СБОРНИК **НАУЧНЫХ ТРУДОВ НГТУ. – 2015. – № 1(79).** – 65–76

СОВРЕМЕННЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

УДК 62-50:519.216

СРАВНЕНИЕ МЕТОДОВ ПРЕОБРАЗОВАНИЯ ПРОГРАММНОГО ЦИКЛА С ИСПОЛЬЗОВАНИЕМ СИМВОЛЬНОЙ НОТАЦИИ*

А.А. ВОЕВОДА[1], Д.О. РОМАННИКОВ[2]

[1] *630073, РФ, г. Новосибирск, пр. Карла Маркса, 20, Новосибирский государственный технический университет, доктор технических наук, профессор кафедры автоматики. E-mail: ucit@ucit.ru*

[2] *630073, РФ, г. Новосибирск, пр. Карла Маркса, 20, Новосибирский государственный технический университет, кандидат технических наук, доцент кафедры автоматики. E-mail: rom2006@gmail.ru*

Современные технические системы – это, возможно, одни из самых сложных технических систем во всей человеческой деятельности. На сегодняшний день такие системы чаще всего состоят из интеграции программной и аппаратной частей. При этом рост сложности самой системы приводит к значительному увеличению сложности разработки и контроля за качеством программной системы. На сегодняшний день в индустрии разработки программного обеспечения нет формальных средств, которые могли бы гарантировать отсутствие программных ошибок. Большое влияние на сложность программного анализа оказывает значительное увеличение программных путей, в особенности из-за наличия условных операторов в циклах. В данной статье рассматривается подход уменьшения количества программных путей, который основан на символьной нотации. Согласно данному подходу программный цикл должен быть представлен как символьное выражение, которое определяет переменные и в котором условные операторы представлены в виде дополнительных переменных. После этого программа должна быть представлена в виде дерева с вершинами в виде контекстов. Далее полученное дерево контекстов может быть проанализировано в любом состоянии путем вычисления выражения, в котором содержатся возможные значения переменных вместо их символьного представления. Статья завершается формулированием общего алгоритма анализа программы. Задача получения формального алгоритма преобразования цикла является темой отдельного исследования и будет рассмотрена в последующих работах, так же как и представление математического аппарата для вычисления возможных значений символьных выражений при подстановке в него возможных значений переменных.

Ключевые слова: программное обеспечение, тестирование, входные интервалы, формальная верификация, динамическая верификация, верификация, проверка моделей, модели программного обеспечения, графы, тотальная корректность программ

DOI: 10.17212/2307-6879-2015-1-65-76

* Статья получена 10 января 2015 г.

ВВЕДЕНИЕ И ОБЗОР ЛИТЕРАТУРЫ

Рост сложности используемого в человеческой деятельности программного обеспечения (ПО) до сих пор делает актуальными задачи разработки ПО в требуемый срок и обеспечения приемлемого качества. Для решения этих задач программная инженерия выработала множество средств: методологические [1, 2], которые определяют основные и сопутствующие процессы разработки ПО, и формальные [3–6], являющиеся автоматизированными инструментами для поиска ошибок в программном коде. Необходимо отметить, что на сегодняшний день методологический аппарат, который является неформальным и во многом зависит от людей, которые его используют, способствует раннему обнаружению ошибок в программном продукте и его стабилизации, но не о гарантии отсутствия ошибок в нем. Формальные методы разделились на несколько групп: 1) инструменты, построенные на принципах абстрактной интерпретации [7] и статического анализа [5, 8–10]; 2) инструменты, основанные на проверке моделей [3, 4]; 3) инструменты, в основе которых лежит принцип доказательства теорем [11]. Среди всех вышеперечисленных на практике используются только статический анализ, который позволяет выявить заранее известные паттерны ошибок в коде, но при этом не гарантирует их полное выявление и появление ложных срабатываний. Проверка моделей, изначально используемая для верификации и исследования электронных схем, не нашла своего применения на практике для анализа ПО в силу чрезвычайно высокой вычислительной сложности построения глобального пространства переходов R [3, 4]. Символьный анализ можно рассматривать как расширение статического анализа, которое позволяет выполнять более формальные проверки исходного кода программы. Например, известны работы, в которых применяется символьный анализ для определения утечек памяти [12, 13], проверок границ переменных и «тупиков» [12–14], верификации параллельных программ [15].

