СБОРНИК **НАУЧНЫХ ТРУДОВ НГТУ. – 2014. – № 3(77).** – 81–98

УДК 62-50:519.216

СПОСОБЫ ПРЕДСТАВЛЕНИЯ ПРОГРАММ И ИХ АНАЛИЗ*

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. 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*

В данной работе рассматриваются различные способы представления программ: от графов и логики Хоара до сетей Петри, также предложен новый способ представления программ, который заключается в записи программы в символах (переменных) и составлении зависимостей между переменными. Предлагаемый способ позволяет значительно снизить число ветвлений при анализе в программе: если при обычном анализе участка программы, в котором условный оператор находится в цикле, состоящем из n итераций, необходимо рассматривать $2n$ вариантов, то предлагаемый способ позволяет значительно снизить число вариантов. Одним из промежуточных результатов предлагаемого способа является система отношений между переменными, с помощью которой можно проверить выполнимость различных условий в коде программы. Результатом такой проверки является подтверждение того, что условие выполняется, или контрпример, показывающий, при каких данных условие может быть не выполнено. Предлагаемый способ был опробован на следующих примерах: с использованием условных операторов, с использованием циклов и задачи коммивояжёра. Положительный результат был получен для всех задач, за исключением последней, для которой будут выполнены дополнительные исследования.

Ключевые слова: тестирование, входные интервалы, формальная верификация, динамическая верификация, верификация, проверка моделей, модели программного обеспечения, графы

ВВЕДЕНИЕ

Для анализа программ требуется их предварительное преобразование из исходных кодов. Причем полученное представление в значительной степени влияет на дальнейший анализ, а именно: на его скорость, потребление памяти,

* Статья получена 1 августа 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».

максимальное количество поддерживаемых состояний и др. Для разных видов анализа [1–6] используются разные виды представлений [1–6]. Далее в работе будут рассмотрены и проанализированы следующие представления программ: модель Крипке [1], ориентированный (цикличный/ацикличный) граф [2–4], логика Хоара [5], сети Петри [6–8] на примере небольшого фрагмента программы, представленной на рис. 1. На данном рисунке S_1, S_2 … S_9 – состояния программы, где каждое состояние – это множество переменных, т. е. $S_i = \{v_1, v_2, \ldots, v_k\}$. На описании v_k стоит остановиться отдельно. Можно рассматривать его несколькими способами: 1) $v_k = Val$ – как значение переменной в данном состоянии; 2) $v_k = \{Val_1, Val_2, \ldots, Val_j\}$ – как множество возможных значений переменной в данном состоянии.

Данная статья далее организована в следующем порядке: в разделе ПРЕДПОСЫЛКИ указаны связанные работы и различные описания ПО, которые в достаточной мере распространены в индустрии разработки ПО; задача анализа ПО в формальном представлении показана в разделе АНАЛИЗ. В разделе ОПИСАНИЕ ПРОГРАММЫ предлагается способ представления программы в удобном для дальнейшего его анализа. В разделе ОПИСАНИЕ ПРОГРАММЫ С МАССИВАМИ предложен способ записи программы, содержащей циклы и массивы, а также примеры такой записи. В разделе ЗАДАЧА КОММИВОЯЖЕРА приведен заключительный пример применения предлагаемого способа.

1. ПРЕДПОСЫЛКИ

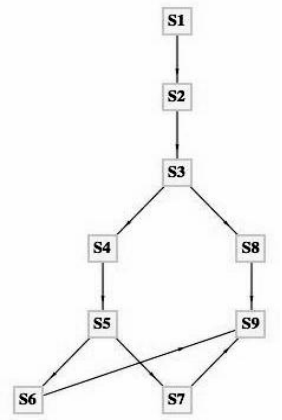

Рис. 1. Пример программы, представленной в виде графа

Традиционными способами представления программы являются графы (см. рис. 1), которые в достаточной мере позволяют информативно представить программу или ее отдельные участки. В графах можно представить программу как на ее конкретных значениях, так и на абстрактных переменных. С графами в какой-то мере связаны все известные способы представления программы. К недостаткам представления программы в виде графа (в графическом представлении) можно отнести его громоздкость на больших программах. Но при представлении графа не в графическом виде этот недостаток нивелируется и графы широко применяются при описании ПО [1–6]. Так, например, в описании модели Крипке из четверки (S, S_0, R, L) в виде графа представлен только R, но при этом

