УДК 519.686.2

А.В. КОЛЧИН

АВТОМАТИЧЕСКИЙ МЕТОД ДИНАМИЧЕСКОГО ПОСТРОЕНИЯ АБСТРАКЦИЙ СОСТОЯНИЙ ФОРМАЛЬНОЙ МОДЕЛИ

Ключевые слова: *верификация, проверка модели, абстракции.*

ВВЕДЕНИЕ

Основная трудность, которую приходится преодолевать в процессе верификации формальных моделей, связана с эффектом комбинаторного взрыва в пространстве состояний. Эта проблема особенно остро возникает в системах, состоящих из многих взаимодействующих компонентов, а также в системах, обладающих высокой степенью недетерминизма и типами данных, которые способны принимать большое число значений. Решению проблемы посвящено множество различных методов: для элиминации избыточного интерливинга разработаны методы частичного порядка [1–3], для сокращения числа значений применяются методы символьного моделирования [4–8] и абстракции предикатов [6, 9–14], использование симметрий при проверке эквивалентности состояний [15–17] эффективно при проверке многокомпонентных систем. Существуют методы абстракции на основе исследования зависимостей [10, 18, 19, 20–25], сжатия состояний [26, 27] и др.

1. КРАТКОЕ ОПИСАНИЕ МЕТОДА

Данная работа описывает метод построения абстракций «на лету» и его использование для выявления таких ошибок: недетерминированное поведение, тупик, ливлок, недостижимость переходов, нарушение условий, заданных пользователем. Интуитивно суть метода заключается в игнорировании некоторых незначимых значений атрибутов. Под «незначимым» на некотором состоянии $\mathtt{q}$ понимается *значение* атрибута, которое не используется ни одним переходом, а также не различается ни одним из проверяемых свойств на всех состояниях, достижимых из $\mathtt{q}$. Соответственно «значимым» на состоянии является значение атрибута, которое используется для определения допустимости перехода, либо для проверки свойств, при этом стоит вопрос об определении факта значимости. Например, непосредственно из таблицы истинности формулы $X \wedge Y \vee Z$ следует, что если значение атомарной формулы X ложно, то значение Y не влияет на результат, т.е. значение Y не является значимым; аналогично значение Z можно не вычислять, если X и Y истинны. Во многих языках программирования, таких как Си [28] и др., бинарные логические операции, вообще говоря, интерпретируются как некоммутативные и, как правило, ассоциируются слева направо. Сначала всегда вычисляется первый операнд; если его значения достаточно для определения результата операции, то второй операнд не вычисляется.

Данный метод определяет значимость динамически путем исследования компонент сильной связности графа поведения модели, используя при этом технику «ленивых» вычислений. Для каждого значимого значения атрибута проводится анализ его истории изменения, использования и зависимости. Так, для каждого пройденного состояния динамически выделяется достаточное для анализа свойств модели подмножество атрибутов, которое используется для проверки эквивалентности состояний. Отношение эквивалентности ослабляется: не различается варьирование значений незначимых атрибутов, при этом достигается существенное сокращение количества состояний и, как следствие, памяти и времени, необходимых для анализа верифицируемой модели.

Описаны основные алгоритмы построения абстракций, в завершение представлены примеры, иллюстрирующие эффективность применения метода, и сравнительный анализ с существующими методами и некоторыми популярными верификаторами.

2. ФОРМАЛЬНЫЕ МОДЕЛИ И ИХ СВОЙСТВА

Пусть задано конечное множество атрибутов $A = v_1, v_2, \ldots, v_n$, и пусть также для каждого атрибута $v_i \in A$ определена конечная область допустимых значений $D(v_i)$.

Определение 1. Состоянием называется множество пар атрибутов и их значений вида $q = \left\{ \bigcup_i (v_i = d_i) \mid v_i \in A, d_i \in D(v_i) \right\}$.

Определение 2. Предусловием называется бесквантерная формула (классической) логики предикатов первого порядка над атрибутами множества A и констант множества $D = \bigcup_i D(v_i)$.

Определение 3. Постусловием $\beta(q, q')$ называется конечное множество присваиваний вида $v := F(q, e)$, где $v \in A$ — атрибут состояния q', e — выражение над константами D и атрибутами состояния q, F — атрибутная функция, вычисляющая значение выражения e в состоянии q.

Определение 4. Размеченным переходом называется тройка вида (α, t, β), где α — предусловие, t — метка (имя перехода), β — постусловие.

Определение 5. Атрибутной транзиционной системой (АТС) называется восьмерка вида $M = (Q, q^0, R, L, A, D, I, F)$, где Q — конечное множество состояний, $q^0 \in Q$ — начальное состояние, R — конечное множество размеченных переходов, L — конечное множество меток, A — конечное множество атрибутов, $D(v)$ — конечная область значений для каждого атрибута $v \in A$. Интерпретация атомарных формул предусловий задана функцией $I : Q \times \Pi \to \{\mathbf{T}, \mathbf{F}\}$, где Π — множество всех атомарных формул из предусловий R, $\mathbf{T}$ и $\mathbf{F}$ — соответственно «истина» и «ложь». Интерпретация присваиваний в постусловиях задана функцией $F : Q \times E \to D$, где E — множество всех выражений из постусловий R.

Аналогично охраняемым командам Дейкстры [29] с помощью отношения R задаются возможные варианты дискретных детерминированных переходов из одного множества состояний в другое. В описании алгоритмов для I и F будет использоваться функция `interpret(<состояние>,<выражение>)`. АТС является удобным математическим аппаратом для описания поведения широкого спектра формальных моделей систем асинхронно-взаимодействующих процессов и их абстракций.

Определение 6. Истинность формулы φ на состоянии `q`, записывается $\mathtt{q} \models \varphi$, определяется индуктивно по структуре формулы φ:

`q |= a` $\equiv$ I(`q,a`)= $\mathbf{T}$, *если атомарная формула a истинна в состоянии q*;

$\mathtt{q} \models \neg\varphi \equiv \mathtt{q} \not\models \varphi$, *если формула* φ *не выполняется в q*;

$\mathtt{q} \models \varphi_1 \wedge \varphi_2 \equiv \mathtt{q} \models \varphi_1 \wedge \mathtt{q} \models \varphi_2$, *если в q выполняются формулы* φ_1 *и* φ_2;

$\mathtt{q} \models \varphi_1 \wedge \varphi_2 \equiv \mathtt{q} \models \varphi_1 \vee \mathtt{q} \models \varphi_2$, *если в q выполняется формула* φ_1 *или* φ_2.