В данной работе предлагается метод символьного анализа, который позволяет существенно сократить количество путей в программе за счет использования символьных выражений.

Далее работа организована в следующем порядке. В следующем разделе представлены основные понятия, определения символьного вычисления. В разделе «Запись циклов при помощи символьных выражений» приведен метод преобразования программного цикла с целью снижения числа путей программы для анализа. В разделе «Выводы» описаны основные результаты, полученные при выполнении данной работы.

1. ПОНЯТИЯ СЕМАНТИКИ СИМВОЛЬНОГО ВЫЧИСЛЕНИЯ

Символьные выражения – один из способов описания программ. При этом каждая переменная *vi* будет описываться своим значением в конкретный промежуток времени (в строке, в итерации и т. д.) и символом *vi* в случае, если значение не известно или может принимать множество вариантов. Например, выражение $b = a$; $a = b + 1$; может быть записано в следующем виде: $a = a$, $b = a$; $a = a + 1$, $b = a$;, где переменная с подчеркиванием означает, что это символ.

Программа представлена в виде графа управления (CFG), который представляет собой направленный маркированный граф $G = \{N, E, n_e, n_x\}$ с множеством узлов N, переходов E, входным узлом n_e и конечным узлом n_x.

Исполнение программы характеризуется определенным состоянием набора переменных на каждом шаге. Тогда состояние $s \in S$ – это функция, которая соотносит программные переменные с их символьными выражениями: $S \subseteq \{f : V \to SymExpr\}$, где V – множество переменных *vi*, а *SymExpr* – символьное выражение (*v* для переменной *v*).

Анализ программы не может быть характеризован только состояниями, кроме них еще важны и ограничения (условия), которые «накапливаются» при переходах от одной инструкции к другой в рамках одного из программных путей. Понятие контекста $c \in C \subseteq [S \times SymPred]$ представлено упорядоченным набором [*s*, *p*], где *s* – состояние, $p \in SymPred$ – условие перехода [17] от одного состояния к другому, выраженное в терминах символьных выражений для переменных из *s*.

Целью анализа программы будем считать проверку того, что в заданном состоянии *S* проверяемое выражение *cond* является истинным при любых возможных значениях переменных состояния *S* (1) [19]. Очевидно, что для выполнения такого анализа необходимо определить все возможные значения переменных в состоянии *S*

$$cond(S) = true\,. \tag{1}$$

Скрытой проблемой данной формулы является то, что количество путей в данных формулах потенциально может быть бесконечно. Большую часть в увеличение возможных путей исполнения программы вносят циклы.

2. ЗАПИСЬ ЦИКЛОВ ПРИ ПОМОЩИ СИМВОЛЬНЫХ ВЫРАЖЕНИЙ

Методы символьного анализа [12–16] не позволяют существенно уменьшить число путей в программе с циклами. В данном разделе представлен подход, который позволяет существенно снизить число программных путей в программе за счет компактной записи цикла. Предлагаемый подход основы-

вается на комбинации идей вычислений интервалов значений для переменных [19] и символьного представления программы. Рассмотрим программу на рис. 1. Типичным способом представления такой программы является построение дерева путем выполнения «раскрытия» цикла. Пример такого дерева представлен на рис. 2.

```
int a, i, b = 10;
for (i = 0; i < n; i++) {
      if (a > 0) {
            b++;
      }
}
```

