Моделирование и анализ информационных систем. Т. 22, № 6 (2015), с. 773–782
Modeling and Analysis of Information Systems. Vol. 22, No 6 (2015), pp. 773–782

DOI: 10.18255/1818-1015-2015-6-773-782

UDC 519.681.3

Loop Invariants Elimination for Definite Iterations over Unchangeable Data Structures in C Programs

Maryasov I. V.[1], Nepomniaschy V. A.

Received October 26, 2015

The C-program verification is an urgent problem of modern programming. To apply known methods of deductive verification it is necessary to provide loop invariants which might be a challenge in many cases. In this paper we consider the C-light language [18] which is a powerful subset of the ISO C language. To verify C-light programs the two-level approach [19, 20] and the mixed axiomatic semantics method [1, 3, 11] were suggested. At the first stage, we translate [17] the source C-light program into C-kernel one. The C-kernel language [19] is a subset of C-light. The theorem of translation correctness was proved in [10, 11]. The C-kernel has less statements with respect to the C-light, this allows to decrease the number of inference rules of axiomatic semantics during its development. At the second stage of this approach, the verification conditions are generated by applying the rules of mixed axiomatic semantics [10, 11] which could contain several rules for the same program statement. In such cases the inference rules are applied depending on the context. Let us note that application of the mixed axiomatic semantics allows to significantly simplify verification conditions in many cases. This article represents an extension of this approach which includes our verification method for definite iteration over unchangeable data structures without loop exit in C-light programs. The method contains a new inference rule for the deifinite iteration without invariants. This rule was implemented in verification conditions generator. At the proof stage the SMT-solver Z3 [12] is used. An example which illustrates the application of this technique is considered.

The article is published in the authors' wording.

Keywords: C-light, loop invariants, mixed axiomatic semantics, definite iteration, unchangeable data structures, Z3, specification, verification, Hoare logic

For citation: Maryasov I. V., Nepomniaschy V. A., "Loop Invariants Elimination for Definite Iterations over Unchangeable Data Structures in C Programs", *Modeling and Analysis of Information Systems*, **22**:6 (2015), 773–782.

On the authors:
Maryasov Ilya Vladimirovich, orcid.org/0000-0002-2497-6484, PhD,
A.P. Ershov Institute of Informatics Systems SB RAS
Akademik Lavrentiev pr., 6, Novosibirsk, 630090, Russia,
e-mail: ivm@iis.nsk.su

Nepomniaschy Valery Aleksandrovich, orcid.org/0000-0003-1364-5281, PhD,
A.P. Ershov Institute of Informatics Systems SB RAS
Akademik Lavrentiev pr., 6, Novosibirsk, 630090, Russia,
e-mail: vnep@iis.nsk.su

Acknowledgments:
[1]This research is partially supported by RFBR grant 15-01-05974.

Introduction

C program verification is an urgent problem at the present time. Many projects (for example [4, 5, 6, 8, 9]) suggest different solutions. But none of them contains any methods for loop verification. As it is known, in order to verify loops we need invariants whose construction is a challenge. So the user has to devise these invariants. In many cases it is a difficult task.

In this paper we suggest a method of loop invariants elimination for definite iteration of special form [14]. We extend our mixed axiomatic semantics of C-light language by a new rule which allows verification of such loops without invariants provided by user.

C-light language [18] is a powerful subset of the C language. To verify C-light programs the two-level approach [19, 20] and the mixed axiomatic semantics method [1, 11] were suggested.

On the first stage, we translate [17] the source C-light program into C-kernel one. C-kernel language [19] is a subset of C-light. On the second stage, the verification conditions are generated by applying the rules of mixed axiomatic semantics [10, 11]. The word "mixed" means that it can be several inference rules for the same program construction which are unambiguously applied depending on its context. In many cases the use of specialized inference rules allows us to simplify verification conditions.

All our methods have theoretical justification. Theorems of correctness of translation of C-light into C-kernel and soundness of the C-kernel axiomatic semantics are proven in [17, 10].

At the proof stage, the automatic theorem prover Z3 [12] is used. Extra axioms can be provided by the user in case the prover has failed to check whether a verification condition is true. If all verification conditions have been proven, then the program is partially correct. Otherwise, the user has to modify the program or its specification and to repeat the verification process in C-light verification system [11, 16].