В дальнейшем используется запись $q \xrightarrow{t} q'$, если предусловие перехода t выполняется в состоянии q, т.е. $\alpha_t \models q$ и выполнение присваиваний постусловия β_t преобразует состояние q в q'; такой переход будет называться допустимым из q.

Определение 7. Путем в M из состояния q^i в состояние q^j называется такая последовательность состояний и переходов $q^i \xrightarrow{t_i} q^{i+1} \xrightarrow{t_{i+1}} q^{i+2} \ldots q^j$, что

$q^k \in Q \wedge t_k \in L$ для всех $k \in \{i, \dots, j\}$. Путь называется простым, если в нем нет повторяющихся состояний.

Лемма 1. Длина простого пути в АТС конечна.

Определение 8. Состояние q^i достижимо из состояния q^j, если существует путь из q^i в q^j. Состояние q достижимо из себя (в этом случае множество переходов пути из q в q будет пустым).

Лемма 2. Если состояние q^i достижимо из q^j, то существует простой путь из q^i в q^j.

Определение 9. Трассой в M называется упорядоченная последовательность $t_0, t_1, \dots, t_n, \dots$ такая, что существует путь $q^0 \xrightarrow{t_0} q^1 \xrightarrow{t_1} \dots \xrightarrow{t_n} q^n \dots$.

Определение 10. Язык, ассоциированный с АТС M, обозначается $\mathcal{L}(M) \subset L^*$ — это набор всех трасс, выходящих из начального состояния q^0.

Определение 11. *Конкретной* АТС называется $M_c = (Q_c, q_c^0, R, L, A, D, I, F)$, в которой каждое состояние $q_c \in Q_c$ конкретное. Конкретное состояние включает в себя значения всех атрибутов: $q_c = \left\{ \bigcup_{0 \le i \le |A|} (v_i = d_i) | v_i \in A \wedge d_i \in D(v_i) \right\}$.

Определение 12. В *абстрактной* АТС $M_a = (Q_a, q_a^0, R, L, A, D, I, F)$ каждое состояние $q_a \in Q_a$ абстрактное. Для каждого абстрактного состояния существует такое подмножество $A_{q_a} \subset A$, что $q_a = \left\{ \bigcup_i (v_i = d_i) | v_i \in A_{q_a} \wedge d_i \in D(v_i) \right\}$.

Далее для конкретного состояния будет использоваться обозначение q_c, а для абстрактного — q_a.

3. ПРОВЕРЯЕМЫЕ СВОЙСТВА ФОРМАЛЬНЫХ МОДЕЛЕЙ

В основном системы верификации осуществляют проверку таких свойств: отсутствие выходов за пределы допустимых значений (правильность индексации, размеров буферов и т.п.), переполнений и потерь значимости (over/underflow), деления на ноль, обращений к неинициализированным атрибутам, а также проверку достижимости, детерминированности переходов, отсутствие тупиков (deadlock, livelock). Иногда проверяются свойства безопасности и живости, сформулированные пользователем. Всякий раз, когда достигнуто состояние, проверяемое на достижимость, либо нарушающее одно из описанных выше свойств, строится трасса (или путь), иллюстрирующая поведение модели из ее начального состояния в текущее.

Определение 13. Условием безопасности называется бескванторная формула (классической) логики предикатов первого порядка над атрибутами множества A и констант множества $D = \bigcup_i D(v_i)$.

Нарушение условий безопасности (сгенерированных или заданных пользователем) требует введения терминальных состояний и переопределения достижимости состояния.

Определение 14. Пусть заданы свойства Ψ. Тогда состояние q^k достижимо в M, если существует путь $q^0 \xrightarrow{t_0} q^1 \xrightarrow{t_1} \dots q^k$, причем $(\varphi \in \Psi) \Rightarrow (q^i \models \varphi)$ для всех $i \in \{0, \dots, (k-1)\}$.

Далее вместо определения достижимости 8 будет использоваться определение 14. Таким образом, всякое состояние, нарушающее какое-либо из заданных для проверки свойств, является терминальным. Множество всех состояний, достижимых из q, обозначается $\delta(q)$.

Проверка достижимости выполняется двумя различными способами. Первый — проверка достижимости *состояния*, при этом в общем случае речь идет о поведении, удовлетворяющем заданной формуле темпоральной логики. Второй способ — проверка достижимости *переходов*. При окончании работы генерируется вердикт, включающий в себя информацию о статистике выполнения переходов.

Определение 15. Переход t достижим, если существует достижимое состояние $q \in \delta(q^0)$ такое, что $q \models \alpha_t \wedge (\varphi \in \Psi \Rightarrow q \models \varphi)$.

Определение 16. Недетермированным называется состояние q^n, для которого существует больше одного допустимого перехода: $q^n \models \alpha_{t_1} \wedge q^n \models \alpha_{t_2} | t_1 \neq t_2 \wedge t_1, t_2 \in R$.

Недетерминизм (transition inconsistency) в модели является не ошибкой ее поведения, а скорее предупреждающим сигналом для дальнейшего анализа. Метод не разделяет естественный недетерминизм (например, моделируемого поведения внешней среды) от ошибочного (неоднозначного поведения детерминированных модулей).

Определение 17. Тупиковым (deadlock) называется состояние q^d, из которого не существует ни одного допустимого перехода: $t \in R \Rightarrow q^d \not\models \alpha_t$.

Ливлоком (livelock) называется ситуация, в которой хотя бы одно состояние, достижимое из начального состояния, более недостижимо. Другими словами, условие отсутствия ливлоков можно сформулировать так: каждое достижимое состояние должно быть достижимо из любого изначально достижимого состояния.

Определение 18. q^l есть состояние ливлока, если $q^l \in \delta(q^0) \wedge \delta(q^0) - \delta(q^l) \neq \varnothing$.

Очевидно, частным случаем ливлока является тупиковое состояние, однако состояние ливлока допускает также (возможно, бесконечное) продолжение пути. Необходимо отметить, что если тупиковая ситуация почти всегда интерпретируется как ошибка, то ливлок в модели требует более детального анализа со стороны пользователя. Например, проверка ливлока не актуальна для систем с ациклическим поведением, в которых определены специальные терминальные состояния.

4. ТОЧНОСТЬ АБСТРАКЦИИ

Недостатком бисимуляцинного отношения, определенного в работе Парка [30], являются его строгие требования, которым часто не удовлетворяют модели на разных уровнях абстракции. В частности, условие совпадения разметки состояний сохраняет консервативность относительно любых формул логики CTL*, однако может стать избыточным условием при проверке только интересующих свойств. Назовем ослабленным отношением бисимуляции такое отношение, которое не учитывает совпадение разметки состояний.