Рис. 1. Пример программы с условным оператором с «неитерируемой» переменной

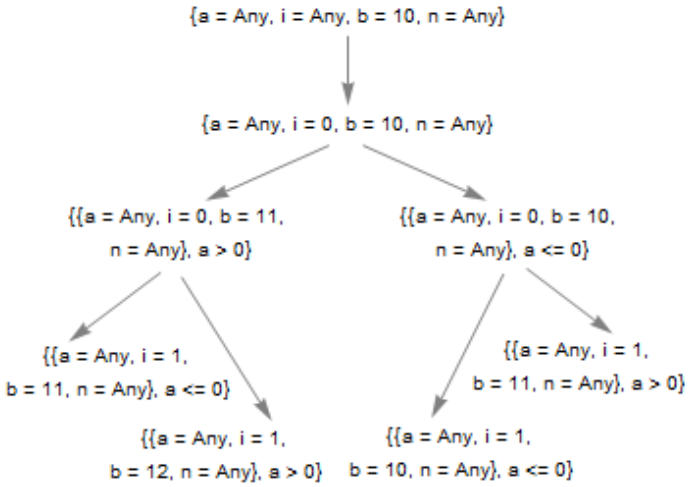

Рис. 2. Дерево вариантов исполнения программы на рис. 1[1]

Как видно из представленного на рис. 2 дерева, число возможных путей исполнения программы, которая внутри цикла содержит всего один условный оператор, увеличивается в два раза на каждой итерации.

Существенно уменьшить количество путей для анализа можно путем записи цикла в виде двух выражений $b = (b + 1)*n$, $a > 0$ и $b = b$, $a \leq 0$. В выше-

[1] Под *Any* подразумевается, что переменная не определена и может принимать любое значение, которое соответствует ее типу данных.

приведенной записи цикл заменен на два выражения, при этом после цикла переменная *b* может принимать значения исходя из этих выражений с учетом того, что *n* – количество итераций в цикле. Дерево возможных путей для символьного представления цикла представлено на рис. 3. Стоит отметить, что значительное уменьшение количества вариантов значений произошло из-за их «упаковывания» в символьное выражение $b = (b + 1)*n$, которое может быть раскрыто в множество значений при $n > 0$.

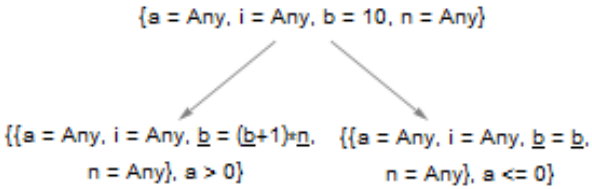

Рис. 3. Дерево вариантов исполнения программы на рис. 1 после применения символьной записи

Далее покажем, как можно использовать полученную запись для анализа программы. Допустим, что программа на рис. 1 продолжена несколькими операциями: $c = b*2$; assert($c > a$);. Тогда при исполнении программы инструкции assert в соответствии с рис. 3 возможны два варианта контекста. Значения для переменной *c* представлены в (2), а в (3) представлен предикат *broken*?, в котором вычисляется возможность того, что выражение в операторе assert будет ложным. Из выражений, приведенных в (3), видно, что существуют значения переменных *b*, *n* и *a*, при которых выражение внутри оператора assert будет ложным:

$$\begin{aligned} c1 &= ((\underline{b}+1)^n)*2 \\ c2 &= \underline{b} \end{aligned} \tag{2}$$

$$\begin{aligned} &broken?((10+1)^{Any})*2 > Any \\ &broken?(10 > Any) \end{aligned} \tag{3}$$

Рассмотрим пример программы на рис. 4. В программе в данном примере содержится условный оператор в теле цикла. Это приводит к тому, что, во-первых, из-за того, что переменная *e* меняется на каждой итерации цикла, то нельзя применить запись, как и в прошлом примере. Во-вторых, в отличие от прошлого примера нет явного значения количества итераций цикла.

```
y=1;
while (e > 0) {
     if (e % 2 == 1) {
     y = y * f;
}
f = f * f;
e = e / 2;
}
```