1. Definite Iteration over Unchangeable Data Structures and Replacement Operation

The method of loop invariants elimination for definite iteration was suggested in [14]. It includes four cases [13, 15]:

1. Definite iteration over unchangeable data structures without loop exit.
2. Definite iteration over unchangeable data structures with loop exit.
3. Definite iteration over changeable data structures possibly with loop exit.
4. Definite iteration over hierarchical data structures possibly with loop exit.

This paper deals with the first case.

Let us remind the notion of data structures which contain a finite number of elements. Let $memb(S)$ be the multiset of elements of the structure S and $|memb(S)|$ be the power of the multiset $memb(S)$. For the structure S the following operations are defined:

1. $empty(S) = true$ iff $|memb(S)| = 0$.

2. $choo(S)$ returns an element of $memb(S)$ if $\neg empty(S)$.

3. $rest(S) = S'$, where S' is a structure of the type of S and $memb(S') = memb(S) \setminus \{choo(S)\}$ if $\neg empty(S)$.

Sets, sequences, lists, strings, arrays, files and trees are typical examples of the data structures.

Let $\neg empty(S)$, then $vec(S) = [s_1, s_2, \ldots, s_n]$ where $memb(S) = \{s_1, s_2, \ldots, s_n\}$ and $s_i = choo(rest^{i-1}(S))$ for $i = 1, 2, \ldots, n$.

$last(S)$ is a partial function such that $last(S) = s_n$.

A function $head(S)$ returns a structure such that $vec(head(S)) = [s_1, s_2, \ldots, s_{n-1}]$ if $\neg empty(S)$.

Let S_1 and S_2 be structures. Then we can define a concatenation operation $con(S_1, S_2)$ as follows:

1. $con(S_1, S_2) = S_2$ if $empty(S_1)$.

2. $choo(con(S_1, S_2)) = choo(S_1)$ and $rest(con(S_1, S_2)) = con(rest(S_1), S_2)$ if $\neg empty(S_1)$.

Consider the statement

$$\textbf{for x in S do v} := \textbf{body(v, x) end}$$

where S is a structure, x is the variable of the type of S element, v is the vector of loop variables which does not contain x and *body* represents the loop body computation, does not modify x and S and terminates for each $x \in memb(S)$. The loop body can contain only the assignment statements and the if statements, possibly nested.

The operational semantics of such statement is defined as follows. Let v_0 be the vector of initial values of variables from v. If $empty(S)$ then the result of the iteration $v = v_0$. Otherwise, if $vec(S) = [s_1, s_2, \ldots, s_n]$, then the loop body iterates sequentially for x taking the values $s_1, s_2, \ldots, s_n$.

To express the effect of the iteration let us define a replacement operation

$$rep(v, S, body) = v_n,$$

where $v_0 = v$ if $empty(S)$, $v_i = body(v_{i-1}, s_i)$ for all $i = 1, 2, \ldots, n$ if $\neg empty(S)$.

A number of theorems which express important properties of the replacement operation was proved in [13, 14, 15]. Let us mention the most important of them.

Theorem 1. $rep(v, con(S_1, S_2), body) = rep(rep(v, S_1, body), S_2, body)$.

Theorem 2. $\neg empty(S) \Rightarrow rep(v, S, body) = body(rep(v, head(S), body), last(S))$.

2. The Inference Rule and Its Implementation

There is no for statement in C-kernel. The loop $\textbf{for}\,(\mathbf{e_1}; \mathbf{e_2}; \mathbf{e_3})\,\mathbf{B}$; is translated first into the *while* statement $\mathbf{e_1}; \textbf{while}\,(\mathbf{e_2})\,\{\mathbf{B}; \mathbf{e3};\}$; and then the common inference rule of the mixed axiomatic semantics is applied:

$$\frac{\begin{array}{c}E, SP \vdash \{P\}\,\mathbf{e_1};\{INV\} \\ E, SP \vdash \{INV \wedge cast(val(e_2, MD), type(e_2), int) \neq 0\}\,\mathbf{B};\mathbf{e_3};\{INV\} \\ E, SP \vdash \{INV \wedge cast(val(e_2, MD), type(e_2), int) = 0\}\,\mathbf{A};\{Q\}\end{array}}{E, SP \vdash \{P\}\,\mathbf{e_1};\{INV\}\,\mathbf{while}\ (\mathbf{e_2})\ \{\mathbf{B};\mathbf{e_3}\}\,\mathbf{A};\{Q\}}$$

Here P is precondition, Q is postcondition, INV stands for loop invariant, $\mathbf{A}$ are program statements after the loop.

