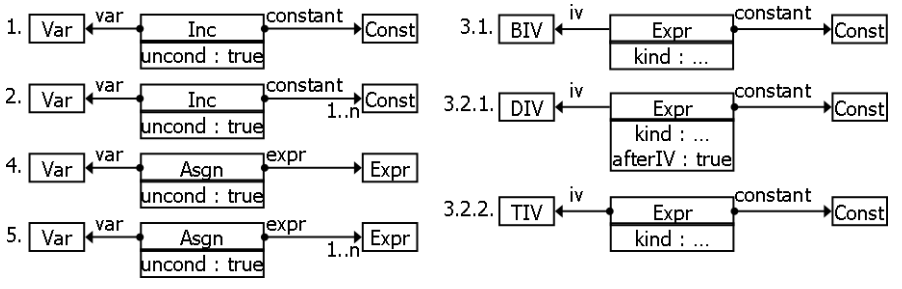

Fig. 1. Diagrams of IV-related patterns

Next, one should improve the diagrams of the patterns, i.e. make the information presented on the patterns more exact: Some entities, links or properties in the diagrams may be renamed or added. The source for such an improvement are those parts of the algorithm that have not been considered yet, i.e. "then" clauses.

Example: Improved Diagrams of IV-related Patterns. "Then" clauses of the items 1 and 2 of the algorithm presented in Fig. 7 say that the variables are in fact basic IVs, "then" clause of the item 3.2.2 says that the temporary dependent IV is related to some subexpression, "then" clauses of the items 4 and 5 say that the variables are in fact dependent IVs.

The corresponding improved diagrams are presented in Fig. 2. ▷

Next, one should check if some entities may be specialized. An entity should be specialized if it has different sets of properties and/or links in the patterns. In this case, the initial entity is called a *generalized entity*.

One should reflect the information about generalization and specialization in a special diagram of generalization. Any entity may occur in the diagram of generalization no more then once. Each specialized entity linked to its generalized entity by a special kind of arrow with big white end. A generalized entity possesses only those properties and links that are common for several entities in the patterns. A specialized entity possesses all properties and links of its generalized entity, and besides, it has some additional properties and links.

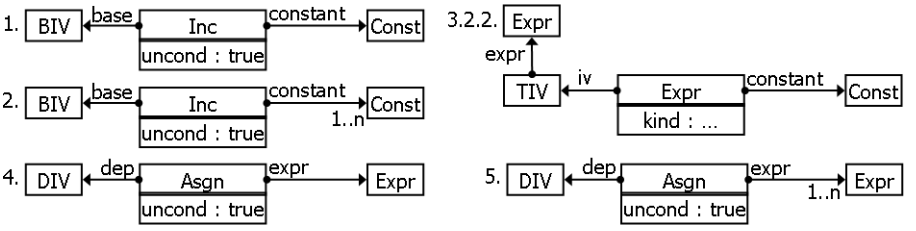

Fig. 2. Improved diagrams of IV-related patterns

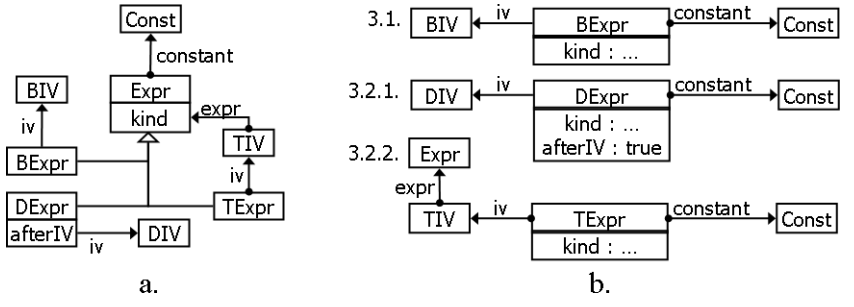

Fig. 3. Generalization of the `Expr` entity

Next, one should improve the initial diagrams of patterns: Rename the generalized entities to corresponding specialized entities.

Note that not all generalized entities can be renamed during such an improvement. If after the improvement some pattern contains a generalized entity, then this entity may be in fact any of its specialized entity.

Example: Generalization of the `Expr` Entity. Occurrences of the `Expr` entity in the patterns 3.1, 3.2.1 and 3.2.2 have different sets of properties and links. Thus, this entity should be specialized. The diagram of generalization is presented in Fig. 3.a.

Now we should improve the diagrams of patterns: We rename the generalized entities `Expr` in the diagrams of the patterns 3.1, 3.2.1 (Fig. 1), and 3.2.2 (Fig. 2) to specialized entities `BExpr`, `DExpr`, and `TExpr` correspondingly. Note that diagrams of the patterns 4 and 5 can not be improved, since these patterns have no information that may be used for specialization of the `Expr` entity.