Рис. 4. Пример программы (из работы [16], рис. 5 и 6) с «итерируемой» переменной

В работе [16] предлагается общий вид записи циклов в виде (4), где первая строка показывает граничные значения переменных в цикле, две следующие строки – изменение переменных в цикле, последняя строка – условие перехода в цикле[2]:

$$r = \begin{cases} \forall v_i \in IV : v_i(0) = s_{in}(v_i) \\ \forall v_i \in IV : \forall [s_x, p_x, r_x] \in \overline{sc} \\ v_i(l+1) = \sigma_{s_{in},l}(s_x(v_i)) if\, \sigma_{s_{in},l}(p_x) \\ rc ::= \vee_{1 \le x \le x_w} (\sigma_{s_{in},l}(p_x)) \end{cases} \quad (4)$$

Для программы на рис. 4 выражение (5) будет иметь следующий вид:

$$r = \begin{cases} y(0) ::= 1 \\ e(0) ::= \underline{e} \\ f(0) ::= \underline{f} \\ y(l+1) ::= \begin{cases} f(l) * y(l) if\, (e(l) \bmod 2 = 1) \\ y(l) else \end{cases} \\ e(l) ::= \underline{e} / 2 \\ f(l) ::= \underline{f^{2^l}} \\ rc ::= e(l) > 0 \end{cases}. \quad (5)$$

[2] IV – множество итерируемых переменных в цикле, $\sigma_{s,e}$ – оператор подстановки в состоянии s выражения e: $\sigma_{s,e} = \{\underline{v}_1 \to v_1(e), ..., \underline{v}_j \to v_j(e)\}, \forall \underline{v}_j \in Dom(s), i \in [1, j]$ [18].

В этой формуле первые три выражения – начальные условия для переменных в цикле. После этого следует выражение, с помощью которого можно посчитать значение переменной *y* на следующей итерации. Важно подчеркнуть, что это выражение включает в себя условный оператор. Дополнительно есть выражения, с помощью которых можно вычислить значение переменных *e* и *f* без знания промежуточных значений в цикле. Последняя строка в формуле (5) выражает условие окончание цикла. Основной недостаток данной формулы заключается в том, что нет никаких рекомендаций о том, как не допустить значительного увеличения числа программных путей.

Так же как, решая задачу анализа для программы в рис. 1, мы можем преобразовать цикл в программе на рис. 4 в символьное выражение, которое представлено в (6):

$$\begin{aligned} &\underline{y} = \underline{y}(0) * \underline{f}(0)^m \\ &\underline{f} = \underline{f}(0)^{2^n} \\ &\underline{e} / 2^n > 0 \\ &0 \le m \le n. \end{aligned} \tag{6}$$

В данном выражении вычисляемые в цикле переменные записаны в форме выражений, которые зависят от начального значения переменных перед циклом (*f, y*) и дополнительно введенных переменных (*n, m*). Представление переменных *f* и *y* в таком виде позволяет выразить все их возможные значения, которые могут быть получены после исполнения цикла. Также важно заметить, что в таком виде оператор присваивания не содержится явно. То есть не произошло потери информации за счет того, что все возможные значения скрыты посредством введения новой переменной и ее возможности принимать различные значения. Побочным эффектом такой записи является наличие новой переменной с неопределенным значением, работа с которым будет показана далее.

Таким образом, можно записать алгоритм преобразования цикла.

1. Переменные программы записать в виде *vi*.

2. Определить значение всех переменных после цикла с использованием через начальные значения переменных *vi*(0).

3. Определить условие окончание цикла.

4. Если новые переменные были задействованы в выражениях во втором пункте алгоритма, то необходимо дописать соотношения для этих переменных (например, $n > 0$, $m < 10$ и т. д.).

5. Основываясь на полученном представлении цикла, построить состояния и контексты для программы.

6. Выполнить анализ требуемого контекста с учетом всех возможных значений переменных в состоянии из контекста.