E is the environment [10] which contains an information about current function (its identifier, type and body) which is verified, an information about current block and label identifier if **goto** statement occurred earlier.

SP is program specification which includes all preconditions, postconditions and invariants of loops and labeled statements.

The function *cast* performs type casting according to ISO C standard, the function *val* calculates the value of the expression e_2, the function *type* returns the type of e.

The meta-variable MD defines the values stored in memory [1, 10].

Now we can introduce the special inference rule for definite iterations:

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v')\ \wedge v = rep(v', S, body)\}\,\mathbf{A};\{Q\}}{E, SP \vdash \{P\}\,\mathbf{for\ x\ in\ S\ do\ v} := \mathbf{body(v, x)\ end\ A};\{Q\}}$$

We use forward tracing: we move from the program beginning to its end and eliminate the leftmost operator (on the top level) applying the corresponding rule. The correctness of this rule can be proved by modification of the proof for backward tracing from [14].

The implementation of this rule extends our verification conditions generator which is based on mixed axiomatic semantics of C-kernel [1, 10].

Note that the common rule adds at least two verification conditions and the rule for definite iteration does not increase the number of verification conditions for a program.

In C-light there is no such statement as **for x in S do v** := **body(v, x) end**. So in fact the generator of verification conditions must be able to determine x, S, v and *body* in a loop of a form **for** $(\mathbf{e_1}; \mathbf{e_2}; \mathbf{e_3})\,\mathbf{B}$;.

Depending on the data structure S we have to introduce several inference rules for each case. For example if S is a subset of integers we have the following rule:

$$\frac{E, SP \vdash \{\exists v'\ P(v \leftarrow v')\ \wedge v = rep(v', (j, j + c, j + 2c, \ldots), body)\}\,\mathbf{A};\{Q\}}{E, SP \vdash \{P\}\,\mathbf{for}\ (\mathbf{i} = \mathbf{j}; \mathbf{i} < \mathbf{k}; \mathbf{i} = \mathbf{i} + \mathbf{c};)\ \mathbf{v} = \mathbf{body(v, i)};\ \mathbf{A}\{Q\}}$$

Here i, j, k, c are integers. In the case when $i \geq k$ or $i = i - c$ the rule looks similarly.

Every time when there is no invariant provided by the user before the for statement the generator of verification conditions tries to apply one of the inference rules suggested by our method. When it fails an error is raised and the user has to provide an invariant himself.

3. Example

To demonstrate the application of our method of loop invariants elimination let us consider the following program. It iterates over an array of integers and for given integer computes the number of entries to this array.

The annotated (in SMT-LIB v2 syntax of Z3) C-kernel program has the form:

```
/* (assert (> length 0)) */
int count(int key, int* arr, int length)
{
    auto int result = 0;
    for (i = 0; i < length; i = i + 1)
         if (arr[i] == key) result = result + 1;
    return result;
}
/* (assert (= result (COUNT key a 0 (- length 1)))) */
```

The function $COUNT$ returns the number of entries of key to arr from $arr[j]$ to $arr[k]$. It is defined recursively as follows:

```
(declare-fun COUNT (Int (Array Int Int) Int Int) Int)

(assert
       (and
           (forall ((j Int) (k Int))
                  (implies
                         (> j k)
                         (= (COUNT key arr j k) 0)
                  )
              )
           (forall ((j Int) (k Int))
                  (implies
                         (and
                             (= j k)
                             (= (select arr k) key)
                         )
                         (= (COUNT key arr j k) 1)
                  )
           )
           (forall ((j Int) (k Int))
                  (implies
                         (and
                             (= j k)
                             (not
                                 (= (select arr k) key)
                             )
                         )
                         (= (COUNT key arr j k) 0)
                  )
           )
           (forall ((j Int) (k Int))
                  (implies
                         (and
                             (< j k)
```

```
                                (= (select arr k) key)
                            )
                            (=
                              (COUNT key arr j k)
                              (+ (COUNT key arr j (- k 1)) 1)
                            )
                        )
                )
                (forall ((j Int) (k Int))
                    (implies
                            (and
                                (< j k)
                                (not (= (select arr k) key))
                            )
                            (=
                              (COUNT key arr j k)
                              (COUNT key arr j (- k 1))
                            )
                    )
                )
        )
)
```

