СБОРНИК **НАУЧНЫХ ТРУДОВ НГТУ. – 2014. – № 4(78).** – 125–138

УДК 62-50:519.216

О МЕТОДЕ АНАЛИЗА ПРОГРАММ*

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, старший преподаватель кафедры автоматики. E-mail: rom2006@gmail.com

Данная работа посвящена разбору метода анализа программного обеспечения, основанного на принципе выявления всех возможных значений переменных и нахождения при помощи этого ошибок путем проверки верифицируемых формул на всех возможных значениях переменных. В работе рассмотрены примеры, на которых показаны основные принципы предлагаемого метода анализа ПО, а именно: пример программы с циклом, который показывает, каким образом можно выполнить сворачивание полного графа к «линейным» участкам программы, а затем выполнить их анализ. Вторым принципом рассматриваемого метода является анализ ПО «линейно», т. е. постепенный анализ каждой ветки программы. При этом отдельный анализ можно выполнять с места ветвления без необходимости выполнения полного анализа сначала. В работе приведены многочисленные примеры того, как необходимо выполнить разбиение основной программы на ветки. Главной особенностью такого разбиения является сохранение малого количества ветвлений. Приведена формальная семантика описания программы, предложенная П. Коусотом и используемая для формализованного описания алгоритмов анализа ПО. Исходная семантика была упрощена в части описания узлов графа – убраны лишние узлы, так как они не актуальны после преобразования графа, и расширена функция поиска возможных значений переменных после исполнения оператора. Также показано, каким обзором можно определить функцию поиска возможных вариантов переменных для бинарных и унарных операций путем ее определения для операций умножения и логического отрицания. Для остальных операций данная функция может быть легко построена по аналогии.

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

DOI: 10.17212/2307-6879-2014-4-125-138

---

* Статья получена 16 июля 2014 г.

Работа выполнена при финансовой поддержке Минобрнауки России по государственному заданию № 2014/138. Тема проекта «Новые структуры, модели и алгоритмы для прорывных методов управления техническими системами на основе наукоемких результатов интеллектуальной деятельности».

ВВЕДЕНИЕ

Данная статья является продолжением работы [1], в которой приводятся идеи по анализу программного обеспечения (ПО) на основе исходных кодов. Предлагаемый метод позволяет выполнить анализ на всем диапазоне значений переменных [2], при этом существенно увеличивается класс обнаруживаемых ошибок по сравнению с [3–6, 12, 15] вплоть до возможности анализа кода по произвольным проверкам.

В работе [1] есть несколько примеров того, как применять предлагаемый метод. В данной работе приведено улучшение метода, которое содержит формальную семантику описания алгоритмов анализа ПО.

1. ОСНОВЫ МЕТОДА АНАЛИЗА

Метод анализа ПО, предлагаемый в работе [1], основан на идее о том, что для определения всего спектра ошибок без появления ложных срабатываний необходимо иметь информацию о всех возможных значениях переменной в произвольном состоянии программы. Данное выражение обобщено в формуле (1) в работе [1]. Предлагаемый метод базируется на нескольких базовых принципах из других способов анализа ПО. В первую очередь это принцип отказа он полного вычисления состояний программы (широко применяется в абстрактной интерпретации [3–5] для того, чтобы можно было осуществить анализ ПО за приемлемое время). В случае абстрактного анализа это приводит к появлению ложных срабатываний (сигнал об ошибках в тех состояниях, где их нет) и отсутствию гарантии обнаружения ошибки, поэтому полный отказ от вычисления состояний не является приемлемым. Второй принцип – анализ исходного кода программ на основе подходов, принятых в символьной верификации. Рассмотрим данные принципы подробнее.

```
int x = 1;
int m = read();
for (; m == 1; m--) {
      x = m * x;
}
```

Рис. 1. Программное представление блок-схемы на рис. 3 в [3]

Рассмотрим пример на рис. 3 из [3]. Данный пример использовался для иллюстрации применения приемов абстрактной интерпретации для вычисления знака переменной в программе. Тот же самый пример в нотации программных кодов на языке С приведен на рис. 1. В анализ данной программы в разных подходах вкладывают разный смысл.

