СБОРНИК **НАУЧНЫХ ТРУДОВ НГТУ. – 2015. – № 4(82).** – 92–107

СОВРЕМЕННЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

УДК 62-50:519.216

АЛГОРИТМ АНАЛИЗА МАССИВА В ПРОГРАММЕ С ИСПОЛЬЗОВАНИЕМ ПРОИЗВОЛЬНЫХ ПРОВЕРОК*

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*

В данной статье анализируется два способа анализа алгоритмов программного обеспечения. Первый из них основан на использовании механизма цепи рекурсии (chain recurrences). Данный способ потенциально может существенно упростить задачу анализа в случае возможности представления программы в виде базовой рекурсии (base recurrence), которая позволяет представить выражение под циклом в виде, из которого можно получить значение выражения на *i*-й итерации за O(1). Так как в настоящее время эта задача не решена, то метод анализа программ, основанный на цепях рекурсии, может быть применен только в ограниченном количестве программ (стоит сказать, что количество таких программ велико), где нет сложных циклов и выражений под ними.
Второй способ – это алгоритмический способ анализа. В статье приведена группа алгоритмов, с помощью которой можно выполнять проверку соотношений больше/меньше на произвольном массиве в программе. В качестве апробации был использован алгоритм сортировки вставкой. Приведенная группа алгоритмов проверки состоит из алгоритма определения путей в программе (при этом подразумевается, что пути строятся из расчета, что обратные ребра графа управления (control flow graph) входят в каждый путь только один раз), алгоритма анализа исследуемого соотношения в программе. Последний алгоритм предполагает предварительную обработку данных, которая не рассматривается в статье, но является достаточно простой, а также выполнение проверки соотношения на всей длине массива, без учета его составных частей. Алгоритм проверки может быть модифицирован (основная его часть будет сохранена) для проверки других видов соотношений.

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

DOI: 10.17212/2307-6879-2015-4-92-107

* Статья получена 29 октября 2015 г.

ВВЕДЕНИЕ И ОБЗОР ЛИТЕРАТУРЫ

В настоящее время при разработке программного обеспечения (ПО) отсутствие ошибок в основных пользовательских сценариях использования ПО проверяется при помощи методологических средств и методик [1] в лабораторных условиях. Разработка и внедрение формальных инструментов, к которым можно отнести инструменты проверки моделей [2–4, 15], статические анализаторы [5–10, 14] и другие, могло бы существенно упростить процесс поиска программных ошибок. Но на практике они не нашли широкого применения либо из-за большого числа ложно-положительных срабатываний, либо из-за недопустимо большого времени анализа ПО. Тем не менее разработка формальных инструментов остается актуальной задачей в индустрии программного обеспечения. Наиболее перспективным, с точки зрения авторов, является метод символьного анализа [11–14, 16–18], который был успешно применен для обнаружения утечек памяти [18], обращения к несуществующим элементам массива [18], определения «тупиков» (deadlocks) [17, 18] и др.

Особое значение в данной тематике уделено проблемам описания программных циклов в виде, который бы позволял получить конечное число вариантов для анализа выражений в цикле [16–19], а также вопросам его формального представления [16–19]. В работе [18] предложена идея для представления выражений в циклах в виде «цепи повторений» (*chain recurrences*), которые, с одной стороны, позволяют существенно ускорить вычисление выражений в цикле, с другой – представляют формализм для описания исходного кода программы. В работе [19] данный формализм использовался для создания символьного «фреймворка» (*framework*), а в работе [20] он был дополнен операторами «монотонности» и «итерирования» для анализа программ.

1. ОСНОВНЫЕ ПОНЯТИЯ

В данной работе рассматривается преобразование выражений под циклом. Общая идея такого преобразования была изложена в работе [20], где предлагается для анализа циклов обозначить переменной n – количество итераций в цикле, а также составить выражения для определения переменной n и переписать выражения внутри цикла с учетом n.

Первой из основных проблем данного подхода является то, что некоторые выражения невозможно представить как функцию $f(n)$, где n – количество итераций в цикле. Например, к таким функциям относится функция $f(n) = f(n-1) + f(n-2)$ – функция вычисления чисел Фибоначчи. Ко второй проблеме можно отнести необходимость проверки условия окончания цикла, в

которой нужно проверить, что для всех значений $i \in [i_0, i_{n-1})$ выполняется условие, при котором цикл не завершает свою работу, а для i_{n-1} оно не выполняется.