The first conjunct describes the case when the second bound is greater then the first bound. The second and the third conjuncts defines the behavior of $COUNT$ when the length of arr is equal to 1. The fourth conjunct increases the value of $COUNT$ by 1 in the case when the entry of key was found at the index k. Otherwise $COUNT$ is equal to $COUNT$ from j to $k-1$ as it is defined by the last conjunct.

Z3 is the SMT-solver but we are interested in verification conditions validity, not satisfiability. So the verification conditions generator produces the negation of the verification condition:

```
(assert
    (not (forall ((key Int) (arr (Array Int Int)) (length Int))
                (implies
                        (exists ((result!1 Int))
                                (and
                                    (> length 0)
                                    (= result!1 0)
                                    (= result (rep result arr length))
                                )
                        )
                        (= result (COUNT key arr 0 (- length 1)))
                )
         )
    )
)
```

And then we expect the answer "unsat" which means that the negation is unsatisfiable so the verification condition is true.

Also the generator produces the recursive definition of the *rep* function for this program:

```
(declare-fun rep (Int (Array Int Int) Int) Int)

(assert (and (forall ((i Int))
                  (implies
                          (< i 0)
                          (= (rep result arr i) 0)
                  )
             )
           (forall ((i Int))
                  (implies
                          (= i 0)
                          (= (rep result arr i) 0)
                  )
           )
           (forall ((i Int))
                  (implies
                          (and
                              (< 0 i)
                              (= (select arr (- i 1)) key)
                          )
                          (=
                            (rep result arr i)
                            (+ (rep result arr (- i 1)) 1)
                          )
                  )
           )
           (forall ((i Int))
                  (implies
                          (and
                              (< 0 i)
                              (not
                                  (= (select arr (- i 1)) key)
                              )
                          )
                          (=
                            (rep result arr i)
                            (rep result arr (- i 1))
                          )
                  )
           )
        )
)
```

Unfortunately Z3 does not support proofs by induction. In this example it goes into infinite loop without any answer. During our experiment we substituted constants for *length* in the verification condition, and it turned out that for example for $length = 15$ it took 70 seconds for Z3 to provide the desired answer "unsat" running on AMD Athlon II X2 245 processor at 2.9 GHz with 4 gigabytes of RAM.

4. Conclusion

This paper represents an extension of the system for C-light program verification. In the case of definite iteration over unchangeable data structures without loop exit this extension allows to generate verification conditions without loop invariants.

This generation is based on the new inference rule for the C-light for statement which introduces the replacement operation. It expresses definite iteration in special form.

K. Rustan M. Leino suggested a rewriting strategy and a heuristic for when to apply it to verify simple inductive theorems [7]. We plan to use this tactic in our generator of verification conditions.

The next step will be the case of loop invariants elimination for changeable data structures possibly with loop exit.

References

[1] I. S. Anureev, I. V. Maryasov, V. A. Nepomniaschy, "C-programs Verification Based on Mixed Axiomatic Semantics", *Automatic Control and Computer Sciences*, **45**:7 (2011), 485–500, http://link.springer.com/article/10.3103/S0146411611070029.

[2] I. Anureev, I. Maryasov, V. Nepomniaschy, "Revised Mixed Axiomatic Semantics Method of C Program Verification", *Proceedings*, Third Workshop "Program Semantics, Specification and Verification: Theory and Applications", PSSV 2012 (Nizhni Novgorod, Russia, July 1–2), eds. Valery Nepomniaschy, Valery Sokolov, 2012, 16–23.

[3] I. S. Anureev, I. V. Maryasov, V. A. Nepomniaschy, "Two-level Mixed Verification Method of C-light Programs in Terms of Safety Logic", Computer Science, **34**, NCC Publisher, 2012, 23–42.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. Rustan M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs", *Formal Methods for Components and Objects*, 4th International Symposium, FMCO 2005 (Amsterdam, The Netherlands, November 1-4), Lecture Notes in Computer Science, **4111**, Springer, 2006, 364–387, http://link.springer.com/chapter/10.1007/11804192_17.