Например, в статическом анализе (и приложениях абстрактной интерпретации) выделяются домены ошибок и выполняется анализ для их выявления. При этом выделить домены для специфических ошибок в контексте конкретного приложения достаточно трудозатратно. В методах проверки моделей анализ выполняется для доказательства корректности формул темпоральной логики [7, 8], т. е. выполняется проверка состояний на конкретных пользовательских значениях. С нашей точки зрения, правильным подходом является комбинированный подход, когда часть ошибок выявляется автоматически (обычно это сугубо «технические» ошибки: выход за пределы цикла, изменение итерируемого массива во время цикла и т. д.) и специфические ошибки для конкретной задачи проверки, которые определяются разработчиком или тем специалистом, который проводит анализ кода. Например, такой проверкой могла быть корректность функции assert ($x > 100$) после цикла в программе на рис. 1. Далее рассмотрим пример с учетом функции assert после цикла.

Вторым аспектом при анализе программы являются пути, по которым может пройти программа. Следует заранее определить, что в данной работе рассматриваются только однопоточные программы, поэтому путями называются варианты исполнения программы, в которых граф управления имеет последовательный вид. О других подходах к преобразованию графа управления говорилось в [1, 9, 10, 13, 14, 16]. Определение всех возможных путей программы важно для определения всех возможных значений переменной в состоянии. В данном примере есть только один путь, по которому может передаваться управление программы. Кроме путей в программе также большую роль играют значения переменных и то, в каком виде они представлены. Поскольку основным аспектом поиска является выявление списка возможных значений переменной в состоянии, то важно определить, в какой форме будет представлен такой список. Он будет зависеть от пути, по которому выполняется программа, и значений других переменных. Для определения значений можно использовать как конкретные значения, так и диапазоны значений переменных. В случае, когда переменная может принимать любое значение, ее будем обозначать как Any (любое).

На рис. 2 представлена блок-схема программы из рис. 1. Данная блок-схема отличается от блок-схемы на рис. 3 из работы [3] тем, что в ней цикл представлен в виде замкнутого самого на себя перехода. Это новое представление позволяет сделать вывод, что определение возможных значений переменных в состояниях выполняется тривиально за исключением того, что непонятным является, сколько раз выполняется переход в цикле. Учитывая, что любой цикл можно свернуть, получим блок-схему на рис. 3. После предобразования

исходной блок-схемы (см. рис. 3) получаем блок-схему (программу), которая не содержит ветвления и для которой не составляет особой проблемы определить возможные значения переменных в любых состояниях, в том числе и в состоянии assert. Такое преобразование программы получилось за счет того, что, во-первых, мы рассматриваем один из вариантов прохождения программы; во-вторых, полученная функция f(X, m) является скрытой формой записи цикла, который далее должен быть заменен рядом (в общем виде не известной, но конечной длины).

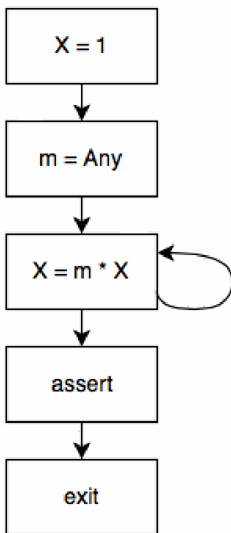

Рис. 2. Блок-схема программы из рис. 1

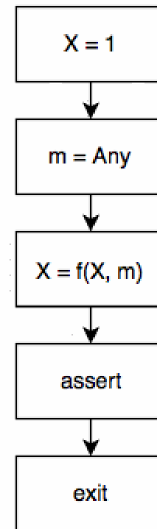

Рис. 3. Измененная блок-схема программы из рис. 1

Важным аспектом является характер роста числа вариантов исполнения программы. Важно чтобы его рост не имел экспоненциального характера, и это должно быть рассмотрено отдельно. Возможно применение алгоритмов, схожих с [14]. Необходимо также рассмотреть дополнительный пример, в котором в цикле содержится более сложный код. На рис. 4 показана программа с несколькими операторами, проанализируем несколько итераций цикла. Перед циклом переменные будут иметь значения: m_0 = Any; $x_0 = 1$; a_0 = Any. На первой итерации цикла будут следующие операции: $a_1 = m_0 + x_0$; $x_1 = m_0 + a_1$; $m_i = m_{i-1} - 1$. Очевидно, что переменные зависят друг от друга, но тем не ме-

нее данный цикл также можно привести к виду: $a = f(m, x)$; $x = g(m, a)$; $a = m_n + x_{n-1}$; $x = m_n + m_n + m_{n-1} + m_{n-2} + \ldots + m_0 + x_0$.