ВЫВОДЫ

В данной работе была рассмотрено символьное представление программы и ее дальнейший анализ, который основан на использовании контекстов, состояний и условий переходов между состояниями. В данной работе выполнено сравнение двух методов подходов к записи программных циклов с целью их дальнейшего анализа. Первый подход взят из работы [16], второй – предложен авторами статьи. Суть предлагаемого подхода заключается в построении выражения для конечных значений переменных при помощи использования только начальных значений переменных. При этом в выражения вводятся новые переменные, обычно целого типа. Полученное представление позволяет выразить набор возможных значений переменных при помощи символьной записи, которая не содержит условных операторов, что приводит к возможности в значительной мере уменьшить число путей для анализа программы. В дальнейшем при анализе предлагается упростить полученное выражение за счет преобразования переменных со значениями *Any*.

СПИСОК ЛИТЕРАТУРЫ

1. *Орлов С.А., Цилькер Б.Я.* Технологии разработки программного обеспечения: современный курс по программной инженерии. – 4-е изд. – СПб.: Питер, 2012. – 608 с. – (Учебник для вузов) (Стандарт третьего поколения).

2. Верификация автоматных программ / С.Э. Вельдер, М.А. Лукин, А.А. Шалыто, Б.Р. Яминов; Санкт-Петербургский государственный университет информационных технологий, механики и оптики. – СПб.: Изд-во СПбГУ ИТМО, 2011. – 242 с.

3. *Кларк Э., Грамберг О., Пелед Д.* Верификация моделей программ. Model Checking. – М.: МЦНМО, 2002. – 416 с.

4. *Карпов Ю.Г.* Model checking. Верификация параллельных и распределенных программных систем. – СПб.: БХВ-Петербург, 2010. – 560 с.

5. *Bush W., Pincus J., Sielaff D.* A static analyzer for finding dynamic programming errors // Software: Practice and Experience. – 2000. – Vol. 30, iss. 7. – P. 775–802. – doi: 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H.

6. *Falk H., Marwedel P.* Source code optimization techniques for data flow dominated embedded software. – New York: Springer Science; Business Media, 2004. – 226 p. – doi: 10.1007/978-1-4020-2829-8.

7. *Cousot P., Cousot R.* Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Los Angeles, California, January, 17–19, 1977. – New York: ACM Press, 1977. – P. 238–252. – doi: 10.1145/512950.512973.

8. Coherent clusters in source code / S. Islam, J. Krinke, D. Binkley, M. Harman // The Journal of Systems and Software. – 2014. – Vol. 88. – P. 1–24. – doi: 10.1016/j.jss.2013.07.040.

9. *Horwitz S., Reps T., Binkley D.* Interprocedural slicing using dependence graphs // ACM Transactions on Programming Languages and Systems (TOPLAS). – 1990. – Vol. 12, iss. 1. – P. 26–60. – doi: 10.1145/77606.77608.

10. *Ferrante J., Ottenstein K.J., Warren J.D.* The program dependence graph and its use in optimization // ACM Transactions on Programming Languages and Systems (TOPLAS). – 1987. – Vol. 9, iss. 3. – P. 319–349. – doi: 10.1145/24039.24041.

11. Handbook of automated reasoning / J.A. Robinson, A. Voronkov, eds. – Amsterdam: Elsevir Science; Cambridge: The Mit Press, 2001. – 2150 p.

12. *Scholz B., Blieberger J., Fahringer T.* Symbolic pointer analysis for detecting memory leaks // Proceedings of ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation" (PEPM'00), Boston, Massachusetts, USA, January 22–23, 2000. – New York: ACM Press, 2000. – P. 104–113. – doi: 10.1145/328690.328704.

13. *Rugina R., Rinard M.* Symbolic bounds analysis of pointers, array indices, and accessed memory regions // PLDI'00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 18–21, 2000. – New York: ACM Press, 2000. – P. 182–195. – doi: 10.1145/358438.349325.