[5] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies, "VCC: A Practical System for Verifying Concurrent C", *Theorem Proving in Higher Order Logics*, 22nd International Conference, TPHOLs 2009 (Munich, Germany, August 17–20), Lecture Notes in Computer Science, **5674**, Springer, 2009, 23–42, http://link.springer.com/chapter/10.1007/978-3-642-03359-9_2.

[6] J.-C. Filliâtre, C. Marché, "Multi-prover Verification of C Programs", *Formal Methods and Software Engineering*, 6th International Conference on Formal Engineering Methods, ICFEM 2004 (Seattle, WA, USA, November 8–12), Lecture Notes in Computer Science, **3308**, Springer, 2004, 15–29, http://link.springer.com/chapter/10.1007/978-3-540-30482-1_10.

[7] K. Rustan M. Leino, "Automating Induction with an SMT Solver", *Verification, Model Checking, and Abstract Interpretation*, 13th International Conference, VMCAI 2012 (Philadelphia, PA, USA, January 22–24), Lecture Notes in Computer Science, **7148**, Springer, 2012, 315–331, http://link.springer.com/chapter/10.1007/978-3-642-27940-9_21.

[8] K. Rustan M. Leino, "Dafny: An Automatic Program Verifier for Functional Correctness", *Logic for Programming, Artificial Intelligence, and Reasoning*, 16th International Conference, LPAR-16 (Dakar, Senegal, April 25–May 1), Lecture Notes in Computer Science, **6355**, Springer, 2010, 348–370, http://link.springer.com/chapter/10.1007/978-3-642-17511-4_20.

[9] X. Leroy, "Formal Verification of a Realistic Compiler", *Communications of the ACM*, **52**:7 (2009), 107–115.

[10] I. V. Maryasov, *The Mixed Axiomatic Semantics Method*, Novosibirsk, Siberian Division of the Russian Academy of Sciences, A. P. Ershov Institute of Informatics Systems, **160**, 2011, http://www.iis.nsk.su/files/preprints/160.pdf.

[11] I. V. Maryasov, V. A. Nepomniaschy, A. V. Promsky, D. A. Kondratyev, "Automatic C Program Verification Based on Mixed Axiomatic Semantics", *Automatic Control and Computer Sciences*, **48**:7 (2014), 407–414, http://link.springer.com/article/10.3103/S0146411614070141.

[12] L. de Moura, N. Bjørner, "Z3: An Efficient SMT Solver", *Tools and Algorithms for the Construction and Analysis of Systems*, 14th International Conference, TACAS 2008 (Budapest, Hungary, March 29–April 6), Lecture Notes in Computer Science, **4963**, Springer, 2008, 337–340, http://link.springer.com/chapter/10.1007/978-3-540-78800-3_24.

[13] V. A. Nepomniaschy, "Symbolic Verification Method for Definite Iteration over Altered Data Structures", *Programming and Computer Software*, **1** (2005), 1–12.

[14] V. A. Nepomniaschy, "Verification of Definite Iteration over Hierarchical Data structures", *Fundamental Approaches to Software Engineering*, Second International Conference, FASE'99 (Amsterdam, The Netherlands, March 22–28), Lecture Notes in Computer Science, **1577**, Springer, 1999, 176–187, http://link.springer.com/chapter/10.1007/978-3-540-49020-3_12.

[15] V. A. Nepomniaschy, "Verification of Definite Iteration over Tuples of Data Structures", *Programming and Computer Software*, **1** (2002), 1–10.

[16] V. A. Nepomniaschy, I. S. Anureev, M. M. Atuchin, I. V. Maryasov, A. A. Petrov, A. V. Promsky, "C Program Verification in SPECTRUM Multilanguage System", *Automatic Control and Computer Sciences*, **45**:7 (2011), 413–420, http://link.springer.com/article/10.3103/S014641161107011X.

[17] V. A. Nepomniaschy, I. S. Anureev, I. N. Mikhaylov, A. V. Promsky, *Towards The Verification of C Programs. Part 3. Translation From C-light To C-light-kernel and Its Formal Proof*, Novosibirsk, Siberian Division of the Russian Academy of Sciences, A. P. Ershov Institute of Informatics Systems, **97**, 2002 (Russian), http://www.iis.nsk.su/files/preprints/097.pdf.