The improved diagrams of the patterns 3.1, 3.2.1 and 3.2.2 are presented in Fig. 3.b. ▷

Finally, one should construct a UML-like data model diagram. Any entity may occur in the data model diagram no more then once. The data model diagram should contain all entities, properties and links that are presented in all the finally obtained diagrams of the patterns. Besides, the data model diagram contains the principal entity that should be linked to some other entities by means of aggregation links.

Example: IV-related Data Model Diagram. A principal entity for the algorithm presented in Fig. 7 is `Loop`. It may contain several `Inc` entities and several `Asgn` entities.

Fig. 4 shows the corresponding data model diagram for the algorithm under consideration. ▷

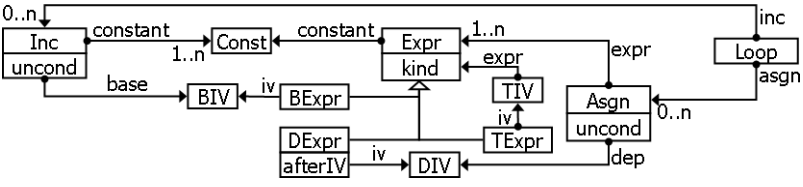

Fig. 4. IV-related data model diagram

The obtained data model diagram is a result of the first phase of the OTK method.

2.2 Formalization of Requirements

The second phase of the OTK method is formalization of requirements. A formal data model is constructed on the basis of the data model diagram elicited on the first phase.

We consider a model representation of a test program as an attributed tree. The role of nodes is played by entities, the role of edges from parents to children is played by aggregation links, the role of attributes is played by properties and reference links.

A formal data model is specified using TreeDL[4] language [19] as follows.

- Each entity is specified using the TreeDL-term "`node`".
- A generalized entity is specified as an "`abstract node`", a specialised entity is specified as a derived node.
- A property of an entity is specified as an "`attribute`" of corresponding node.
- An aggregation link of an entity is specified as a "`child`" of corresponding node.
- An reference link of an entity is specified as a "`attribute late`" of corresponding node.
- The cardinality of properties and links is specified using the following modifiers:
  - without modifiers – exactly one element;
  - "`?`" – 0 or 1 element;
  - "`*`" – 0 or more elements;
  - "`+`" – 1 or more elements.

[4] TreeDL stands for "Tree Description Language".

Example: IV-related Formal Data Model. Fig. 5 demonstrates a formal data model for the algorithm presented in Fig. 7. ▷

```
node Loop : <OtkNode> {
    child Asgn* asgn;
    child Inc* inc;
}
node Inc : <OtkNode> {
    attribute <boolean> uncond;
    child BaseIV base;
    child Const+ constant;
}
node Asgn : <OtkNode> {
    attribute <boolean> uncond;
    child DepIV dep;
    child Expr+ expr;
}
node TIV : <OtkNode> {
    child Expr expr;
}
node BIV : <OtkNode> {
}
node DIV : <OtkNode> {
}
node Const : <OtkNode> {
}
abstract node Expr : <OtkNode> {
    attribute <int> kind;
    child Const constant;
}
node BExpr : Expr {
    attribute late BIV iv;
}
node DExpr : Expr {
    attribute <boolean> afterIV;
    attribute late DIV iv;
}
node TExpr : Expr {
    child TIV iv;
}
```

Fig. 5. IV-related formal data model

2.3 Automated Tests Generation and Tests Execution

Here we proceed with brief description of the third and fourth phases of the OTK method. Detailed description may be found in [10].

The third phase of the OTK method is automated tests generation from the formal data model.

A test coverage criterion is formulated in terms of the data model. A goal of test generation is to cover various combinations of model entities. Tests should contain both combinations that match some of the patterns and combinations that unmatch the patterns in some way. Practice shows that such an approach allows to achieve high level of code coverage of the optimizer under test.

Test program generator is constructed as a structured system of generators of separate data model elements. Such generators in their turn are constructed from generators of subelements, and so on. For example, generator of assignments is usually constructed from two generators of subexpressions and generator of dependent induction variables. All these generators work with model representation of test program structure. The text of test programs appears after applying special mapper component transforming model representation into textual and constructed also on the base of data model structure.

The OTK method is supplied by a tool kit for data model formal description and for developing all required components of a test generator [16].

The fourth phase of the OTK method is automated tests execution.