Стоит отметить, что в данном случае характер изменения итерируемой переменной был достаточно просто (декремент переменой m на каждой итерации), но в общем случае он может быть иметь более сложный характер: пропуск некоторых значений, произвольное изменение итерируемого значения (например, функцией rand). Поэтому необходимо применять символьную за-

Отдельно стоит сказать о массивах в циклах и вне их. Основным вариантом использования массивов является запись или чтение в/из массива в цикле. Массив по своей сути представляет удобную запись (и адресацию) определенного множества переменных, поэтому к массивам также можно применять те же самые принципы к сворачиванию цикла.

```
int x = 1;
int m = read();
int a = read();
for (; m == 1; m--) {
      a = m + x;
      x = m * a;
  }
```

Рис. 4. Модифицированный пример программы из рис. 1

2. АЛГОРИТМЫ МЕТОДА

В данном разделе будут рассмотрены алгоритмы анализа вышеприведенного метода и семантика их записи. Различные методы анализа предлагают разные семантики для записи алгоритмов анализа. Например, модель Крипке [7, 8] является основой решения задач проверки моделей и состоит из нескольких множеств; П. Коусот (P. Cousot) для описания алгоритмов абстрактной интерпретации предложил семантику [3–5], которую условно можно разделить на несколько частей. Далее рассмотрим общую часть для описания любой программы. Согласно данной нотации [3, 5], все узлы программы делятся на «Вход», «Присвоение», «Тест» и «Переход», также каждый узел имеет входные и выходные узлы: n-succ, n-pred: Nodes → 2^{Nodes}. Переходы являются множеством Nodes x Nodes: Arcs = {<m, n> | (m ∈ Nodes) and (n ∈ n-succ(m))}. При помощи такой семантики достаточно просто построить программу в виде направленного графа:

```
next: States → States
next(<m,n>, ρ) = case n in
  Assignments: (a-succ(n), ρ[val[expr(n)]ρ/id(n)])
  Tests: cond(val[test(n)]ρ, (a-succ-t(n), ρ), (a-succ-f(n), ρ))
```

```
    Junctions: (a-succ(n), ρ)
    Exits: (<m,n>, ρ)
  esac
```

где id, expr, test, val – функции выборки, ρ[v/I] где ρ∈ Env, I∈ Ident, v∈ Values – результат обновления значения ρ в I со значением v.

Такую семантику предлагается использовать для описания алгоритмов анализа программ рассматриваемым методом. То есть исходя из раздела «Основы метода анализа» необходим алгоритм анализа графа, в котором |a-succ(m)| = |a-pred(n)| = 1. В рассматриваемом случае данное описание избыточно, так как мы рассматриваем только одну ветку программы и Junction и Test узлы не нужны. А рассматривать можно только Assignment, но не с целью перезаписи результата, а с целью определения возможных значений. Тогда пусть v для I в ρ будет значением переменной I в «памяти» ρ, а функция o(I) = options(n, I), где o – Values, I ∈ Ident, n: States означает множество состояний, в которых может быть переменная после исполнения выражения expr(n). Тогда алгоритм будет иметь следующий вид:

```
  next: States → States
  next(<m,n>, ρ) = case n in
    Assignments: o(id(n)) = options(n, id(n))
    Exits: (<m,n>, ρ)
  esac
```

Теперь необходимым становится, чтобы функция options была определена для каждого выражения expr(n). Несколько примеров такого определения показаны в таблице, а для остальных операций легко определить такие функции по аналогии.