[18] V. A. Nepomniaschy, I. S. Anureev, A. V. Promsky, "Towards Verification of C Programs. C-light Language and Its Formal Semantics", *Programming and Computer Software*, **28** (2002), 314–323.

[19] V. A. Nepomniaschy, I. S. Anureev, A. V. Promsky, "Towards Verification of C Programs. Axiomatic Semantics of The C-kernel Language", *Programming and Computer Software*, **29** (2003), 338–350.

[20] V. A. Nepomniaschy, I. S. Anureev, A. V. Promsky, "Verification-Oriented Language C-Light and Its Structural Operational Semantics", *Perspectives of System Informatics*, 5th International Andrei Ershov Memorial Conference, PSI 2003 (Akademgorodok, Novosibirsk, Russia, July 9–12), Lecture Notes in Computer Science, **2890**, Springer, 2003, 103–111, http://link.springer.com/chapter/10.1007/978-3-540-39866-0_12.

DOI: 10.18255/1818-1015-2015-6-773-782

Элиминация инвариантов циклов для финитной итерации над неизменяемыми структурами данных в Си программах

Марьясов И. В.[1], Непомнящий В. А.

получена 26 октября 2015

Верификация C-программ является актуальной проблемой современного программирования. Для применения известных методов дедуктивной верификации необходимо аннотировать циклы посредством инвариантов, что во многих случаях является трудной задачей. В этой статье мы рассматриваем язык C-light, который является выразительным подмножеством языка C, соответствующего стандарту ISO. Для верификации C-light программ нами были предложены двухуровневый подход [19, 20] и метод смешанной аксиоматической семантики [1, 3, 11]. На первой стадии этого подхода исходная C-light программа транслируется [17] в программу на языке C-kernel [19], который является подмножеством языка C-light. Теорема о корректности этой трансляции была доказана в [10, 11]. По сравнению с C-light в языке C-kernel меньше операторов, что позволяет уменьшить число правил вывода при разработке аксиоматической семантики. На второй стадии этого подхода для программ на языке C-kernel порождаются условия корректности по правилам смешанной аксиоматической семантики [10, 11], которая может содержать несколько правил вывода для одной и той же программной конструкции. В таких случаях правила вывода применяются однозначно в зависимости от контекста. Заметим, что во многих случаях использование смешанной аксиоматической семантики позволяет упростить условия корректности. В этой статье представлено расширение данного подхода, которое включает наш метод верификации для финитной итерации над неизменяемыми структурами данных без выхода из тела цикла в C-light программах. Данный метод содержит новое правило вывода для таких финитных итераций без инвариантов. Это правило было реализовано в генераторе условий корректности. На стадии доказательства используется SMT-решатель Z3 [12]. Рассмотрен пример, иллюстрирующий применение данного подхода.

Статья представляет собой расширенную версию доклада на VI Международном семинаре “Program Semantics, Specification and Verification: Theory and Applications”, Казань, 2015.

Статья публикуется в авторской редакции.

Ключевые слова: Си, инварианты циклов, смешанная аксиоматическая семантика, финитная итерация, неизменяемые структуры данных, спецификация, верификация, логика Хоара

Для цитирования: Марьясов И. В., Непомнящий В. А., "Элиминация инвариантов циклов для финитной итерации над неизменяемыми структурами данных в Си программах", *Моделирование и анализ информационных систем*, **22**:6 (2015), 773–782.

Об авторах:
Марьясов Илья Владимирович, orcid.org/0000-0002-2497-6484, канд. физ.-мат. наук,
Институт систем информатики им. А.П. Ершова СО РАН
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090, Россия,
e-mail: ivm@iis.nsk.su

Непомнящий Валерий Александрович, orcid.org/0000-0003-1364-5281, канд. физ.-мат. наук, доцент,
Институт систем информатики им. А.П. Ершова СО РАН
пр. Академика Лаврентьева, 6, г. Новосибирск, 630090, Россия,
e-mail: vnep@iis.nsk.su

Благодарности:
[1]Эта работа выполнена при поддержке гранта РФФИ 15-01-05974.