в разные этапы развития проверки моделей он имел разное представление: от графа переходов (или функции переходов) до упорядоченных двоичных разрешающих диаграмм (OBBD) [1]. Идея OBBD достаточно интересна для ее использования при анализе программ. Однако у авторов нет явного понимания, как наилучшим образом переиспользовать достоинства OBBD. Из-за того, что терминальные вершины в OBBD имеют значения «1» и «0», нет возможности представить значения переменных программы. При использовании логики Хоара программа представляется с помощью функций $\{P\}\ C\ \{Q\}$, где P – предусловие операции, C – операция и Q – постусловие операции. Данная программа широко применяется при дедуктивной верификации. При таком описании программы существуют некоторые сложности с выбором инварианта, а также сложность ее применения к программам, в которых содержатся циклы и указатели. Сети Петри широко применяются в разработке программного обеспечения [6–8] в части анализа параллельных систем.

Таким образом, общим недостатком всех вышепредставленных способов представления программ является анализ программы на одном наборе начальных значений переменных, тогда как множество ошибок в программе связано с неучтенными начальными условиями или их комбинацией. Далее представлен метод, который позволяет учесть такие особенности.

2. АНАЛИЗ

С точки зрения разработчика, для анализа ПО достаточно проверки условия какой-нибудь переменной. Темпоральная логика, которая предлагается в теории проверки моделей, достаточно мощный инструмент, но он является избыточным, так как любое его условие можно выразить через одну или несколько переменных в коде программы. Таким образом, описание проверок для анализа имеет вид

$$cond(v_k) = true,\ v_k \in S_j, \tag{1}$$

где *cond* – функция, определяющая проверяемое условие; v_k – переменная, для которой выполняется анализ в состоянии S_j. Очевидно, что для проверки условия (1) необходимо знать все значения, которые может принимать v_k.

Таким образом, задача анализа ПО (1) сводится к следующему:

1) к задаче определения возможных значений переменной v_k в S_j: $\{Val_1, Val_2, \ldots, Val_q\}$ и проверке выполнимости проверяемого условия;

2) к поиску возможных путей P_i для определения значений v_k и возможности поиска ошибочных сценариев.

Рассматривая вышеприведенное представление, можно сказать, что проверка (1) на модели Крипке выполняется достаточно легко, но само составление тотального множества переходов R и множества состояний S является более трудоемкой задачей.

3. ОПИСАНИЕ ПРОГРАММЫ

Описания, приведенные в разделе ПРЕДПОСЫЛКИ, в полной мере не удовлетворяют потребности в описании ПО для анализа (1). Рассмотрим программу на рис. 1 подробнее. Целью анализа является определение значений переменной *v* в *S*. Также в программе есть другая полезная информация, которая не попадает в выражение для *S*. Например, зная конкретное значение для переменной Val_1 из выражения для *S*, могла остаться зависимость $Val_1 > Val_2$. Таким образом, общее выражение для *S* следует записать так: S: {{Val_1, [$Cond_1$]}, {Val_2, [$Cond_2$] … {Val_q, [$Cond_q$]}}, где Var_i – переменные, $Cond_i$ – условия (зависимости), которые накладываются на переменную Var_i.

Необходимо отметить, что в данной работе будем рассматривать программу как набор логических и арифметических операций, а также операций контроля управления. Многие функции, свойственные императивным языкам программирования (работа с файлами, сетью, принтерами и т. д.), не рассматриваются в данной работе, так как для этого необходимо получить их полное представление вплоть до низкоуровневых операций, т. е. до примитивов. С другой стороны, нерассматриваемые функции также реализованы с помощью логических и арифметических операций и обращений к функциям более нижнего уровня вплоть до работы со специализированными микросхемами[1]. Очевидно, что значение каждой переменной зависит от пути, по которому пройдет управление программы, а путь, в частности, от начальных условий. Поэтому целесообразно анализировать каждый оператор с учетом всех его возможных значений. Основной идеей нахождения всех возможных значений в состоянии является определение их для оператора *n* и последующее их определение для *n* + 1 на основании возможных значений предыдущих.

Как сказано выше, для проверки (1) достаточно знать список возможных значений. Но не всегда есть возможность получить такой список для конкретных значений: например, если для программы входная целочисленная переменная действительна во всем своем диапазоне или входной переменной является произвольная строка, тогда осуществить расчет всех возможных значе-

[1] Исходя из этой логики можно сделать вывод, что применяемые в данной работе правила должны сработать для всех функций. Но для этого необходимо расширить описание системы: например, для рассмотрений функций работы с файлами (*fopen*, *fseek* и др.) без изменения исходного кода программы необходимо реализовать виртуальную файловую систему с помощью переписанных библиотек и использовать их при анализе. Например, при вызове функции *fopen* будет создаваться файл не на файловой системе, а в виртуальной памяти, где файл будет представлен в виде массива *char* элементов, с которыми есть представление, как работать. Аналогично для функций для работы с сетью и другими.