Пример определения функции options

Выражение	Определение функции

`V = Op1 * Op2;`	`I = id(n); R = {}`
	`for v1 in o(Op1)`
	`  for v2 in o(Op2):`
	`    R.add(v1*v2)`

	`R`

Окончание таблицы

Выражение	Определение функции

`V = !Op1;`	`I = id(n); R = {}`
	`for v1 in o(Op1)`
	`  R.add(!v1)`
	`R`

3. О ПОИСКЕ ВОЗМОЖНЫХ ПУТЕЙ ИСПОЛНЕНИЯ

В работе [1] предлагается метод анализа кода программ на ошибки, где одним из первых шагов является поиск возможных путей исполнения программы. В данном разделе рассматривается, каким образом определять возможные пути исполнения программы. Объектом исследования являются операторы в однопоточных программах, в которых исполнение программы может быть выполнено по-разному: условный оператор и оператор цикла, а также различные их варианты.

Перед тем как начать рассматривать, какие возможные пути содержатся в том или ином участке кода, введем понятие оператора возможности: Pos(st) – оператор возможности (от possibility) принимает один аргумент, для которого определяется возможность того, что результатом выражения st будет истина. В выражениях могут использоваться переменные, значения которых не известны или известны частично. Например, Pos(4 > 3) = {true} означает, что выражение 4 > 3 истинно при любых значениях. Pos(4 > a) = {a <= 4} также означает, что есть вероятность обращения выражения в истину. При S = {a} = = {7}, Pos(4 > a) = {}, т. е. нет таких значений, когда выражение обращается в истину. Отдельно стоит рассмотреть случай, когда st – составное выражение. Pos((a > 3 || b < 10) && t) следует вычислять согласно правилам логики. Также интересно знать значения переменных, при которых возможен переход. Для этого предлагается использовать оператор

Первым рассмотрим условный оператор, который можно записать как R(cond; op1; op2). Для условного оператора есть всего два возможных варианта: либо исполняется оператор op1, либо оператор op2 в зависимости от истинности условия cond. Стоит заметить, что условие cond также может быть

составным, что означает, что существует несколько возможных путей, при которых исполняется оператор op1. Найти возможные значения, при которых выполняется условный оператор, можно при помощи оператора Pos: Pos(cond).

Более сложным случаем является исполнение цикла, который можно представить как R(cond, op1, op2, op3), где op1 – выполняемый перед циклом оператор, op2 – оператор, выполняемый после цикла, op3 – оператор тела цикла и cond – условие завершения цикла. С самим циклом есть несколько вариантов исполнения: 1) условие cond ложно, тогда op2 и op3 не исполняются; 2) условие cond истинно, тогда выполняется оператор цикла до тех пор, пока cond не будет ложным. При более детальном рассмотрении циклов видно, что в общем случае число итераций может быть любым. При этом если в цикле есть условный оператор или еще вложенные циклы, то возникает ситуация, при которой посчитать количество вариантов выполнения программы становится нетривиальным. Далее рассмотрим несколько примеров поиска возможных вариантов исполнения кода в программах с циклами и выявим общие зависимости.

```
for (i = 0 ; i < 100 ; i++) {
  a = a + i;
}
```

Рис. 5. Пример программы с циклом известной длины

```
for (i = 0 ; i < 100 ; i++)
{
  a &= i;
}
```

Рис. 6. Пример программы с логическим выражением в цикле

```
for (i = 0 ; i < 100 ; i++) {
  if (a > 10) {
    b += 10;
  }
}
```

Рис. 7. Пример программы с циклом с условным оператором 1

```
for (i = 0 ; i < 100 ; i++)
{
  if (i > 15) {
    b += 10;
  }
}
```

Рис. 8. Пример программы с циклом с условным оператором 2

```
for (i = 0 ; i < 100 ; i++) {
  if (a[i] > 10) {
    b[i] += 10;
  }
}
```

```
for (i = 2; i < 100 ; i++) {
  a[i] += a[i - 2];
}
```

Рис. 9. Пример программы с циклом с массивом в условном операторе

Рис. 10. Пример программы с циклом с операциями над массивом

```
for (i = 0 ; i < 100 ; i++) {
  for (j = 0; j < i; j++)
    if (a[i] > 10) {
      b[i] += 10;
    }
  }
}
```