2. ПРИМЕРЫ АНАЛИЗА

Рассмотрим пример на рис. 1, на котором приведен код программы из цикла с выражениями внутри. В данном цикле количество итераций *n* неизвестно. Для выражений под циклом можно выполнить следующие преобразования (рис. 2). В результатах преобразования получен набор выражений, который не находится под циклом. Обозначение переменных после преобразования имеет следующую структуру: <variable_name>_n, где variable_name – имя переменной до преобразования, _n – значение переменной после цикла (после исполнения *n* итераций). Очевидно, что переменная a, принимающая значение константы на каждой итерации цикла, будет иметь значение этой константы после исполнения хотя бы одной итерации.

```
int i, a, b, c, d, n;
for (i = 0; i < n; ++i) {
  a = 0;
  b = b + 1;
  c = a + b;
  d = b + d;
}
```

Рис. 1. Пример программы для анализа (цикл с выражениями)

```
int i, a, b, c, d, n;
a_n = 0;
b_n = b_0 + 1*n;
c_n = a_n + b_n = 0 + b_0 +
1*n;
d_n = b_n + d_n-1 = n*b_0 +
1*(n + n-1 + n-2 + ... + 1) +
d_0;

i_0 = 0; i_n >= n; i_j = i_j-1
+ 1; i_j < n.
```

Рис. 2. Пример преобразованной программы на рис. 1

При помощи программного пакета Mathematica можно найти решение составленной системы уравнений.

В качестве следующего примера рассмотрим пример на рис. 3 – алгоритм сортировки вставками (*insertion sort*).

Граф переходов (*Control Flow Graph (CFG)*) для алгоритма на рис. 3 приведен на рис. 4. На представленном графе содержатся два типа состояний – состояние с действиями (2, 4, 5, 6) и состояния условного перехода (1, 3). Далее рассмотрим возможные пути исполнения программы. При этом будем рассчитывать пути таким образом, что «обратные» ребра графа будут входить в путь только один раз. Тогда, получим следующие пути: a) 1→exit; b) 1→2→3→5→6→1→exit; c) 1→2→3→4→5→6→1→exit.

```
void InsertionSort(int* arr, int len) {
  int i, j, key;
  for (int i = 1; i < len; ++i) {
    key = arr[len];
    j = i - 1;
    while (j >= 0 && key <= arr[j]) {
      arr[j+1] = arr[j];
      --j;
    }
    arr[j+1] = key;
  }
}
```

Рис. 3. Алгоритм сортировки вставками

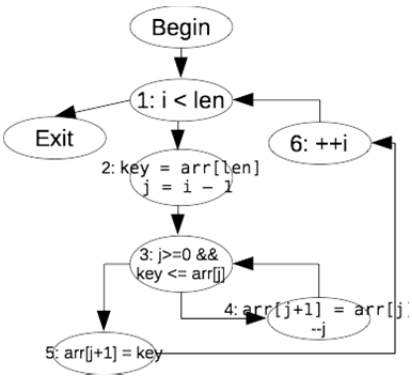

Рис. 4. CFG для программы на рис. 3

Рассмотрим (см. таблицу) подробнее возможные пути исполнения программы и проанализируем их. При исполнении первого пути выполняются состояние «Begin», переменная «i» принимает значение единицы, а из состояния «1» управление передается в «Exit». При этом условие «i < len» ложно. При исполнении второго пути изменяется массив «arr» в соответствии с рис. 5. При этом известно, что выражение «j >= 0 && key <= arr[j0]» ложно, что возможно либо при «j < 0», либо при «key > arr[j0]». Первая часть выражения выполняется, так как «j = i – 1 = 1 – 1 = 0», вторая – неизвестно, так как на момент выполнения программы нет информации о значениях в массиве, но предположение о том, что программа выполняется по этому пути, означает, что выражение ложно, а это возможно при «key > arr[j0]». Исполнение третьего пути аналогично двум предыдущим с учетом модификации массива, как показано на рис. 6. При этом