ний в каком-либо состоянии невозможно. Рассмотрим, каким образом можно оценивать такие переменные для различных операторов.

В [9] показано, как выполнять такой анализ для случая с условными операторами и определенными переменными. Рассмотрим программу на рис. 2: переменные *a*, *b*, *c* в ней не определены и подразумевается, что они имеют произвольные значения во всем диапазоне возможных значений. На рис. 2 приведен граф возможных переходов для вышеуказанной программы. Проанализируем поведение переменных *a*, *b*, *c* в состоянии S_q. В исследуемое состояние ведут четыре пути:

$$\begin{aligned} P_1 &: S_0 S_1 S_3 S_4 S_q \\ P_2 &: S_0 S_2 S_3 S_4 S_q \\ P_3 &: S_0 S_1 S_3 S_5 S_q \\ P_4 &: S_0 S_2 S_3 S_5 S_q \end{aligned} \qquad (2)$$

Исходя из (2) можно найти множество значений переменных в состоянии S_q путем вычислений множества значений переменных на каждом пути и последующего их объединения.

```
S0:   a,b,c;
      if (a > b) {
S1:     c = 10;
      } else {
S2:     c = 12;
      }
S3:   b = 2*c;
      if (b == 3*c) {
S4:     c = 18;
      } else {
S5:     b = 22;
      }
Sq:
```

Рис. 2. Пример программы с условными переходами

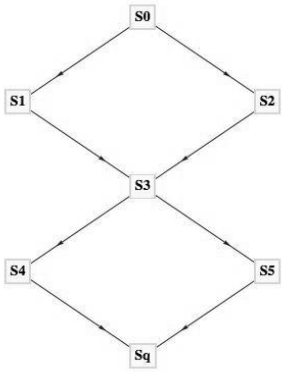

Рис. 3. Граф переходов для программы из рис. 2

P_1 : S_0 = {*Any*, *Any*, *Any*}, S_1 = {$a > b$, $b <= a$, 10}, S_3 = {$a > b(S_1)$, 20, 10}, S_4 = {$a > b(S_1)$, 20, 18};

P_2 : S_0 = {*Any*, *Any*, *Any*}, S_2 = {$a <= b$, $b > a$, 12}, S_3 = {$a <= b(S_2)$, 24, 12}, S_4 = {$a > b(S_2)$, 24, 18};

P_3 : S_0 = {*Any*, *Any*, *Any*}, S_1 = {$a > b$, $b <= a$, 10}, S_3 = {$a > b(S_1)$, 20, 10}, S_5 = {$a > b(S_1)$, 22, 10};

P_4 : S_0 = {*Any*, *Any*, *Any*}, S_2 = {$a <= b$, $b > a$, 12}, S_3 = {$a <= b(S_2)$, 24, 12}, S_4 = {$a > b(S_1)$, 22, 18}.

После получения значений переменных на каждом пути необходимо объединить их для получения всех возможных значений в состоянии S_q = {{$a > b(S_1)$, 20, 18}, {$a > b(S_2)$, 24, 18}, {$a > b(S_1)$, 22, 10}, {$a > b(S_1)$, 22, 18}}. Теперь, вычислив (1) на множестве переменных в состоянии S_q, можно *гарантировать корректность программы* в данном состоянии по формуле (1). Необходимо отметить, что вычисление всех путей в программе не обязательно, оно применялось в вышеприведенном примере из-за того, что было необходимо проанализировать все переменные в последнем состоянии. При исследовании части переменных можно воспользоваться графом зависимости [2–4], в котором будет содержаться часть путей.