Определение 19 (ослабленная бисимуляция). Пусть заданы две АТС $M_i = (Q_i, q_i^0, R_i, L_i, A_i, D_i, I_i, F_i)$, $i = 1, 2$. Отношение $B \subset Q_1 \times Q_2$ называется отношением бисимуляции, а модели M_1 и M_2 называются бисимуляционно эквивалентными тогда и только тогда, когда для любой пары состояний моделей $(s, q) \in B$ выполняются следующие условия:

1) для любого состояния $s' \in Q_1$ такого, что $s \xrightarrow{t} s'$, найдется такое состояние $q' \in Q_2$, что $q \xrightarrow{t} q'$, и при этом выполняется $(s', q') \in B$;

2) для любого состояния $q' \in Q_2$ такого, что $q \xrightarrow{t} q'$, найдется такое состояние $s' \in Q_1$, что $s \xrightarrow{t} s'$, и при этом выполняется $(s', q') \in B$;

3) два состояния s и q называются бисимуляционно эквивалентными, если и только если существует бисимуляция B такая, что $(s, q) \in B$;

4) $(q_1^0, q_2^0) \in B$.

Для систем детерминированных переходов такое отношение бисимуляции совпадает с трассовой эквивалентностью. Таким образом, для определения достижимости переходов, проверки условий живости и безопасности, наличия ошибочных состояний, тупиков и ливлоков достаточно отношения эквивалентности, сформулированного в следующем определении.

Определение 20. Абстрактная АТС M_a есть *точная абстракция* конкретной АТС M_c по отношению к проверяемым свойствам Ψ, если одинаково выполняются все свойства $(\varphi \in \Psi) \Rightarrow (M_a \models \varphi \Leftrightarrow M_c \models \varphi)$ и множества трасс совпадают $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.

5. ОСНОВНЫЕ ПРЕДПОЛОЖЕНИЯ

Для описания алгоритмов будет использоваться язык высокого уровня, состоящий из Упрощенного Алгола [31] и правил переписывания термов языка алгебраического программирования APLAN [32]. Из соображений простоты описания и доказательств приведенные алгоритмы не претендуют на оптимальность. Не теряя общности, предполагается, что ни предусловия, ни постусловия не допускают деления на ноль, выходов за пределы допустимых значений, переполнений и потерь значимости, а также использования неинициализированных атрибутов. Анализ таких свойств потребует введения соответствующих вспомогательных атрибутов и проверок; требуемая модификация не является критичной для доказательства основных свойств алгоритмов.

6. ОПИСАНИЕ ОСНОВНЫХ АЛГОРИТМОВ

Далее описан алгоритм, преобразующий логическую формулу в программу, интерпретирующую предусловия переходов с учетом некоммутативности логических операторов.

Алгоритм 1. Интерпретация предусловий.

Вход. Имя перехода `t` и формула предусловия `Pre`.

Выход. Процедура `precond[t]`, интерпретирующая предусловие `Pre`.

```
interpret_pre := procedure(t, Pre) begin
   return ` precond[t] := procedure(S)local(R_SET, result) begin
                R_SET ← ∅;' ! truth_table(Pre, T) !
                `return (result; R_SET) end'
end
```

Кавычки `` ` ' `` используются для обработки строк, знак «!» обозначает операцию конкатенации. Оператор `truth_ table` представляет собой систему переписывающих правил, которая сопоставляет входной терм левой части (до знака «=») каждого правила сверху вниз до первого подходящего, и на выходе строит терм согласно правой части правила; **T**, **F** обозначают `True`, `False`; функция `interpret` вычисляет значение атомарных формул:

```
truth_table := rewrite_system(a, b)begin
1. (a \/ b, T)  = `begin ' ! truth_table(a,T) ! `end
          if(result=F)then do begin ' ! truth_table(b,T) ! `end',
2. (a \/ b, F)  = `begin ' ! truth_table(a,F) ! `end
          if(result=T)then do begin ' ! truth_table(b,F) ! `end',
3. (a & b, T)   = `begin ' ! truth_table(a,T) ! `end
          if(result=T)then do begin ' ! truth_table(b,T) ! `end',
4. (a & b, F)   = `begin ' ! truth_table(a,F) ! `end
          if(result=F)then do begin ' ! truth_table(b,F) ! `end',
5. (~(a), T)    = truth_table(a, F),
6. (~(a), F)    = truth_table(a, T),
7.      (a, T)  = read(a) ! `result ← interpret (S, ' ! a ! ')',
8.      (a, F)  = read(a) ! `result ← ~( interpret (S, ' ! a ! '))',
end
```

Процедура read строит операторы для добавления атрибутов, входящих в формулу, в некоторое множество R_SET:

```
read := procedure(atomic_formula)local(r)begin
  r ← '';
  for attr ∈ atomic_formula do r ← r ! 'R_SET ← R_SET ∪' ! attr ! ';';
  return r
end
```

□

Алгоритм 1 применяется также для интерпретации свойств модели, заданных пользователем формулами логики предикатов первого порядка. При этом вместо имени перехода первым параметром указывается идентификатор проверяемого свойства. На рис. 1 показаны примеры построения процедуры для перехода t1 и проверки свойства safety1.

Вход:
```
interpret_pre(t1, (a>0 \/ b=1))
```
Выход:
```
precond[t1] := procedure(S)
            local (R_SET, result)
begin R_SET ← ∅;
   R_SET ← R_SET ∪ a;
   result ← interpret(S, a>0);
   if (result = F) then do begin
      R_SET ← R_SET ∪ b;
      result ← interpret(S,b=1)
   end
   return (result; R_SET)
end
```

Вход:
```
interpret_pre(safety1,~(a>0 & b=1))
```
Выход:
```
precond[safety1] := procedure(S)
              local (R_SET, result)
begin R_SET ← ∅;
   R_SET ← R_SET ∪ a;
   result ← ~(interpret(S, a>0));
   if (result = F) then do begin
      R_SET ← R_SET ∪ b;
      result ← ~(interpret(S,b=1))
   end
   return (result; R_SET)
end
```

Рис. 1

Лемма 3. Алгоритм 1 строит процедуру precond, которая:

а) интерпретирует логическую формулу согласно таблицам истинности для некоммутативных логических выражений, содержащих только пропозициональные связки и имеет на выходе **T**, если формула истинна, и **F** в противном случае;

б) для бинарных операций не вычисляет второй операнд, если значения первого операнда достаточно для определения результата операции;

в) множество R_SET на выходе содержит только атрибуты, входящие в операнды, значения которых вычислялись.