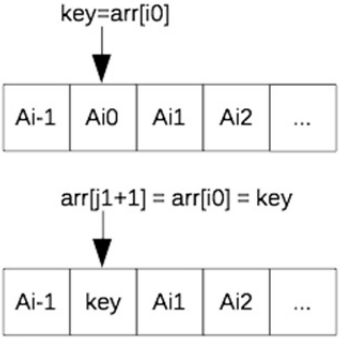

Рис. 5. Модификация массива «arr» в программе (путь 2) на рис. 3

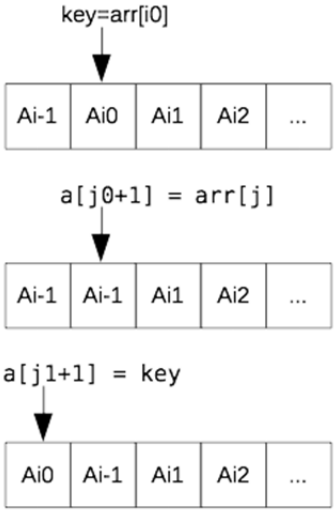

Рис. 6. Модификация массива «arr» в программе (путь 3) на рис. 3

Пути исполнения программы и их анализ

Номер пути	Значения переменных и соотношение между ними
1	`i = 1` `i < len - false`

2	`i0 = 1`
	`key = arr[i0]`
	`j = i0 – 1 = 0`
	`arr[j0 + 1] = key`
	`j0 >= 0 && key <= arr[j0] - false`

3	`i0 = 1`
	`key = arr[i]`
	`j = i0 – 1 = 0`
	`arr[j0+1] = arr[j0]`
	`j1 = j0 - 1`
	`arr[j1 + 1] = key`
	`j0 >= 0 && key <= arr[j0] - true`

следует отметить несколько моментов: во-первых, итерируемая переменная рассматривается только на одной итерации цикла, но это позволяет сделать вывод о том, что случается с переменными в программе при прохождении ветки. На основании этой информации и информации о том, при каких условиях начинает исполняться данная ветка программы, можно сделать вывод о том, какие соотношения будут с переменными при ее исполнении. Например, в пути 2 используется итерируемая переменная «i», но, несмотря на то что тут учитывается только одно ее значение – начальное, можно сделать вывод о том, что будет в программе при других значениях переменной «i», которые определяются выражением {1, +, 1}.

Исполнение программы по пути 3 рассмотрим более подробно. В этом варианте содержится вложенный цикл и модификация элементов массива. Во вложенном цикле выполняется «a[j0+1] = arr[j0]», при этом модифицируется массив «arr» согласно рис. 6 (вторая строчка). Исполнение вложенного цикла приводит к зависимостям (см. рис. 7). На рис. 7 учитывается только одна итерация переменной «j», но их может быть большее количество. Условием исполнения цикла является выражение «j >= 0 && key <= arr[j0]», следовательно цикл заканчивается при «j < 0» или при «key > arr[jn]», где «jn» – последнее значение «j» перед выходом из цикла.

```
 i :  0  1  2 …  i-2  i-1  i    i+1 …
 j0: -1  0  1 … j0-1  j0  j0+1 j0+2 …
arr: a0 a1 a2 … ai-2 ai-1 ai-1 ai+1 …
ai > ai-1 || j1 < 0
j1 = j0 – 1 = i – 1 – 1
```

Рис. 7. Зависимости между переменными и итоговый массив «arr»

Рассмотрим переменные выражения, которые зависят от итерируемой переменной во вложенном цикле. В случае программы на рис. 3 это все выражения во вложенном цикле, а также зависимости, полученные на рис. 7. Рассмотрим зависимости подробнее. На основании одной итерации цикла можно сделать вывод, что в исходном массиве было выполнено изменение переменной «arr[j+1]» на переменную «arr[j]»; при условии, что в цикле выполнена не одна, а произвольное количество итераций, то данную замену можно переписать как «arr[jn+1] = arr[jn]», где «jn» - {in –1, +, –1}. Зависимости между переменными, полученными на основании исполнения итераций цикла принимают вид «arr[jn] <= arr[in]».