In the OTK method, an oracle for back-end testing automatically checks preservation of program semantics during back-end pass. To perform this, a mapper should map a model structure to a program with functional semantics being fully described by program's output trace. For such programs, the problem

of checking program semantics preservation during optimizer pass is reduced to comparison of output trace of an optimized program with some reference trace.

Checking optimizer correctness is organized as comparison of traces generated by program compiled with optimization and without it.

Example: IV-related Test Program. Fig. 6 shows an example of a test program generated with OTK from the formal data model presented in Fig. 5. The program consists of one loop with several statements, each of which is modification of some induction variable. Some of the statements are located within if-statements that reflect conditionally executable instructions. The program takes several parameters that are used as induction variables, which are modified by the assignments inside the loop and then are printed in the trace. Traces of optimized and nonoptimized programs' executions with several arrays of parameters are compared to find differences in their behavior. Each difference detected is further analyzed for being caused by a bug in an optimizer unit. ▷

```
void f_0(int i_0, int i_1, int s_0, int s_1) {
    int k;
    for( k = 0; k < 100; k++ ) {
        if( cond_asgn() ) {
            s_0 = i_0 - 7;
            s_0 = 7 - i_1;
        }
        if( cond_asgn() ) {
            s_1 = -i_0;
            s_1 = s_0 * 7;
        }
        if( cond_inc() ) {
            i_0 = i_0 + 7;
            i_0 = i_0 + 7;
        }
        i_1 = i_1 + 7;
        i_1 = i_1 + 7;
    }
    printf( "%d %d %d %d\n", i_0, i_1, s_0, s_1 );
}
```

Fig. 6. Example of generated test program for IV optimization

3 Practical Applications

The OTK method was used in several case studies.

During joint project of ISP and Intel in 2001–2003, the OTK method has been applied in testing several optimizing compilers for modern architectures, namely, in GCC, Open64, Intel C++/Fortran compiler.

Test sets developed with the help of the OTK method and targeted to the following compiler's components have been used for testing:

- Common subexpression elimination;
- Jump optimizations;
- Loop fusion optimization;
- Induction variable optimization;

– Linear loop transformations;
– Loop carried dependence detection;
– Register allocation;
– Loop rerolling;
– Subscripts dependence detection;
– Separable and coupled subscripts detection.

Desctiptions of these components may be found in [14,1].

As a result of test execution, several bugs in compilers under test have been found. In the case of GCC testing, we have achieved about 90% of code coverage of the units under test.

During joint project of ISP and Intel in 2004, the OTK method has been successfully applied in testing exception handling mechanism[5] in Intel C++ compiler.

During joint project of ISP and DaimlerChrysler AG in 2005, the OTK method has been successfully applied in testing optimizers of graphical models [23].

Obtained practical results prove effectiveness of the OTK method.

4 Conclusion

This paper presents the OTK method that implements model-based testing approach to optimizing compiler testing. The OTK method supports test development phases starting on requirements elicitation from an algorithm of the optimization under test and ending on automated tests generation and test execution. The process of analyzing an optimization algorithm and building a formal data model is considered in details.

The OTK method is supplied by a tool kit that supports creating formal data models and developing test generators. A generator developed with the help of the OTK allows automatic generating sets of tests that meet a chosen coverage criteria and are targeted to an optimizer under test.

The OTK may be also used in test development for processors of complex structured text.

The OTK method was used in several case studies including commercial compiler testing projects. Obtained practical results prove effectiveness of the OTK method.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, San Francisco (2002)
2. Arkhipova, M.V.: Semantic analyzer tests generation. Numerical Methods and Programming, vol. 7, pp. 55–70 (in Russian) (2006) http://num-meth.srcc.msu.su/english/zhurnal/tom_2006/v7r206.html

[5] Checking correctness in this case has been organized as comparison of traces generated by program compiled with compiler under test and compiled by GCC.