Рассмотрим пример нахождения возможных вариантов для программы, содержащей циклы (рис. 4) и соответствующий ей граф (рис. 5). Для анализа данной программы не получится построить все пути. Так как переменная *a* может принимать любые значения, то цикл можно считать бесконечным. Но с учетом того, что в данной работе рассматриваются только логические и арифметические операторы, то согласно [10] циклы можно преобразовать. В [10] такое преобразование использовалось для удобства распараллеливания вычислений, а в данной работе мы будем использовать такой прием для упрощения записи и анализа программы.

```
S0:  a, b, i;
S1:  for (; i < a; i++) {
S2:    b++;
       if (a > 100) {
S3:      b -= a;
       }
     }
Sq:
```

Рис. 4. Пример программы с циклом

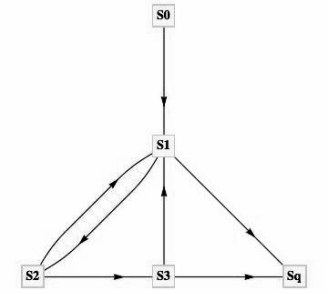

Рис. 5. Граф переходов для программы из рис. 4

На рис. 6 представлена программа, полученная после преобразования программы на рис. 4. В данной программе цикл с условным оператором преобра-

зован в два цикла без условных операторов. Необходимо отметить, что данный прием может быть использован сколько угодно раз, до тех пор пока программа не будет преобразована до циклов без условных операторов. После выполнения такого преобразования становится элементарным нахождение путей, по которым может пройти программа.

```
S0:  a, b, i;
     if (a > 100) {
S1:    for (; i < a; i++) {
S2:      b++;
S3:      b -= a;
       }
     } else {
S4:    for (; i < a; i++) {
S5:      b++;
       }
     }

  Sq:
```

Рис. 6. Пример программы с циклом

Покажем эти пути. Для этого предварительно запишем циклы в состояниях S_1 и S_4 в виде преобразованных состояний S_{1c} и S_{4c}:

$$b(S_2) = b(S_1) + 1; \qquad (3)$$

$$b(S_3) = b(S_2) - a;$$

$$b(S_{1c}) = (b(S_2) - a) * n = (b(S_1) + 1 - a) * n;$$

$$b(S_5) = b(S_4) + 1;$$

$$b(S_{4c}) = b(S_4) * n,$$

где $b(S)$ – значение переменной b в состоянии S. После получения (3) преобразованных состояний S_{1c} и S_{4c} можно записать пути, по которым проходит программа:

$$P_1 : S_0 S_{1c} S_q; \qquad (4)$$

$$P_2 : S_0 S_{4c} S_q.$$

После определения путей (4) легко получить список значений в состоянии $S_q = \{a, b, i, n\}$.

$P_1 : S_0 = \{Any, Any, 0, Any\}$, $S_{1c} = \{Any, (b + 1 - a) * n, 0, Any\}$, $S_q = \{Any, Any, 0, Any\}$;

$P_2 : S_0 = \{Any, Any, 0, Any\}$, $S_{1c} = \{Any, b * n, 0, Any\}$, $S_q = \{Any, Any, 0, Any\}$.

Таким образом, любой цикл можно преобразовать к множеству независимых циклов, которые легко преобразовываются к одному состоянию. Также это позволяет значительно уменьшить количество веток, по которым может проходить программа (если в цикле находится условный оператор, то число путей в программе громадно). Подводя итог вышесказанному, можно вывести следующий алгоритм.

Алгоритм 1. Определение возможных значений переменных в произвольном состоянии программы

1. Программа $Pr = O_1O_2O_3 \ldots O_k$ как набор операторов O.

2. Составить размеченную $Pr\_labled = S_1S_2S_3 \ldots S_m$ как набор состояний S.

3. Выполнить преобразование циклов таким образом, чтобы циклы не содержали вложенных условных операторов. Результатом преобразования будет являться множество состояний S_c и $Pr\_labled' = S_1S_2S_3 \ldots S_j$, $S_c \in Pr\_labled'$.

4. Определить пути $P_1P_2P_3 \ldots P_{ii}$, по которым проверяемое состояние может быть достигнуто.

5. Вычислить возможные значения переменных согласно пути до искомого состояния.

6. Проверить выполнимость условия (1) на найденном в шаге 5 множестве.

Результатом шага 5 алгоритма 1 является набор переменных и множество значений в соответствии каждой переменной. Такое описание хорошо подходит для одиночных переменных, однако для описания массивов переменных как потенциально бесконечных сущностей его нужно рассмотреть отдельно.

4. ОПИСАНИЕ ПРОГРАММЫ С МАССИВАМИ

Описание массивов отличается от описания одиночных переменных из-за того, что массивы потенциально могут быть бесконечными. Например, в C++ абстракция над массивом vector или list может быть расширена до тех пор, пока есть свободная оперативная память или описание строки в C как указатель на переменную типа (char*), также будет неограниченной строкой.

```
S0: a,b,len,i,j
    for (i = 0; i < len; i++) {
      if (a[i] == 'a' &&
          a[i+1] == 'b' &&
          a[i+2] == 'c'
      )
S1:     a[i] = b[0];
S2:     a[i+1] = b[1];
S3:     a[i+2] = b[2];
      }
    Sq:
```

Рис. 7. Пример программы с циклом

Рассмотрим пример программы (рис. 7), в которой выполняются какие-то манипуляции с массивом данных. В программе проверяется, содержит ли входная строка *a* подстроку «*abc*», и если да, то выполняется ее замена на первые три символа из строки *b*. Рассмотрим данный пример подробнее.