Далее рассмотрим алгоритмы анализа программы на основании полученных зависимостей для проверки вида «assert(arr[i] <= arr[j])» $\forall$ i, j $\in$ [0, arr-length - 1], i < j, где «arr-length» – длина массива «arr» (рис. 8).

```
GetPaths(cfg, startIndex, endIndex):
  paths = []
  Stack s
  if (cfg.length >= startIndex):
    return paths
  s.push(cfg[startIndex])
  while (!s.empty()):
    path = s.top()
    s.pop()
    currentNode = path[path.size - 1]
    while (currentNode != IF_NODE &&
           currentNode[0] существует &&
           переход из currentNode в currentNode[0] не является
обратным переходом):
      path += currentNode
      currentNode = currentNode[0]
    if (currentNode == IF_NODE):
      s.push(path + currentNode[0])
      s.push(path + currentNode[1])
    else if (currentNode == cfg[endIndex]):
      paths += path
  return paths
```

Рис. 8. Алгоритм определения путей для анализа

На рис. 8 представлен алгоритм поиска путей в графе управления (*Control Flow Graph*) с учетом того, что все обратные ребра графа (*back edge*) представлены в пути только один раз. Алгоритм является модификацией обхода графа в глубину, в которой вместо отмечания уже пройденных вершин графа учитываются только маршруты, в которые обратные ребра входят не более одного раза (условие в *while* операторе на рис. 8). Алгоритм начинает свою работу с добавления первого узла в стек (если узла не существует, то он завершается). Далее, пока стек не будет пустым, а это может произойти при прохождении всех путей в графе, выполняется формирование пути до тех пор, пока текущая вершина (*currentNode*) не будет условным оператором (*IF_NODE*) или будет последней в пути программы, или будет обратным переходом, который уже был в пути. Стоит отметить, что, возможно, для определения обратных переходов потребуется предварительно выполнить обход в глубину и поместить все обратные переходы в хеш-таблицу для осуществления быстрой проверки перехода. Далее возможны несколько случаев, при которых текущая вершина (*currentNode*): 1) оказывается условным переходом (*IF_NODE*), и тогда оба возможных пути помещаются в стек и исследуются; 2) является вершиной, до которой ищутся пути. При этом путь добавляется в

массив путей «paths» и алгоритм переходит к анализу остальных путей; 3) является конечной вершиной в пути, но не той, до которой мы ищем пути. В этом случае необходимо продолжать исследовать пути без добавления их в массив «paths». Отдельно стоит отметить, что текущий алгоритм может быть улучшен за счет предварительного анализа и учета циклов, для чего необходимо определить циклы при первом прохождении обходом в глубину (которое используется для определения обратных переходов) и формировании дополнительных путей с циклом при втором прохождении.

После определения всех путей в программе необходимо их все пройти и собрать информацию о соотношениях между переменными на программном пути. Данная задача не решается в статье и должна быть рассмотрена отдельно. На рис. 9 приведен алгоритм анализа программы на основании данных, собранных при анализе каждого из пройденных путей из предыдущего алгоритма.

Входными данными данного алгоритма являются проверяемое выражение (*ratioAssert*) и набор «путь: соотношения между переменными», представленные в предподготовленном формате (см. рис. 10).

```
VerifyArrayRatio(paths, ratioAssert):
  errors = []
  for path in paths:
    errors += CheckOnPath(path, ratioAssert)
  if errors не пустой массив:
    отобразить список контр-примеров errors


CheckOnPath(path, ratioAssert):
  # функция проверки выражения на одном из пути в программе
  assignments = GetAssignments(path)
  ratioAssertPath = []
  for state in path:
    if Possible(state, assignments) == false:
      return []
    if массив из ratioAssert не содержится в state:
      continue
    ratioAssertPath += SimplifyState(state, assignments)
  finalState = {}
  for state in ratioAssertPath:
    finalState = объединение finalState и state с большего
                 приоритета состояний из state
```