Рис. 11. Пример программы с вложенными циклами

```
for (i = 0 ; i < 100 ; i++)
{
  a[i] += i;
  for (j = 0; j < i; j++)
    if (a[i] > 10) {
      b[i] += a[i];
    }
  }
}
```

Рис. 12. Пример программы с зависимыми переменными между циклами

На рис. 5–12 приведены участки программы с различными вариантами циклов. Найдем возможные варианты их исполнения:

– рис. 5, 6: согласно вышеприведенному анализу для цикла в общем случае возможны варианты исполнения. Так как $S = \{i\} = \{0\}$ и $Pos(i< 100) = \{i>=0, i < 100\}$, то в данном случае возможен только лишь один вариант исполнения программы;

– рис. 7, 8: на данных рисунках представлены циклы с условными операторами. Возможны два варианта: когда срабатывает условный оператор и когда не срабатывает. Если не рассматривать варианты преобразования цикла в целях упрощения [17–20], то можно сказать, что остается два варианта из параграфа выше, причем вариант исполнения цикла делится на два варианта в зависимости от срабатывания условного оператора;

– рис. 9: в данном рисунке представлен цикл с условным оператором, где в условии содержится итерируемый массив. Несмотря на то что в примере указано максимальное число итераций, рассмотрим также случай с произвольным количеством итераций. Для обоих случаев возможны сценарии, когда цикл не исполняется. Также в первом варианте возможны два сценария в зависимости от условного оператора. В отличие от прошлого случая условный оператор зависит от итерации цикла: $a[i] > 10$;

– рис. 10, 11: несмотря на зависимости между переменными, возможных вариантов всего два;

– рис. 12: возможны следующие варианты исполнения программы: два варианта исполнения внешнего цикла и такие же варианты исполнения внутреннего цикла, как и для программы на рис. 9.

Вышеприведенные варианты показывают, что в однопоточных приложениях число вариантов исполнения программы зависит от числа операторов и их вида, а также от того, какие значения могут принимать переменные. В дальнейшем будем использовать оператор Pos для определения возможности исполнения оператора на множестве значений. Причем для определения всех возможных путей можно использовать известные алгоритмы обхода графа программы.

ВЫВОД

В данной работе продолжено развитие метода анализа ПО, изначально предложенного в [1]. В частности, в статье приводится описание того, как выполнять разбиение графа программы на множество «линейных» участков, которые можно достаточно легко проанализировать. Приведена формальная семантика, предложенная П. Коусотом для описания абстрактной верификации, и показано, каким образом она может быть применена для формализованного описания алгоритмов анализ ПО.

Остаются открытые вопросы для отдельного исследования о характере роста числа «линейных» отрезков и числа вариантов значений переменных. Также темой отдельного исследования является сравнение различных семантик описания программ [1, 3, 5] и выбор наиболее подходящей семантики для дальнейшего расширения рассматриваемого метода.

СПИСОК ЛИТЕРАТУРЫ

1. *Воевода А.А., Романников Д.О.* Способы представления программ и их анализ // Сборник научных трудов НГТУ. – 2014. – № 3 (76). – С. 81–98.

2. *Романников Д.О.* О поиске входных интервалов // Сборник научных трудов НГТУ. – 2014. – № 1 (75). – С. 140–145.

3. *Abramsky S., Hankin C.* An introduction to abstract interpretation. – URL: https://www.cs.virginia.edu/~weimer/2007-615/reading/AbramskiAI.pdf (accessed 20.12.2014).

4. *Cousot P., Cousot R.* A gentle introduction to formal verification of computer systems by abstract interpretation // Logics and Languages for Reliability and

Security. NATO Science Series III: Computer and Systems Sciences. – Amsterdam: IOS Press, 2010. – P. 1–29.

5. *Глухих М.И., Ицыксон В.М., Цесько В.А.* Использование зависимостей для повышения точности статического анализа программ // Моделирование и анализ информационных систем. – 2011. – Т. 18, № 4. – С. 68–79.

6. *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.

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

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

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

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.