Доказательство:

а) преобразование пропозициональных связок (truth_table, правила 1–6) и вычисление результата интерпретации атомарных формул, т.е. вызов функции interpret (правила 7–8) производятся с учетом полярности (**T**, **F**) согласно таблицам истинности для некоммутативных логических выражений;

б) согласно семантике операторов if–then вычисление второго операнда (truth_table , правила 1–4) осуществляется только в случае, когда значения первого операнда недостаточно для определения результата операции;

в) базис рекурсии (последние два правила) содержит вызов процедуры interpret, которая вычисляет результат атомарного выражения, а так как процедура read вызывается только перед вызовом interpret с тем же параметром, и только она формирует множество R_SET, то множество будет содержать только атрибуты, входящие в операнды, значения которых вычислялись. □

Следствие 1. Значений атрибутов множества R_SET на выходе процедуры precond достаточно для определения допустимости перехода.

Процедура property_check формирует множество атрибутов R_SET, которые используются для проверки требуемых свойств (выраженных бесквантворными формулами логики предикатов) модели на состоянии S:

```
property_check := procedure(S,Ψ)local(RESULT, R_SET, res, r)begin
   RESULT ← T; R_SET ← ∅;
   for p ∈ Ψ do begin
      (res; r) ← precond[p](S);
      if (res = F) then do RESULT ← F;
      R_SET ← R_SET ∪ r
   end
   return (RESULT; R_SET)
end
```

Следствие 2. Значений атрибутов множества R_SET на выходе процедуры property_check достаточно для определения выполнимости свойств множества Ψ.

Необходимо отметить, что R_SET будет содержать не обязательно все атрибуты, входящие в предусловие перехода или формулу условия безопасности модели. Для краткости атрибуты множеств R_SET будут называться R-атрибутами.

Далее рассмотрен алгоритм 2, который будет использоваться для построения процедур для интерпретации постусловий. Из соображений простоты описания предполагается, что присваивания в постусловии коммутативны, т.е. не содержат повторных присваиваний и не используют новых значений атрибутов.

Алгоритм 2. Интерпретация постусловий.

Вход. Переход t, множество присваиваний постусловия Post, а также исходное состояние Src и целевое Dst.

Выход. Процедура postcond[t] для интерпретации присваиваний Post, а также множества W_SET и V_SET[].

```
interpret_post := procedure(t, Post)local(r, w, expr, v)
  begin
1.  r ← 'postcond[t] := procedure(Src, Dst)local(W_SET,V_SET[ ])
          begin W_SET ← ∅; V_SET[ ] ← ∅;' ;
2.  for (w := expr) ∈ Post do begin
3.     r ← r ! 'W_SET ← W_SET ∪ ' ! w ! ';' ;
4.     for v ∈ expr do
5.        r ← r ! 'V_SET[' ! w ! ']← V_SET[' ! w ! ']∪ ' ! v ! ';' ;
6.     r ← r ! 'Dst->' ! w ! '← interpret(Src, expr)' ! ';'
    end
7.  r ← r ! 'return (Dst; W_SET; V_SET) end' ;
8.  return r
  end
```

Далее приведен пример результата работы алгоритма 2 (рис. 2).

```
Вход:
interpret_post(t1, (a := a - 1; c := b))
Выход:
postcond[t1] := procedure(Src, Dst)local(W_SET, V_SET[])
begin
   W_SET ← ∅; V_SET[]←∅;
   W_SET ← W_SET ∪ a;   V_SET[a] ← V_SET[a] ∪ a;
   Dst->a ← interpret(Src, a - 1);
   W_SET ← W_SET ∪ c;   V_SET[c] ← V_SET[c] ∪ b;
   Dst->c ← interpret(Src, b);
   return (Dst, W_SET, V_SET)
end
```

Рис. 2

Лемма 4. Алгоритм 2 строит процедуру postcond, которая:

а) выполнит все присваивания постусловия из состояния β(Src,Dst);

б) сформирует множество атрибутов W_SET, которым осуществлялось присваивание;

в) для каждого атрибута w из множества W_SET формирует множество V_SET[w] атрибутов, которые входят в выражение, формирующее значение перезаписываемого атрибута w.

Доказательство:

а) строка 2 предполагает выполнение строки 6 для каждого присваивания; в строке 6 присваивание значения, вычисляемого из предыдущего состояния Src, выполняется для атрибута нового состояния Dst;

б) строка 2 предполагает выполнение строки 3 для каждого присваивания; строка 3 выполняет формирование множества W_SET, добавляя перезаписываемый атрибут;

в) строка 2 выполнит строку 4 для каждого присваивания; строка 4 предполагает выполнение добавления (строка 5) каждого атрибута, входящего в выражение, формирующее значение перезаписываемого атрибута w во множество V_SET[w].

Для краткости атрибуты множества W_SET называются W-атрибутами, а V_SET[] — V-атрибутами. Множества R- и V-атрибутов различаются намеренно, потому что значения W-атрибутов могут не использоваться, и таким образом, соответствующие значения V-атрибутов также останутся неиспользованными, а значит, «незначимыми». Однако если W-атрибут (на продолжении пути) будет участвовать в определении поведения или в проверке свойств Ψ, т.е. станет R-атрибутом, то элементы, формировавшие значение такого W-атрибута (V-атрибуты), также станут «значимыми». Далее определена функция перехода transit, которая на основании имени перехода Transition и состояния Src формирует множество R-атрибутов, и, если переход допустим, — новое состояние Dst, а также множества V- и W-атрибутов:

```
transit := procedure(Src, Transition, Dst) local(RESULT, R_SET, W_SET, V_SET)
begin
   (RESULT; R_SET) ← precond[Transition](Src);
   if(RESULT = T) then do(Dst; W_SET; V_SET) ← postcond[Transition](Src, Dst);
   return (RESULT; Dst; R_SET; W_SET; V_SET)
end                                                                    □
```

На рис. 3 показан пример работы процедуры transit.

Предусловие перехода t1 : (a>0 \/ b=1).
Постусловие перехода t1 : (a := a - 1; c := b).
Вход:
transit((a = 2, b = 0), t1, (a = 2, b = 0)).
Выход:
(**T**;(a = 1, b = 0, c = 0);(a,c);([a]a,[c]b)).

Рис. 3

Согласно леммам 3 и 4 процедура transit удовлетворяет определению перехода АТС.

В дальнейшем для формирования состояний пути применяется структура path:

```
path = (link, deep, state, abstract, pred, idem,
        V_SET[], W_SET, explored, deadlock).
```

Атрибуты link и deep используются для нахождения сильно связных компонент графа поведения модели. Множество state необходимо для хранения значения всех атрибутов модели, а abstract — для формирования абстрактных состояний. Поле pred служит для хранения указателя на предыдущее состояние пути, а множество idem — для указателей на потенциально эквивалентные данному абстрактные состояния пути. Множества R_SET, V_SET[] и W_SET формируются из соответствующих множеств при осуществлении перехода процедурой transit. Множество explored служит для запоминания исследованных переходов, а флаг deadlock — признаком тупиковой ситуации.