Рис. 9. Алгоритм анализа программы на основании соотношений между переменными. Начало (см. также с. 100 и 101)

```
  # проверка выражения ratioAssert
  errors = []
  for cond in finalState:
    if cond не может быть проверен на ratioAssert:
      continue
    if cond противоречит ratioAssert:
      errors += ошибка выполнения ratioAssert на пути path в
                условии cond
  return cond


GetAssignments(path):
  # функция возвращает словарь, в котором указаны состояния,
  # где выполнялось изменение значения переменных
  ret = {} # возвращаемый словарь
  for state in path:
    for var in state:
      if переменная была изменена:
        ret[var.name] += [state.number, var.value]
  return ret


Possible(state, assignments):
  # функция возвращает true, если выполнение переданного
  # состояния возможно. Например, i0 < len, где i0 = 1,
  # а len = any возможно, а 1 > 2 – нет
  if !state.condition: # все состояния, в которых только
    return true        # операции присвоения допустимы
  vars = {}
  for var in state:
    vars[var.name] = список значения переменной из assignments
  # vars содержит словарь переменная: массив возможных значений
  for var1 in vars:
    for var2 in vars:
      if var1 == var2:
        continue
      op = операнд для выражения var1 op var2 из state
      if не существует op:
        return
      for val1 in var1:
        for val2 in var2:
          if невозможно ли выражение val1 op val2:
```

Рис. 9. Продолжение (см. также с. 99 и 101)

```
            return false
  return true


SimplifyState(state, assignments):
  simplifiedState ={}
  for var in state:
    sVar = значение переменной var из assignments
           с учетом последовательной замены входящих в
           переменную var составляющих (например, для key
           можно последовательно заменить на arr[j0], а потом
           на arr[i0 - 1])
    simplifiedState[var] = sVar
  return simplifiedState
```

Рис. 9. Окончание (см. также с. 99 и 100)

В данном алгоритме поддерживается проверка выражения *ratioAssert* в виде a[i] op a[j], i > j, где op – оператор больше или меньше, сравнение выполняется по всей длине массива. В приведенном алгоритме можно увеличить число проверяемых выражений путем модификации окончания функции *CheckOnPath* (от комментария о проверке выражения *ratioAssert*), при этом основная часть алгоритма останется неизменной.

```
[
  1: [{i0: 1, j, k, int* arr, len}, {i0 >= len}],
  2: [{i0: 1, j, k, int* arr, len}, {i0 < len}, {key: arr[i0]},
{j0: i0 – 1}, {j0 < 0 || key > arr[j0]}, {arr[j0 + 1]: key},
{i1: i0 + 1}, {i1 >= len}],

  3: [{i0: 1, j, k, int* arr, len}, {i0 < len}, {key: arr[i0]},
{j0: i0 – 1}, {j0 >= 0 && key <= arr[j0]}, {arr[j0+1]:
arr[j0]}, {j1: j0 -1} {arr[j0 + 1]: key},  {i0 >= len}]
]
```

Рис. 10. Формат структуры данных *path* (из алгоритма на рис. 9) для проверки соотношений в программе

Алгоритм на рис. 9 начинается вызовом функции *VerifyArrayRatio*, в которую передается проверяемое выражение и само выражение *ratioAssert*. Далее для каждого пути ищутся возможные контр-примеры и, если они были найдены, выводится сообщение о неисполнении условия на проверяемом условии.

Функции *CheckOnPath* используется для проверки исполнения исследуемого выражения на одном пути. Она начинается с получения словаря (*assignments*), в котором для каждой переменной на пути содержатся присваивания и номера состояний, в которых эти присваивания произошли. Данный словарь далее используется в функции проверки возможности исполнения состояния и для упрощения выражения в состоянии. Для каждого состояния на проверяемом пути выполняется проверка на возможность его исполнения и наличия переменных из исследуемого выражения. При неисполнении данных условий цикл переходит к следующей итерации. Если условия выполняются, то упрощенное функцией *SimplifyState* выражение записывается в переменную *ratioAssertPath* и далее используется для проверки выражения *ratioAssert*. Функция *Possible* нужна для того, чтобы исключить из проверки пути, по которым программа заведомо не может быть исполнена. Она основана на проверке выражения на всех значениях выражения из *state*.