Эффективное использование разбиения циклов, описанного в предыдущем разделе, при преобразовании программы, представленной на рис. 7, невозможно из-за того что в условии условного оператора присутствует элемент массива с индексом. Результатом вынесения его из-под цикла будет *len* услов-

ных операторов, в которых проверяется условие с конкретными значениями индекса цикла: *if* ((*a*[0] == *'a'*) && (*a*[1] == *'b'*) && (*a*[2] == *'c'*)), что в общем случае может быть бесконечной последовательностью. Возможным решением данной проблемы была бы сокращенная запись вариантов вида: $a = \{a, ba[3..len], a[0..n]ba[n+3..len], b \subset a; index = Any\}$, где запись $ba[n..m]$ означает, что конкатенируется из значений b и a от n до m; запись $b \subset a$; $index = Any$ означает, что b является подстрокой a при любом индексе $index$.

Таким образом, для записи массива a будем использовать следующую форму:

$$a_i,\ i \in [0, n],\ a_i = op,$$

где a_i – массив, n – длина массива, op – оператор над элементами массива, если такой есть в программе.

Согласно вышеприведенной форме записи можно найти значения переменных в состоянии S_q в примере программы на рис. 7.

S_0: $\{a; b\} = \{a_i, i \in [0, n]; b_i, i \in [0, n]\}$

S_1: $\{\{a_i, i \in [0, n], a_i = b[0]; a_i = 'a', a_{i+1} = 'b', a_{i+2} = 'c'\}; b_i, i \in [0, n]\}$;

S_2: $\{\{a_i, i \in [0, n], a_i = b[0], a_{i+1} = b[1]; a_i = 'a', a_{i+1} = 'b', a_{i+2} = 'c'\}; b_i, i \in [0, n]\}$;

S_3: $\{\{a_i, i \in [0, n], a_i = b[0], a_{i+1} = b[1], a_{i+2} = b[2]; a_i = 'a', a_{i+1} = 'b', a_{i+2} = 'c'\}; b_i, i \in [0, n]\}$.

5. ПРИМЕР АНАЛИЗА ПРОГРАММЫ

Рассмотрим применение вышеизложенного материала на примере сортировки массива методом «пузырька» (рис. 8).

В качестве варианта анализа будем использовать (1) в состоянии S_q, где *cond* – выражение, в котором проверяется переменная x (рис. 9).

Теперь распишем состояния программы и возможные состояния переменных. Так как переменная N объявлена как константа, то во всех состояниях она будет иметь одинаковое значение, т. е. можно ее опустить из записи.

S_0: $\{n = Any; i = Any; j = Any\}$;

S_1: $\{n = Any; i = Any; j = Any; a[1000] = \{Any, Any, \ldots, Any\}\}$.

Далее циклы в состояниях S_2, S_3 можно расписать как цикл, в котором выполняется условный оператор, и цикл, в котором он не выполняется. Так как цикл в состоянии S_3 содержит только условный оператор, то второй случай будет пустым, и его рассматривать нет смысла. Таким образом, результатом

первой итерации анализа цикла в S_3 будет $a = [a_1, a_2, a_3 \dots a_n]$, $\{a_1 > a_n, a_2 > a_n, a_3 > a_n \dots a_{n-1} > a_n\}$, результатом второй $a = [a_1, a_2, a_3 \dots a_n]$, $\{a_1 > a_{n-1}, a_2 > a_{n-1}, a_3 > a_{n-1} \dots a_{n-2} > a_{n-1}\}$, n-й $a = [a_1, a_2, a_3 \dots a_n]$, $\{a_1 > a_2\}$. Результатом S_4 будет выражение $a = \{a_1 > a_2 > a_3 > \dots > a_n\}$. Теперь для анализа условия необходимо выполнить выражение на рис. 9 с учетом известных данных.

```
#include<stdio.h>
#define N 1000

int main() {
S0:  int n, i, j;
S1:  int a[N];

S2:  for (i = 0 ; i < n ; i++) {
S3:    for (j = 0 ; j < n - i - 1 ; j++) {
S4:      if (a[j] > a[j+1]) {
S5:        int tmp = a[j];
S6:        a[j] = a[j+1];
S7:        a[j+1] = tmp;
         }
       }
     }
}
Sq:
```