11. *Воевода А.А., Марков А.В., Романников Д.О.* Разработка программного обеспечения: проектирование с использованием UML диаграмм и сетей Петри на примере АСУ ТП водонапорной станции // Труды СПИИРАН. – 2014. – Вып. 3 (34). – С. 218–232.

12. *Романников Д.О.* Нахождение ошибок обращения к несуществующим элементам массива на основании результатов анализа сети Петри // Сборник научных трудов НГТУ. – 2012. – № 1 (67). – С. 115–120.

13. *Марков А.В., Романников Д.О.* Алгоритм трансляции диаграммы активности в сеть Петри // Доклады Академии наук высшей школы Российской Федерации. – 2014. – № 1 (22). – С. 104–112.

14. *Воевода А.А., Романников Д.О.* Редуцирование пространства состояний сети Петри для объектов из одного класса // Научный вестник НГТУ. – 2011. – № 4 (45). – С. 146–150.

15. *Романников Д.О., Марков А.В., Зимаев И.В.* Обзор работ, посвященных разработке ПО с использованием UML и сетей Петри // Сборник научных трудов НГТУ. – 2011. – № 1 (63). – С. 91–104.

16. *Марков А.В.* Анализ отдельных частей дерева достижимости сетей Петри // Сборник научных трудов НГТУ. – 2013. – № 3 (73). – С. 58–74.

17. *Воевода А.А., Марков А.В.* Рекурсия в сетях Петри // Сборник научных трудов НГТУ. – 2012. – № 3 (69). – С. 115–122.

18. *Воевода А.А., Марков А.В.* Тестирование UML-диаграмм с помощью аппарата сетей Петри на примере разработки по для игры «Змейка» // Сборник научных трудов НГТУ. – 2010. – № 3 (61). – С. 51–60.

19. *Воевода А.А., Зимаев И.В.* Верификация workflow-моделей с применением сетей Петри // Научный вестник НГТУ. – 2010. – № 4 (41). – С. 151–154.

20. *Воевода А.А., Саркенов Д.О., Хассоунех В.* Моделирование протоколов с учетом времени на цветных сетях Петри // Сборник научных трудов НГТУ. – 2004. – № 3 (37). – С. 133–136.

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, старший преподаватель кафедры автоматики Новосибирского государственного технического университета. Основные направления научных исследований: формальная верификация, проверка моделей. Имеет 31 публикацию. E-mail: rom2006@gmail.com

About the method of program analysis*

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. E-mail: ucit@ucit.ru*

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

The paper is devoted to analysis of a method of software analysis that based on principles of detection of all possible variables values in states and finding out software errors with help of checking verification formulas on all possible values in state. Some examples in this work show base principles of offered approach of software analysis, i.e.: example of a program with loop that show how to execute folding of whole program graph to lineal piece of a program and then perform an analysis. The second principle of discussed method is lineal software program analysis, i.e. progressive analysis of the each program branch. Wherein separate analysis of each branch might be performed from the disjoin place. Multiple examples of dividing main graph of a program to branches are shown. The main feature of this dividing is a keeping of a small amount of

* *Received 16 July 2014.*

Work is executed at financial support of the Minobrnauka Russia state job № 2014/138, the theme of the project "New patterns, models and algorithms for breakthrough methods of control of technical systems based on high results intellectuality activity".

branches. Formal semantic of a program description is shown. This semantic allows describing algorithms of program analysis. Initial semantic was simplified in part of graph analysis: excessive nodes were removed; and function for searching possible variable values was added. Shown how to determine such function for binary and unary operators. For other operators this function might be obtained similarly.