Определение 21. Пусть задана структура пути p, и пусть s — указатель на состояние этого пути. Тогда prev(p,s) — множество всех состояний пути, предшествующих s:

$$\texttt{prev(p,s)} = \texttt{s->pred} \cup \begin{Bmatrix} \texttt{prev(p,s->pred)}, & \text{если } \texttt{s->pred} \neq \varnothing \\ \varnothing, & \text{если } \texttt{s->pred} = \varnothing \end{Bmatrix}.$$

Очевидно, что для вычисления значения атрибута на предыдущем состоянии пути в случае, когда постусловие последнего перехода содержит присваивание этому атрибуту, достаточно знать значения всех атрибутов, находящихся в правой части такого присваивания. Разумеется, если постусловие последнего на пути перехода не содержит присваивания атрибуту, то его значение не изменится.

Определение 22. Пусть s — указатель на состояние некоторого пути. Тогда историей значения атрибута a в состоянии s является множество H_a^s, состоящее из самого атрибута a, или если его значение перезаписывалось в постусловии предыдущего перехода, то все атрибуты, содержащиеся в правой части соответствующего присваивания, запишем

$$H_a^s = \begin{Bmatrix} \texttt{a}, & \text{если } \texttt{a} \notin \texttt{s->W\_SET} \\ \texttt{s->V\_SET[a]}, & \text{если } \texttt{a} \in \texttt{s->W\_SET} \end{Bmatrix}.$$

Очевидно, что значение атрибута a в состоянии s однозначно определяют значения атрибутов множества H_a^s в состоянии s->pred. Легко видеть, что процедура H правильно вычисляет соответствующую функцию H_a^s:

```
H := procedure(s, a) begin
   if (s = ∅) then do return ∅;
   if (a ∈ s->W_SET) then do return s->V_SET[a];
   else do return a
end                                    □
```

Для построения абстрактных состояний используется алгоритм 3 формирования историй значений атрибутов. На основании множеств R_SET, W_SET и V_SET[] каждого состояния некоторого пути алгоритм рекурсивно формирует множества атрибутов в каждом состоянии пути, значений которых достаточно для однозначного определения значения атрибута a в состоянии s.

Алгоритм 3. Формирование истории значений атрибутов.

Вход. Состояние s, атрибут a.

Выход. Дополненные историей значения атрибута a в состоянии s множества abstract заданного пути.

```
form_abstract := procedure(s, a)local(v)
begin
1.  if (s = ∅) then do return;
2.  s->abstract ← s->abstract ∪ a;
3.  for v ∈ H(s, a) do form_abstract(s->pred, v);
4.  return
end                                    □
```

Пример работы алгоритма 3 приведен на рис. 4.

Лемма 5 (правильность формирования истории значений атрибутов). Пусть задана структура пути path, и пусть s — указатель на состояние этого пути, a — некоторый атрибут. Тогда после завершения работы процедуры form_abstract(s,a) значений атрибутов множества q->abstract достаточно для однозначного определения значения атрибута a в состоянии s для всех $\texttt{q} \in \{\texttt{s} \cup \texttt{prev(s)}\}$.

Доказательство. Строка 1 процедуры form_abstract обрабатывает базис рекурсии, осуществляя возврат в случае, когда весь путь пройден. Строка 3 обеспечивает рекурсивный вызов для каждого элемента, формирующего согласно опреде-

лению 22 историю атрибута a, строка 2 добавляет во множество abstract текущего состояния сам атрибут a. Очевидно, что последовательное выполнение постусловий переходов пути q−>…−>s, где q∈prev(s), приведет к тому, что в состоянии s атрибут a имеет однозначно определенное значение. □

```
Путь: path = (
path[0]: state = {cf = 1, b = 0, z = 1};
         abstract = cf; W_SET=∅; V_SET=∅.
t[0]:    pre:  (cf = 1);
         post: (cf := 2; b := z).
path[1]: state = {cf = 2, b = 1, z = 1};
         abstract = cf; W_SET=b; V_SET:[b]=z.
t[1]:    pre:  (cf = 2);
         post: (cf := cf + 1).
path[2]: state = {cf = 3, b = 1, z = 1};
         abstract = ∅; W_SET=cf; V_SET:[cf]=cf    ).
Вход:
s = path[2]; a = b
Выход: (в abstract добавлены атрибуты z и b)
path = (
path[0]: abstract = cf,z; W_SET=∅; V_SET=∅
path[1]: abstract = cf,z; W_SET=b; V_SET:[b]=z
path[2]: abstract = b; W_SET=cf; V_SET:[cf]=cf )
```

Рис. 4

Далее описан основной алгоритм проверки свойств модели, базирующийся на алгоритме Тарьяна [33]. Алгоритм 4 реализует обход пространства поведения модели, а также находит компоненты сильной связности. Последнее свойство используется и для обнаружения ливлоков.

Алгоритм 4. Проверка свойств модели.

Вход. АТС (в которой множество состояний еще не построено) представленная функциями переходов transit, множеством переходов Transitions и начальным состоянием q^0, функция property_check и множество проверяемых свойств Ψ.

Выход. Трассы и множество абстрактных состояний visited.

```
model_check := procedure(transit, q^0, property_check, Ψ) local(s)
begin
   visited ← ∅; stack ← ∅;
   s ← new_state(∅); s->deep ← 0; s->state = q^0;
   traverse(s)
end
```

Множество visited будет содержать достижимые абстрактные состояния Q_a, stack — состояния компонент сильной связности. Для инициализации состояний используется процедура new_state:

```
new_state := procedure(s)local(s_new)
begin
   if (s ≠ ∅) then do begin
      s_new->state ← s->state;
      s_new->deep ← s->deep + 1
   end
   s_new->pred = s;
   s_new->abstract ← ∅; s_new->V_SET ← ∅; s_new->W_SET ← ∅;
   s_new->explored ← ∅; s_new->deadlock ← T; s_new->idem ←∅;
   return s_new
end
```