3. Beizer, B.: Software Testing Techniques. van Nostrand Reinhold (1990)
4. Bourdonov, I.B., Kossatchev, A.S., Kuliamin, V.V., Petrenko, A.K.: UniTesK Test Suite Architecture. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 77–88. Springer, Heidelberg (2002)
5. Duncan, A.G., Hutchison, J.S.: Using Attributed Grammars to Test Designs and Implementation. In: Proceedings of the 5th international conference on Software engineering, Piscataway, NJ, USA, pp. 170–178. IEEE Press, New York (1981)
6. Hannan, J., Pfenning, F.: Compiler Verification in LF. In: Proc. 7th Annual IEEE Symposium on Logic in Computer Science, pp. 407–418 (1992)
7. Harm, J., Lämmel, R.: Two-dimensional Approximation Coverage. Informatica Journal, 24(3) (2000)
8. Jaramillo, C., Gupta, R., Soffa, M.L.: Comparison Checking: An Approach to Avoid Debugging of Optimized Code. In: Nierstrasz, O., Lemoine, M. (eds.) Software Engineering - ESEC/FSE '99. LNCS, vol. 1687, pp. 268–284. Springer, Heidelberg (1999)
9. Kalinov, A., Kossatchev, A., Posypkin, M., Shishkov, V.: Using ASM Specification for automatic test suite generation for mpC parallel programming language compiler. In: Proc. 4th International Workshop on Action Semantic, AS', BRICS note series NS-02-8, pp. 99–109 (2002)
10. Kossatchev, A.S., Petrenko, A.K., Zelenov, S.V., Zelenova, S.A.: Application of Model-Based Approach for Automated Testing of Optimizing Compilers. In: Proceedings of the International Workshop on Program Understanding. Novosibirsk, pp. 81–88 (2003)
11. Lämmel, R.: Grammar testing. In: Proc. of Fundamental Approaches Software Engineering, vol. 2029, pp. 201–216 (2001)
12. McKeeman, W.: Differential testing for software. Digital Technical Journal 10(1), 100–107 (1998)
13. McNerney, T.S.: Verifying the Correctness of Compiler Transformations on Basic Blocks using Abstract Interpretation. In: Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 106–115 (1991)
14. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco (1997)
15. Necula, G.: Translation Validation for an Optimizing Compiler. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 83–95 (2000)
16. OTK: Optimizer Testing Kit. `http://www.unitesk.com/content/category/9/17/35/`
17. Petrenko, A.K.: Specification Based Testing: Towards Practice. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) PSI 2001. LNCS, vol. 2244, pp. 287–300. Springer, Heidelberg (2001)
18. Purdom, P.: A Sentence Generator For Testing Parsers. BIT 2, 336–375 (1972)
19. TreeDL: Tree Description Language. `http://treedl.sourceforge.net/treedl/treedl_en.html`
20. UniTESK Technology Web-site. `http://www.unitesk.com/`
21. Wand, M., Wang, Zh.: Conditional Lambda-Theories and the Verification of Static Properties of Programs. In: Proc. 5th IEEE Symposium on Logic in Computer Science, pp. 321–332 (1990)
22. Wand, M.: Compiler Correctness for Parallel Languages. In: Conference on Functional Programming Languages and Computer Architecture (FPCA), pp. 120–134 (1995)

23. Zelenov, S.V., Silakov, D.V., Petrenko, A.K., Conrad, M., Fey, I.: Automatic Test Generation for Model-Based Code Generators. In: Proc. 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, ISoLA (2006)
24. Zelenov, S., Zelenova, S.: Automated Generation of Positive and Negative Tests for Parsers. In: Grieskamp, W., Weise, C. (eds.) FATES 2005. LNCS, vol. 3997, pp. 187–202. Springer, Heidelberg (2006)

Appendix

Identifying basic IVs.
We sequentially inspect all variables in all instructions in the body of a loop.

1. If *a variable i is modified by exactly one instruction of the form $i = i + c$ or $i = c + i$, where c is a loop constant, and the instruction is unconditionally executable*, then i is a basic IV.
2. If *a variable i is modified by two or more instructions of the form $i = i + c_n$ or $i = c_n + i$, where all c_n are loop constants, and all the instructions are unconditionally executable*, then i is replaced by corresponding quantity of different basic IVs.

Identifying dependent IVs.
We repetitively inspect all subexpressions in all instructions in the body of a loop.

3. If *a subexpression has any of the forms $\{i*c, c*i, i+c, c+i, i-c, c-i, -i\}$, where i is an IV, c is a loop constant*, then in the following cases we define new temporary dependent IV j whose value is equal to the subexpression, and we replace the subexpression by j:
  3.1. if *i is a basic IV*, then j depends on i;
  3.2. if
    3.2.1. *i is a dependent IV* or
    3.2.2. *i is a temporary dependent IV*,
    and the subexpression is located after modification of i in the body of the loop, then j and i depends on the same basic IV.
4. If *a subexpression described in the item 3 is assigned to a variable k, and all assignments to k are unconditionally executable*, then we does not define a temporary IV for the subexpression, but we state that k is a dependent IV.
5. If *there are two or more cases described in the item 3 of modification of one variable k*, then k is replaced by corresponding quantity of different dependent IVs.

Fig. 7. The "if–then" form of the principal part of the IV optimizations algorithm (identifying IV) with marked patterns (printed in italic) and marked entities in the patterns (underlined)