Рис. 8. Пример программы сортировки массива методом «пузырька»

```
x = true;
for (i = 0; i < n - 1; i++) {
  if (a[i] > a[i+1]) {
    x = false;
    break;
  }
}
  assert(x);
```

Рис. 9. Условие проверки программы в состоянии Sq

После рассмотрения анализа примера на рис. 8 можно дополнительно к оговоренным в алгоритме 1 сформировать несколько правил анализа циклов. Данные правила будут касаться анализа циклов, в которых есть действия над элементами массива. Как видно из примера в предыдущем разделе, анализ таких элементов программы отличается от анализа не содержащих индексов переменных циклов. Рассмотрим анализ циклов с операциями над массивами подробнее.

В примере на рис. 8 состояния в условном операторе можно записать так:

$S_0 = \{\text{tmp}; a\} = \{Any, a\}$;

$S_5 = \{a[j];\ a,\ a_0[j] = a[j+1],\ a[j] > a[j+1]\}$;

$S_7 = \{a[j],\ a,\ a_0[j] = a[j+1],\ a_0[j+1] = a[j],\ a[j] > a[j+1]\}$;

$S_q = \{a\} = \{Any,\ a_0[j] = a[j+1],\ a_0[j+1] = a[j],\ a[j] > a[j+1]\}$, $j \in [0,\ n-1-i)$, $i \in [0,\ n)$;

Таким образом, в состоянии S_q переменная *a* может принимать любые значения, но кроме этого есть некоторые зависимости, по которым можно определить, выполняется проверяемое условие или нет.

```
   for (i = 0; i < n - 2; i++) {
S1:  if (b[i] == 0) {
S2:    b[i + 2] = 2;
     }
S3:}
Sq:
```

Рис. 10. Пример программы с циклом

Рассмотрим еще пример цикла на рис. 10. В данном цикле есть переменные *i*, *n*, *b*, и значения всех этих переменных неизвестно. По аналогии с предыдущими рассуждениями найдем возможные варианты массива *b* после исполнения цикла. В начале исполнения программы значение массива неизвестно: $b = Any$. После исполнения условного оператора *b* будет принимать следующие значения:

$S_1 = \{b,\ b[i] <> 0\}$;

$S_2 = \{b,\ b[i + 2] = 2,\ b[i] = 0\}$;

$S_3 = \{S_1; S_2\} = \{b,\ b[i] <> 0;\ b,\ b[i + 2] = 2,\ b[i] = 0\}$, $i \in [0,\ n\text{-}2)$.

6. ЗАДАЧА КОММИВОЯЖЕРА

Последний пример в данной работе, для которого применяется предлагаемая методика, это задача коммивояжера[2]. Входными данными данной задачи являются натуральное число *N* и квадратная матрица *D* порядка *N*, содержащая длины дорог между всеми парами городов. Если какой-либо дороги не существует, соответствующий элемент матрицы равен нулю. Выходные данные – минимальная длина пути. На рис. 11 приведена возможная ее реализация.

[2] На территории государства *X* расположено *n* городов, между некоторыми парами которых проложены дороги (не обязательно двунаправленные). Каждая из них характеризуется положительным числом – своей длиной. Из одного города в другой ведет не более одной дороги. Бродячий торговец хочет обойти все города ровно по одному разу и вернуться в исходный пункт (начинать движение можно в произвольном городе). Необходимо найти минимальное расстояние, которое ему придется пройти, или выяснить, что искомого маршрута не существует.

```
#include <iostream>

using namespace std;
const int inf = 1E9, NMAX = 16;
int n, i, j, k, m, temp, ans, d[NMAX][NMAX], t[1<<NMAX][NMAX];

bool get(int nmb, int x) {
  return (x & (1<<nmb)) != 0;
}

int main()
{
S0:  cin >> n;
S3:  for (i = 0; i < n; ++i) {
S2:    for (j = 0; j < n; ++j) {
S1:      cin >> d[i][j];
       }
     }

S4:  t[1][0] = 0;
S5:  m = 1 << n;


S13: for (i=1; i<m; i += 2) {
S12:   for (j = (i==1) ? 1 : 0; j<n; ++j) {
S6:      t[i][j] = inf;
S11:     if (j > 0 && get(j, i)) {
S7:        temp = i ^ (1 << j);
S10:       for (k = 0; k < n; ++k) {
S9:          if (get(k, i) && d[k][j] > 0) {
S8:            t[i][j] = min(t[i][j], t[temp][k] + d[k][j]);
             }
           }
         }
       }
     }


S16: for (j = 1, ans = inf; j<n; ++j) {
S14:   if (d[j][0]>0) {
S15:     ans = min(ans, t[m-1][j] + d[j][0]);
       }
     }
     if (ans == inf) {
       cout << -1;
     } else {
       cout << ans;
     }
}
```