ЗАКЛЮЧЕНИЕ

Таким образом, в данной статье анализируются подходы к проверке алгоритмов на примере алгоритма сортировки вставкой. Изначально рассматривался вариант анализа с использованием методов преобразования итерируемых выражений, основанных на *chain recurrence* (CR). Данный способ мог бы существенно помочь в анализе программ при условии, что все выражения в циклах в программе могут быть представлены в виде *base recurrence* (BR), а не CR. Данное допущение позволило бы определять выражение в цикле на *i*-й итерации за O(1), а также позволило бы получить выражение для системы неравенств, решив которую можно определить контр-примеры в программе.

В качестве иллюстрации второго способа анализа программ приведен алгоритм для проверки операций больше/меньше для элементов массива в программе (на данный момент алгоритм работает только для проверки вышеприведенных выражений, но может быть модифицирован с сохранением основной его части для проверки более существенного числа выражений).

В качестве целей дальнейшего исследования авторы видят усовершенствование алгоритма в части поэтапной проверки массива (т. е. случаев, когда несколько выражений обеспечивают общую верность выражения на массиве, но не полностью, а отдельными частями) и расширения количества используемых выражений для анализа; поддержку различных типов циклов в алгоритме анализа; более формализованную часть сравнения циклов в алгоритме.

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. *Abramsky S., Hankin C*. An introduction to abstract interpretation [Electronic resource]. – URL: https://www.cs.virginia.edu/~weimer/2007-615/reading/AbramskiAI.pdf (accessed: 10.03.2016).

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, 17–19 January 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. – 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. – 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, 22–23 January 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, 18–21 June 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. 81–98.

20. *Engelen R.A. van*. The CR# algebra and its application in loop analysis and optimization: technical report TR-041223 / Florida State University. – Florida, 2004. – 13 p.

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 более 40 публикаций. E-mail: rom2006@gmail.ru

Algorithm array analysis using random inspections*

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

[1] *Novosibirsk State Technical University, 20 K. Marx Prospekt, Novosibirsk, 630073, Russian Federation, D. Sc. (Eng.), professor of the automation department. E-mail: ucit@ucit.ru*
[2] *Novosibirsk State Technical University, 20 K. Marx Prospekt, Novosibirsk, 630073, Russian Federation, Ph. D. (Eng.), associate professor of the automation department. E-mail: rom2006@gmail.ru*

This article examines two methods of analysis software algorithms. The first of these is based on the mechanism of recursion chain (chain recurrences). This method has the potential to greatly simplify the task of analysis in the case of the possibility of presenting the program as a basic recursion (base recurrence), which allows you to submit an expression of a cycle in the form from which you can get the value of the expression on the i-th iteration is O (1). Since at the present time, this problem is not solved, the program analysis method based on recursion chains may be applied only in a limited number of programs (it should be said that the number of such programs is large), where there are no complicated cycles and expressions for them.
The second way - it is an algorithmic method of analysis. The article describes a group of algorithms, which can be used to perform checks of relations over / under on any array in the program. As testing sorting algorithm was used insert. The above review team consists of algorithms determine how the algorithm in the program (it being understood that the paths are built on the basis of that return control edges of the graph (control flow graph) are included in each path only once), algorithm analysis investigated the relation to the program. Last algorithm assumes data pre-processing, which is not considered in the article, but it is quite simple, as well as the ratio of the scan on the entire length of the array, without regard to its constituent parts. checking algorithm can be modified (base portion is retained) for checking other types of relations.

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

DOI: 10.17212/2307-6879-2015-4-92-107

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 Publ., 2011. 242 p.

3. Clarke E.M., Grumberg O., Peled D. *Model checking*. Cambridge, London, MIT Press, 2001. 314 p. (Russ. ed.: Klark E., Gramberg O., Peled D. *Verifikatsiya modelei program. Model checking*. Moscow, MTsNMO Publ., 2002. 416 p.).

* *Received 29 October 2015.*

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 Publ., 2010. 560 p.

5. Abramsky S., Hankin C. An introduction to abstract interpretation. Available at: https://www.cs.virginia.edu/~weimer/2007-615/reading/AbramskiAI.pdf (accessed 10.03.2016)

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, 17–19 January 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*, 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*, 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, 22–23 January 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, 18–21 June 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. doi: 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 Trans-*

actions 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.

20. Engelen R.A van. *The CR# algebra and its application in loop analysis and optimization*. Technical report TR-041223. Florida State University. Florida, 2004. 13 p.