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

REFERENCES

1. Voevoda A.A., Romannikov D.O. Sposoby predstavleniya programm i ikh analiz [Methods of program representation and analysis]. *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2014, no. 3 (76), pp. 81–98.
2. Romannikov D.O. O poiske vkhodnykh intervalov [On the search for input intervals]. *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2014, no. 1 (75), pp. 140–145.
3. Abramsky S., Hankin C. An introduction to abstract interpretation. Available at: https://www.cs.virginia.edu/~weimer/2007-615/reading/AbramskiAI.pdf (accessed 20.12.2914)
4. Cousot P., Cousot R. A gentle introduction to formal verification of computer systems by abstract interpretation. Logics and Languages for Reliability and Security. NATO Science Series III: Computer and Systems Sciences. Amsterdam, IOS Press, 2010, pp. 1–29.
5. Glukhikh M.I., Itsykson V.M., Tses'ko V.A. Ispol'zovanie zavisimostei dlya povysheniya tochnosti staticheskogo analiza programm [The use of dependencies for improving the precision of program static analysis]. *Modelirovanie i analiz informa-tsionnykh sistem – Modeling and Analysis of Information Systems*, 2011, vol. 18, no. 4, pp. 68–79.
6. 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.
7. 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.).
8. 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.
9. Horwitz S., Reps T., Binkley D. Interprocedural slicing using dependence

graphs. *ACM Transactions on Programming Languages and Systems*, 1990, vol. 1, iss. 12, pp. 26–60.

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.

11. Voevoda A.A., Markov A.V., Romannikov D.O. Razrabotka programmnogo obespecheniya: proektirovanie s ispol'zovaniem UML diagramm i setei Petri na primere ASU TP vodonapornoi stantsii [Software development: software design using UML diagrams and Petri nets for example automated process control system of pumping station]. *Trudy SPIIRAN – SPIIRAS Proceedings*, 2014, iss. 3 (34), pp. 218–232.

12. Romannikov D.O. Nakhozhdenie oshibok obrashcheniya k nesushchestvuyushchim elementam massiva na osnovanii rezul'tatov analiza seti Petri [Finding errors or nonexistent array's elements based on the results of the analysis Petri nets]. *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2012, no. 1 (67), pp. 115–120.

13. Markov A.V., Romannikov D.O. Algoritm translyatsii diagrammy aktivnosti v set' Petri [Algorithm of automatic conversion of the activity diagram into Petri-net structure formats]. *Doklady Akademii nauk vysshei shkoly Rossiiskoi Federatsii – Proceedings of the Russian higher school Academy of sciences*, 2014, no. 1 (22), pp. 104–112.

14. Voevoda A.A., Romannikov D.O. Redutsirovanie prostranstva sostoyanii seti Petri dlya ob"ektov iz odnogo klassa [Reducing the state space of Petri nets for objects of one class]. *Nauchnyi vestnik NGTU – Science Bulletin of Novosibirsk State Technical University*, 2011, no. 4 (45), pp. 146–150.

15. Romannikov D.O., Markov A.V., Zimaev I.V. Obzor rabot, posvyashchennykh razrabotke PO s ispol'zovaniem UML i setei Petri [The review of works devoted to development of the software with use UML and Petri nets]. *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2011, no. 1 (63), pp. 91–104.

16. Markov A.V. Analiz otdel'nykh chastei dereva dostizhimosti setei Petri [Analysis of individual pieces of wood reachability Petri nets]. *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2013, no. 3 (73), pp. 58–74.

17. Voevoda A.A., Markov A.V. Rekursiya v setyakh Petri [The concepts recursion in Petri nets]. *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2012, no. 3 (69), pp. 115–122.

18. Voevoda A.A., Markov A.V. Testirovanie UML-diagramm s pomoshch'yu apparata setei Petri na primere razrabotki PO dlya igry "Zmeika" [About testing

UML-diagrams by means of the device of Petri nets on the example of software engineering for game «Snake»]. *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2010, no. 3 (61), pp. 51–60.

19. Voevoda A.A., Zimaev I.V. Verifikatsiya workflow-modelei s primeneniem setei Petri [Verification of workflow-models using Petri-nets] *Nauchnyi vestnik NGTU – Science Bulletin of Novosibirsk State Technical University*, 2010, no. 4 (41), pp. 151–154.

20. Voevoda A.A., Sarkenov D.O., Khassounekh V. Modelirovanie protokolov s uchetom vremeni na tsvetnykh setyakh Petri [Protocol modeling with regards of time on colored Petri nets] *Sbornik nauchnyh trudov NGTU – Transaction of Scientific Papers of Novosibirsk State Technical University*, 2004, no. 3, pp. 133–136.