Основной процедурой алгоритма является процедура traverse:

```
traverse:=procedure(s)local(R_T, R_V, R_P, W, V, t, res, a, c, j, x)
begin
 1.   затолкнуть s в stack;
     // проверка свойств
 2.   (res, R_P) ← property_check(s, Ψ);
 3.   for a ∈ R_P do form_abstract(s, a);
 4.   if (res = T) then do put_trace(s, "property")
 5.   else do
     // выполнение переходов
 6.   for t ∈ Transitions \ s->explored do begin
 7.      s->explored ← s->explored ∪ t;
 8.      s_new ← new_state(s);
 9.      s_new->link ← s_new->deep;
10.      (res, s_new->state, R_T, W, V) ←
                          ← transit(s->state, t, s_new->state);
11.      for a ∈ R_T do form_abstract(s, a);
12.      if (res = T) then do begin
13.         if (s->deadlock = F) then do put_trace(s, "nondet")
14.         else do s->deadlock ← F;
     // проверка цикла
15.         (res, c) ← check_cycle(s_new);
16.         if (res = T) then do begin
17.            for a ∈ c->abstract do form_abstract(s_new, a);
18.            s->link ← min(s->link, c->deep);
19.            c->idem ← c->idem ∪ s_new;
20.            return
               end
     // проверка пройденного состояния
21.         (res, R_V) ← check_visited(s_new);
22.         if (res = T) then do begin
23.            for a ∈ R_V do form_abstract(s_new, a);
24.            return
               end
     // заполнение структуры path и шаг рекурсии
25.         s_new->W_SET ← W; s_new->V_SET ← V;
26.         traverse(s_new);
27.         s->link ← min(s->link, s_new->link)
         end
      end
28.   for x ∈ s->idem do
29.      for a ∈ s->abstract do form_abstract(x, a);
30.      if (equ(x, s, s->abstract) = F) then do begin
31.         s->idem ← s->idem \ x;
32.         traverse(x);
33.         s->link ← min(s->link, x->link)
         end

       // обработка сильносвязной компоненты
34.   if (s->link = s->deep) then do begin
35.      if (s->deadlock = T) then do put_trace(s, "deadlock")
36.      else do put_trace(s, "livelock");
37.      repeat
38.         вытолкнуть x из вершины stack;
39.         store_visited(x, x->abstract);
40.      until x = s
      end
41.   return
   end                                                    □
```

Процедура equ сравнивает состояния s_1 и s_2 с требуемой точностью: состояния считаются (потенциально) эквивалентными, если равны значения атрибутов из множества attr_set.

```
equ := procedure(s1, s2, attr_set)local(a) begin
   if (forall a ∈ attr_set) interpret(s1, a) = interpret(s2, a)
   then do return T else do return F
end
```

Процедура store_visited используется для сохранения пройденных состояний. Согласно строке 39 множество visited формируются только из абстрактных состояний.

```
store_visited := procedure(S, Aa)local(qa, attr) begin
   qa ← ∅;
   for attr ∈ Aa do qa ← qa ∪ (attr = interpret(S, attr));
   visited ← visited ∪ qa
end
```

Процедура check_visited проверяет принадлежность абстракции состояния s множеству пройденных ранее состояний visited и в случае обнаружения возвращает **T** и множество атрибутов ранее пройденного абстрактного состояния, иначе **F**.

```
check_visited := procedure(s)local(qa, attr) begin
   for qa ∈ visited do
    if (forall(attr = value) ∈ qa) value = interpret(s,attr)
       then do return (T, {attributes ∈ qa});
   return (F; ∅)
end
```

Процедура check_cycle проверяет эквивалентность текущего состояния с состояниями в стеке для обнаружения циклов и пройденных состояний внутри компоненты сильной связности. Возвращает **T** и указатель на соответствующее абстрактное состояние или **F**.

```
check_cycle := procedure(s)local(q, attr) begin
1.   for q ∈ stack do begin
2.      if equ(q, s, q->abstract) then do
3.         return (T, q);
4.   return (F, ∅)
  end
```

Таким образом, согласно строке 4 процедуры check_cycle(s) результатом ее работы будет пара $(\mathbf{F}, \varnothing)$ в случае отсутствия цикла в состоянии s или согласно строкам 2, 3 — пара $(\mathbf{T}, q)$, если существует такой абстрактный цикл, в котором значения атрибутов множества q->abstract совпадают.

Лемма 6. Если процедура check_cycle обнаружила цикл в графе поведения абстрактной АТС $C_a = q_a^0, \ldots, q_a^k, \ldots, q_a^{k+z}$ $(q_a^k = q_a^{k+z})$, то существует соответствующий ему цикл в графе поведения конкретной АТС $C_c = q_c^0, \ldots, q_c^k, \ldots, q_c^{k+z}, \ldots$ $\ldots, q_c^{k+n\cdot z}$ $(q_c^k = q_c^{k+n\cdot z},\ n \geq 1)$, что $q_a^{k+i} \subset q_c^{k+n\cdot z+i}$ для всех $i = \{0, \ldots, z\}$.

Доказательство. Так как $q_a^k = q_a^{k+z}$, то этот же участок пути (длины z) допустим и из состояния q_a^{k+z}, а значит, такой путь можно продлевать сколько угодно, добавляя состояния и переходы из участка $q_a^k \ldots q_a^{k+z}$. В силу леммы 1 длина соответствующего простого пути в конкретной АТС конечна. А поскольку согласно

строкам 3, 11, 17, 23, 29 и лемме 5 множества q_a^k –>abstract в состоянии q_a^k достаточно для определения значений всех атрибутов, определяющих допустимость всех переходов участка пути $q_a^k \dots q_a^{k+z}$, то существует такое конкретное состояние $q_c^{k+n\cdot z}$ $(n \geq 1)$, что $q_c^{k+n\cdot z} = q_c^k$, причем так как каждое абстрактное состояние согласно определению 12, строке 39, а также процедуре store_visited является подмножеством конкретного, то $q_a^{k+i} \subset q_c^{k+n\cdot z+i}$ для всех $i = \{0, \dots, z\}$. □

Процедура put_trace запишет трассу на диск, если не исчерпано соответствующее ограничение на число записываемых трасс, определяемое пользователем. Трасса может быть построена из событий переходов. Такие события описывают наблюдаемое поведение модели и могут использоваться в качестве сценарных контрпримеров для тестирования.

Вход:
конкретная АТС:
Атрибуты: s, p.
Значения: s={1,2,3,4}, p={0,1,…,255}.
Начальное состояние: q^0:(s=1, p=0).
Переходы модели:
t_1: (s=1)->(s:=2; p:=p+1),
t_2: (s=2)->(s:=3;p:=p+1),
t_3: (s=3)->(s:=2),
t_4: (s=3 & p=3)->(s:=4).
Свойства для проверки: $\Psi = \varnothing$.

Рис. 5

Пример работы алгоритма 4 представлен на рис. 5, а также на рис. 6, в котором показаны пути и состояния, порожденные в процессе работы этого алгоритма.