14. *Bush W.R., Pincus J.D., Sielaff D.J.* A static analyzer for finding dynamic programming errors // Software: Practice and Experience. – 2000. – Vol. 30, iss. 7. – P. 775–802. – 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H.

15. Combining symbolic execution with model checking to verify parallel numerical programs / S.F. Siegel, A. Mironova, G.S. Avrunin, L.A. Clarke // ACM Transactions on Software Engineering and Methodology. – 2008. – Vol. 17, iss. 2. – P. 10:1–10:34. – doi: 10.1145/1348250.1348256.

16. *Burgstaller B., Scholz B., Blieberger J.* A symbolic analysis framework for static analysis of imperative programming languages // The Journal of Systems and Software. – 2014. – Vol. 85, iss. 6. – P. 1418–1439. – doi: 10.1016/j.jss.2011.11.1039.

17. *Fahringer T., Scholz B.* Advanced symbolic analysis for compilers. – Berlin; Heidelberg: Springer-Verlag, 2003. – 136 p. – (Lecture Notes in Computer Science; vol. 2628). – doi: 10.1007/3-540-36614-8.

18. *Burgstaller B.* Symbolic evaluation of imperative programming languages: technical report N 183/1-138 / Vienna University of Technology, Department of Avtomation. – Vienna, 2005. – 146 p.

19. *Воевода А.А., Романников Д.О.* Способы представления программ и их анализ // Сборник научных трудов НГТУ. – 2014. – № 3 (77). – С. 81–99.

Воевода Александр Александрович, доктор технических наук, профессор кафедры автоматики Новосибирского государственного технического университета. Основное направление научных исследований – управление многоканальными объектами. Имеет более 200 публикаций. E-mail: ucit@ucit.ru.

Романников Дмитрий Олегович, кандидат технических наук, доцент кафедры автоматики Новосибирского государственного технического университета. Основное направление научных исследований – формальная верификация, проверка моделей. Имеет 34 публикации. E-mail: rom2006@gmail.ru.

Comparison of methods of program cycle transformation with using of the symbolic notation*

A.A. Voevoda[1], D.O. Romannikov[2]

[1] *Novosibirsk State Technical University, 20 Karl Marks Avenue, Novosibirsk, 630073, Russian Federation, doctor of Technical Sciences, professor of the automation department. E-mail: ucit@ucit.ru*

[2] *Novosibirsk State Technical University, 20 K. Marks prospekt, Novosibirsk, 630073, Russian Federation, PhD (Eng.), associate professor of the automation department. E-mail: rom2006@gmail.ru*

Modern technical systems might be the most difficult artefacts. It is a common nowadays that a technical system is an integration of hardware and software parts. Moreover, growth of system complexity leads to significant growth of program complexity. Nowadays we do not have special tools that can guarantee absence of program errors. Big influence of difficulty of program analysis makes an increase of program paths, especially if path forks are in a cycle. In this paper, we offer an approach of reducing of a number of program paths that based on a symbolic notation. According to the approach, a program cycle should be represented in form of symbolic expressions that define expressions for variables. Wherein conditional operators are represented as additional variables in symbolic expressions. After that program should be

* *Received 10 January 2015.*

depicted as tree of contexts. Subsequently program can be analysed in any state of this tree by solving an expression with possible values instead of variables.
At the end of the paper algorithm of program analysis exists. A task of carrying out of formal cycle transformation algorithm will be considered in future works, like a mathematical apparatus for calculation of possible values of symbolic expressions by substituting of all possible values of variables.

Keywords: software, testing, input intervals, formal verification, dynamic verification, verification, model checking, software models, graphs, total correctness of programs

DOI: 10.17212/2307-6879-2015-1-65-76

REFERENCES

1. Orlov S.A., Tsil'ker B.Ya. *Tekhnologii razrabotki programmnogo obespecheniya: sovremennyi kurs po programmnoi inzhenerii* [Technologies of software development: modern course on program engineering]. St. Petersburg, Piter Publ., 2012. 608 p.