Рис. 11. Пример программы для решения задачи коммивояжера

Для анализа данной задачи рассмотрим, каким образом меняются состояния S_i в ходе исполнения программы. S = {inf, NMAX, n, i, j, k, m, temp, ans, d[NMAX][NMAX], t[1<<NMAX][NMAX]}, но для упрощения записи и экономии места константы, входящие в выражение S, и переменные для итерирования записывать не будем: S = {temp, d[NMAX][NMAX], t[1<<NMAX][NMAX]}.

S_0 = {temp, d[16][16], t[65537][16]};

S_1 = {temp, ans, d[i][j] = *Any*, t};

S_2 = {temp, ans, d[i] = *Any*, t};

S_3 = {temp, ans, d = *Any*, t};

S_4 = {temp, ans, d = *Any*, {*Any*, t[1][0] = 0}};

S_5 = {temp, ans, d = *Any*, {*Any*, t[1][0] = 0}};

S_6 = {temp, ans, d = *Any*, t = {*Any*, t[i][j] =1E9}};

S_7 = {i ^ (1 << j), ans, d = *Any*, t = {*Any*, t[i][j] = 1E9}};

S_8 = {i ^ (1 << j), ans, d = *Any*, t = {*Any*, t[i][j] = t[temp][k] + d[k][j]}}, 1E9 > t[temp][k] + d[k][j] ||

{i ^ (1 << j), ans, d = *Any*, t = {*Any*, t[i][j] = 1E9}}, 1E9 < t[temp][k] + d[k][j];

S_9 = S8, (((i & (1 << k)) != 0) && d[k][j] > 0) = true ||+

S7, (((i & (1 << k)) != 0) && d[k][j] > 0) = false;

S_9 = {i ^ (1 << j), ans, d = *Any*, t = {*Any*, t[i][j] = t[temp][k] + d[k][j]}}, 1E9 > t[temp][k] + d[k][j], (get(k, i) && d[k][j] > 0) = true ||

{i ^ (1 << j), ans, d = *Any*, t = {*Any*, t[i][j] = 1E9}}, 1E9 < t[temp][k] + d[k][j], (get(k, i) && d[k][j] > 0) = true ||

{i ^ (1 << j), ans, d = *Any*, t = {*Any*, t[i][j] = 1E9}}, (get(k, i) && d[k][j] > 0) = false;

S_{10} = S9, k in [0, n-1];

S_{11} = S10, j > 0 && get(j, i) = true || S6, j > 0 && get(j, i) = false;

S_{12} = S11, j in [0, n - 1];

S_{13} = S12, i in [0, m - 1], i += 2.

Теперь для анализа выражений, которые помечены как *Spre*, будем использовать алгоритм, в котором проверяется выполнимость условия в assert в каждом состоянии $s \in S$. Состояние S_1 является внутренним состоянием цикла в S_2, а состояния $S_1 – S_2$ внутренними по отношению к S_3. В состоянии S_1 переменная d[i][j] считывается из цикла, а в S_2 выполняется сам цикл, j $\in$[0, n – 1]. Тогда d[i] = *Any*, так как d[i][j] = *Any*. В S_3 цикл по i $\in$[0, n – 1], тогда

каждый из d = *Any*. Рассмотрим более сложные циклы в $S_8 – S_{13}$. Рассматриваемый подход для данных состояний применяется сложнее из-за большого количества вложенных циклов и требует дополнительного исследования.

ВЫВОД

Исходя из вышеизложенного описания массивов можно сделать следующие выводы.

1. Анализ участков кода, в которых известны все значения переменных, неэффективен, так как гораздо проще и быстрее будет просто вычислить данный код. Таким образом, проводить анализ кода необходимо только для тех участков, в которых присутствуют переменные, значения которых заранее не известны (или, как чаще всего бывает в программе, смешанный набор переменных).

2. Переменные, значения которых не заданы (значение *Any*), можно оценивать с помощью операторов сравнения ($a > 5$, $b < 10$, $a > c$), операторов присваивания ($a = 10$, $b = c$) и/или других программных операторов. Результатом такой оценки является система зависимостей, в которой участвуют переменные.