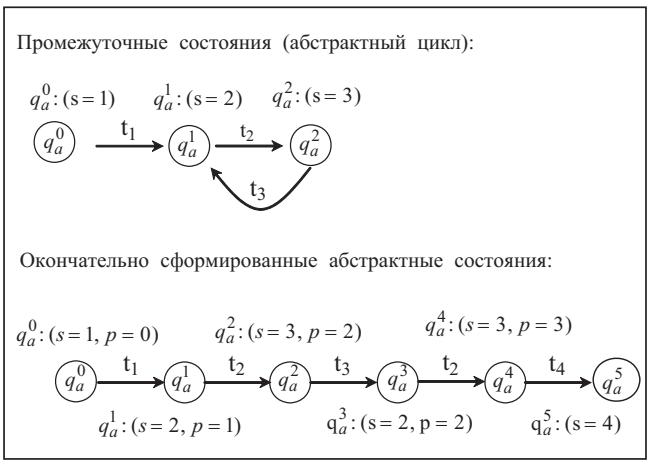

Рис. 6

Лемма 7 (сюръективность абстракции). Каждому абстрактному состоянию соответствует как минимум одно конкретное достижимое состояние: $q_a \in Q_a \Rightarrow$ $\Rightarrow (\exists\, q_c \in \delta(q_c^0) | q_c \supset q_a)$.

Доказательство. Следует непосредственно из того, что все п ройденные алгоритмом 4 состояния достижимы из $\delta(q_c^0)$, а также из того, что согласно определению 12 и процедурам store_visited и form_abstract абстрактное состояние всегда является подмножеством конкретного. □

Лемма 8 (функциональность абстракции). Одному конкретному состоянию соответствует абстрактное, и только одно:

$$(q_a \in Q_a \wedge q'_a \in Q_a \wedge q_c \in \delta(q_c^0) \wedge q_a \subset q_c \wedge q'_a \subset q_c) \Rightarrow (q_a = q'_a).$$

Доказательство. Следует из процедур check_cycle, check_visited, строк их вызовов 15, 21 и строк завершений работы процедуры traverse 20 и 24. □

Следствие 1. Количество достижимых абстрактных состояний не больше достижимых состояний конкретной модели, т.е. $|Q_a| \leq |\delta(q_c^0)|$.

Следствие 2. На множествах Q_c, Q_a можно определить функцию абстракции $Abs: Q_c -> Q_a$. В дальнейшем используется запись $Abs(q_c)$ для обозначения соответствующего абстрактного состояния $q_a \subset q_c$.

Лемма 9. Время работы алгоритма 4: $O(|Q_a| \cdot \max(|\Psi|, |Q_a| \cdot |R| \cdot |A|))$.

Доказательство. Согласно строкам 26 и 32, а также 20 и 24 процедура `traverse` будет вызвана для каждого абстрактного состояния не больше одного раза (вызов в строке 32 осуществляется только для состояний множества `idem`, для которых процедура `traverse` согласно строке 20 не вызывалась).

Согласно строке 2 процедура `property_check` будет вызвана столько же раз, сколько процедура `traverse`, а время ее работы пропорционально количеству проверяемых свойств $|\Psi|$.

Процедура `check_cycle` согласно строке 15 будет вызвана для каждого абстрактного состояния не больше $|R|$ раз (согласно 6, 7, 8, 12), время ее работы $O(|Q_a| \cdot |A|)$, поскольку всякий раз проверяется эквивалентность с каждым состоянием в стеке, размер которого согласно строке 1 может достигать $|Q_a|$.

Процедура `check_visited` согласно строке 21 будет вызвана для каждого абстрактного состояния не больше $|R|$ раз, время ее работы $O(|Q_a| \cdot |A|)$, поскольку эквивалентность проверяется с каждым состоянием из множества пройденных, мощность которого может достигать $|Q_a|$.

Количество вызовов процедуры `form_abstract` согласно строкам 3, 11, 17, 23 и 29 будет равным соответственно $O(|Q_a| \cdot |A|)$, $O(|Q_a| \cdot |R| \cdot |A|)$, $O(|Q_a| \cdot |R| \cdot |A|)$ и $O(|Q_a|^2 \cdot |R| \cdot |A|)$, так как в строке 29 вызов будет осуществлен для каждого цикла в графе поведения абстрактной АТС, число которых $O(|Q_a| \cdot |R|)$. Однако при этом очевидно, что для одного абстрактного состояния суммарное время выполнения процедуры `form_abstract`, включая рекурсивные вызовы, не превышает $O(|Q_a| \cdot |A|)$.

Следовательно, в силу конечности множества конкретных состояний алгоритм 4 завершит свою работу за время, пропорциональное произведению количества абстрактных состояний и времени, затрачиваемому на работу процедуры `traverse` (вместе с обработкой всех процедур, вызываемых внутри нее), т.е. $O(|Q_a| \cdot \max(|\Psi|, |Q_a| \cdot |R| \cdot |A|))$. Очевидно, участки алгоритма 4, отличные от процедур `traverse`, `check_cycle`, `form_abstract` и `check_visited`, можно реализовать за это время. Тем самым оценка времени установлена. □

Таким образом, асимптотическая оценка времени работы алгоритма 4 в худшем случае ($Q_a = Q_c$) такая же, как и для алгоритма проверки свойств без построения абстрактных состояний. В худшем случае алгоритм 4 затратит больше времени на столько, сколько потребуется для вызовов процедуры `form_abstract`. Однако необходимо отметить, что (см. примеры ниже) на практике алгоритм может построить абстрактную АТС, рост количества абстрактных состояний в которой будет полиномиальным, тогда как в соответствующей конкретной АТС — экспоненциальным; также возможно использование более оптимальных алгоритмов обнаружения циклов и компонент сильной связности при проверке темпоральных свойств [34, 35].

Теорема 1 (слабая бисимуляция). Алгоритм 4 построит такую абстракцию M_a конкретной АТС M_c, что $(q_c, q'_c \in \delta(q_c^0) \wedge t \in R) \Rightarrow (q_c \xrightarrow{t} q'_c \Leftrightarrow Abs(q_c) \xrightarrow{t} Abs(q'_c))$.