2. Vel'der S.E., Lukin M.A., Shalyto A.A., Yaminov B.R. *Verifikatsiya avtomatnykh programm* [Automate program verification]. St. Petersburg, SPbGU ITMO, 2011. 242 p.

3. Clarke E.M., Grumberg O., Peled D. *Model checking*. Cambridge, London, MIT Press, 2001. (Russ. ed.: Klark E., Gramberg O., Peled D. *Verifikatsiya modelei programm: Model checking*. Moscow, MTsNMO Publ., 2002. 416 p.).

4. Karpov Yu.G. *Model Checking. Verifikatsiya parallel'nykh i raspredelennykh programmnykh sistem* [Model Checking. Verification of parallel and distributed software systems]. St. Petersburg, BHV-Petersburg, 2010. 560 p.

5. Bush W., Pincus J., Sielaff D. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 2000, vol. 30, iss. 7, pp. 775–802. doi: 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H

6. Falk H., Marwedel P. *Source code optimization techniques for data flow dominated embedded software*. New York, Springer Science, Business Media, 2004. 226 p. doi: 10.1007/978-1-4020-2829-8

7. *Cousot P., Cousot R.* Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL'77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Los Angeles, California, January, 17–19, 1977. New York, ACM Press, 1977, pp. 238–252. doi: 10.1145/512950.512973

8. Islam S., Krinke J., Binkley D., Harman M. Coherent clusters in source code. *The Journal of Systems and Software*, 2014, vol. 88, pp. 1–24. doi: 10.1016/j.jss.2013.07.040

9. Horwitz S., Reps T., Binkley D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1990, Vol. 12, iss. 1, pp. 26–60. doi: 10.1145/77606.77608

10. Ferrante J., Ottenstein K.J., Warren J.D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987, vol. 9, iss. 3, pp. 319–349. doi: 10.1145/24039.24041

11. Robinson A., Voronkov A., eds. *Handbook of automated reasoning*. Amsterdam, Elsevir Science, Cambridge, The Mit Press, 2001. 2150 p.

12. Scholz B., Blieberger J., Fahringer T. Symbolic pointer analysis for detecting memory leaks. *Proceedings of ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation" (PEPM'00)*, Boston, Massachusetts, USA, January 22–23, 2000. New York, ACM Press, 2000, pp. 104–113. doi: 10.1145/328690.328704

13. Rugina R., Rinard M. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *PLDI'00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 18–21, 2000. New York, ACM Press, 2000, pp. 182–195. doi: 10.1145/358438.349325

14. Bush W.R., Pincus J.D., Sielaff D.J. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 2000, vol. 30, iss. 7, pp. 775–802. 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H

15. Siegel S.F., Mironova A., Avrunin G.S., Clarke L.A. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology*, 2008, vol. 17, iss. 2, pp. 10:1–10:34. doi: 10.1145/1348250.1348256

16. Burgstaller B., Scholz B., Blieberger J. A symbolic analysis framework for static analysis of imperative programming languages. *The Journal of Systems and Software*, vol. 85, iss. 6, pp. 1418–1439. doi: 10.1016/j.jss.2011.11.1039

17. Fahringer T., Scholz B. Advanced symbolic analysis for compilers. *Lecture Notes in Computer Science*. Vol. 2628. Berlin, Heidelberg, Springer-Verlag, 2003. 136 p. doi: 10.1007/3-540-36614-8

18. Burgstaller B. *Symbolic evaluation of imperative programming languages*. Technical report no. 183/1-138. Vienna University of Technology, Department of Avtomation. Vienna, 2005. 146 p.

19. Voevoda A.A., Romannikov D.O. Sposoby predstavleniya programm i ikh analiz [Methods of program representation and analysis]. *Sbornik nauchnykh trudov NGTU – Transaction of scientific papers of the Novosibirsk state technical university*, 2014, no. 3 (77), pp. 81–98.