3. В [11, 12] зависимости между переменными использовались для более точного определения ошибок при статическом анализе, но в них не было предложено выполнять анализ в конкретных состояниях на всем множестве значений переменных.

СПИСОК ЛИТЕРАТУРЫ

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

2. Coherent clusters in source code / S. Islam, J. Krinke, D. Binkley, M. Harman // Journal of systems and software. – 2014. – Vol. 88. – P. 1–24.

3. *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.

4. *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.

5. *Шелехов В.И.* Методы доказательства корректности программ с хорошей логикой [Электронный ресурс] // Международная конференция «Совре-

менные проблемы математики, информатики и биоинформатики», посвященная 100-летию со дня рождения члена-корреспондента АН СССР А.А. Ляпунова, 11–14 октября 2011 г.: доклады. – Новосибирск, 2011. – С. 1–21. – URL: http://conf.nsc.ru/files/conferences/Lyap-100/fulltext/74974/75473/Shelekhov_prlogic.pdf (дата обращения: 31.10.2014).

6. *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.

7. *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.

8. *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.

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

10. *Falk H., Marwedel P.* Source code optimization techniques for data flow dominated embedded software. – New York: Springer Science: Business Media, 2004. – 226 p.

11. *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.

12. *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.

Воевода Александр Александрович – доктор технических наук, профессор кафедры автоматики НГТУ. 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 НГТУ. Основное направление научных исследований – формальная верификация, проверка моделей. Имеет 31 публикацию. E-mail: rom2006@gmail.com

Methods of program representation and analysis*

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. E-mail: ucit@ucit.ru*
[2] *Novosibirsk State Technical University, 20 Karl Marks Avenue, Novosibirsk, 630073, Russian Federation, candidate of Technical Sciences, senior lecturer at the department of automation. E-mail: rom2006@gmail.com*

Different ways of program representations are described in the paper: begins from graphs and Hoare logic to Petri nets. Also new way of program representation is offered. The main idea of offered approach is in writing program in symbols (variables) and making system of dependencies between of them. Also offered approach allows to greatly reduce number of braches for analysis in program: is usual analysis part of program that contains conditional operator in loop with *n* iterations need to check *2n* different options; offered approach allows to greatly reduce number of options for checking. One of the intermediate results of considered approach is a system of dependencies between the variables with help of that we can check feasibility of some conditions in program code. The result of such checks is a confirmation that code condition is true or contrexample that shows conditions (values of variables) when check can be false. Offered approach was tested on the following examples: with using of condition operators, with using of cycles and condition operators, the traveling salesman problem. Positive results were gotten for all tasks except the last for which additional investigations are required.

Keywords: testing, input intervals, formal verification, dynamic verification, verification, model checking, software models, graphs

REFERENCES

1. 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.).

2. Islam S., Krinke J., Binkley D., Harman M. Coherent clusters in source code. *Journal of systems and software*, 2014, vol. 88, pp. 1–24.

3. 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

* *Received 1 August 2014.*

4. 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

5. Shelehov V.I. [Rules of correctness proof for programs with simple logic]. *Mezhdunarodnaya konferentsiya "Sovremennye problemy matematiki, informatiki i bioinformatiki", posvyashchennaya 100-letiyu so dnya rozhdeniya chlena-korrespondenta AN SSSR A.A. Lyapunova, 11–14 oktyabrya 2011 g.: Doklady* [International conference "Modern problems of mathematics, informatics and bioinformatics", devoted to the 100th anniversary of professor Alexei A. Lyapunov, Novosibirsk, Russia, 2011, October 11–14: Reports]. Novosibirsk, 2011, pp. 1–21. Available at: http://conf.nsc.ru/files/conferences/Lyap-100/fulltext/74974/75473/Shelekhov_prlogic.pdf (accessed 31.10.2014)

6. 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.

7. Voevoda A.A., Markov A.V. Romannikov D.O. Razrabotka programmnogo obespecheniia: proektirovanie s ispol'zova-niem 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. 218–231.

8. Romannikov D.O. Nakhozhdenie oshibok obrashcheniia k nesushchestvuiushchim 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.

9. 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.

10. Falk H., Marwedel P. Source code optimization techniques for data flow dominated embedded software, New York, Springer Science, Business Media, 2004. 226 p.

11. 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 informatsionnykh sistem – Modeling and analisys of information systems*, 2011, vol. 18, no. 4, pp. 68–79.

12. 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.