Доказательство. Утверждение теоремы можно разбить на два составляющих:

$$(q_c \in \delta(q_c^0) \wedge t \in R \wedge Abs(q_c) \xrightarrow{t} Abs(q'_c)) \Rightarrow (\exists\, q'_c \in \delta(q_c^0) | q_c \xrightarrow{t} q'_c), \quad \text{(У1)}$$

$$(q_c, q'_c \in \delta(q_c^0) \wedge t \in R \wedge q_c \xrightarrow{t} q'_c) \Rightarrow (Abs(q_c) \xrightarrow{t} Abs(q'_c)). \quad \text{(У2)}$$

Для **доказательства У1** необходимо показать, что

$$(q_a = Abs(q_c) \wedge q_a \models \alpha_t \wedge q_a \models \varphi) \Rightarrow (q_c \models \alpha_t \wedge q_c \models \varphi), \quad (1)$$

$$q_c \in \delta(q_c^0) \wedge q_a \subset q_c \wedge \beta_t(q_a, q'_a) \Rightarrow \exists\, q'_c | \beta_t(q_c, q'_c) \wedge q'_a \subset q'_c. \quad (2)$$

Справедливость свойства (1) вытекает из следствий 1 и 2 леммы 3, а также лемм 7 и 8. Справедливость свойства (2) — следствие того, что все переходы по определению детерминированы.

Для **доказательства У2** вначале необходимо показать, что те абстрактные состояния стека, атрибут `link` которых не меньше `s->deep`, находятся в той же

компоненте сильной связности графа поведения абстрактной АТС, что и абстракция состояния `s`. Для этого достаточно показать [31] индукцией по числу тех вызовов процедуры `traverse`, которые завершили работу, что выполнение условия `s->link = s->deep` в строке 34 по окончании работы `traverse(s)` гарантирует, что абстракция состояния `s` является корнем сильно связной компоненты абстрактной АТС. Строки 9, 18, 27 и 33 повторяют вычисление [31] значения НИЖНЯЯСВЯЗЬ алгоритма Тарьяна. Таким образом, выполнение условия `s->link = s->deep` в строке 34 дает возможность утверждать, что все абстрактные состояния, находящиеся в стеке выше состояния `s`, образуют сильно связную компоненту. Пусть SCC_a — множество таких состояний.

Строки 8, 25, а также процедура `new_state` осуществляют вычисления `W`- и `V`-атрибутов, обеспечивая необходимые предпосылки леммы 5. Доказательство использует индукцию по числу компонент сильной связности графа поведения абстрактной АТС.

Базис индукции. От противного. Пусть У2 нарушено при первом вызове процедуры `store_visited` (q_c^v). Тогда должен существовать путь в конкретной АТС $\pi_c = q_c^0 \xrightarrow{t_0} \ldots, q_c^v, \ldots, q_c^s \xrightarrow{t_s} q_c^{s+1}$ такой, что:

А) $Abs(q_c^s) \notin SCC_a$;

Б) $Abs(q_c^s) \in SCC_a \wedge (\delta(Abs(q_c^s)) = \varnothing \vee Abs(q_c^s) \not\models \alpha_{t_s})$;

В) $Abs(q_c^s) \in SCC_a \wedge Abs(q_c^s) \models \alpha_{t_s} \wedge \beta_{t_s}(q_a^s, q_a^{s+1}) \wedge q_a^s = Abs(q_c^s) \wedge q_a^{s+1} \not\subset q_c^{s+1}$;

Г) $Abs(q_c^s) \in SCC_a \wedge (q_a \in SCC_a \Rightarrow q_c^{s+1} \neq Abs^{-1}(q_a))$.

Опровержение для случая ***А***, очевидно, сводится к опровержению случаев ***Б***, ***В***, ***Г***, которые рассмотрены и опровергнуты ниже.

Для случаев ***Б***, ***В*** и ***Г*** без потери общности можно предположить, что в абстрактной АТС существует путь $\pi_a = q_a^0 \xrightarrow{t_0} \ldots, q_a^v, \ldots, q_a^s \xrightarrow{t_s} q_a^{s+1}$, соответствующий пути π_c, такой, что $q_a^i \subset q_c^i$ и У2 справедливо для всех $i \in (0 \ldots s)$. Пусть У2 нарушено на продолжении этого пути.

В случае ***Б*** возможны следующие варианты:

1) путь π_a не имеет продолжения, т.е. состояние q_a^s терминальное;

2) переход t_s недопустим в состоянии q_a^s, т.е. $q_a^s \not\models \alpha_{t_s}$.

Вариант 1 допускает такие причины: а) нарушено одно из проверяемых свойств; б) состояние q_a^s тупиковое. Первая причина предполагает $\varphi \in \Psi \wedge q_a^s \not\models \varphi$, но поскольку $q_a^s \subset q_c^s$, в силу следствия 2 леммы 3 и строк 2, 3 процедуры `traverse` должно выполняться $\varphi \in \Psi \wedge q_c^s \not\models \varphi$, т.е. состояние q_c^s также нарушает проверяемое свойство и является согласно строке 5 терминальным, что приводит к противоречию. Вторая причина предполагает, что из состояния q_a^s не выполняется ни одно предусловие ни одного перехода, а значит, и перехода t_s. Этот случай рассмотрен в варианте 2.

Вариант 2. Предполагается, что $q_a^s \not\models \alpha_{t_s}$, что могло произойти только в силу таких причин: а) значения атрибутов в состоянии q_a^s таковы, что результатом работы процедуры `precond`$[t_s]$ (q_a^s) стало **`F`**; б) состояние q_a^s не содержит атрибутов, необходимых для выполнения процедуры `precond` $[t_s](q_a^s)$.

Первая причина приводит к противоречию по рассуждениям, аналогичным изложенным в случае а) варианта 1: в силу леммы 3 и строк 6, 10 и 11 процедуры `traverse` можно утверждать, что переход t_s недопустим и в состоянии q_c^s, т.е. $(q_a^s \subset q_c^s \wedge q_a^s \not\models \alpha_{t_s}) \Rightarrow (q_c^s \not\models \alpha_{t_s})$.

Поскольку согласно строкам 6, 10 и 11 процедуры `traverse` абстрактные состояния содержат все необходимые атрибуты для определения допустимости всех

переходов из данного состояния, то вторая причина возможна только в случае, если процедура `traverse` не была вызвана с параметром q_c^s. А это возможно только тогда, когда алгоритм обнаружил цикл или пройденное ранее состояние на участке пути $q_a^v \dots q_c^s$ и не выполнил переход t_s (случай, когда q_a^s — терминальное состояние, рассмотрен в варианте 1). К этой же ситуации обратимся в случае ***Г*** (см. ниже).

Случай ***В*** предполагает, что после выполнения перехода $q_c^s \xrightarrow{t_s} q_c^{s+1}$ выполняется $q_a^{s+1} \not\subset q_c^{s+1}$. Истории значений атрибутов состояния q_a^{s+1} согласно строкам 3, 11, 17, 23 и 29 сформированы. Значений атрибутов множества q_a^s согласно лемме 5 достаточно для однозначного определения значений всех атрибутов множества q_a^{s+1}. А поскольку переходы детерминированы и $q_a^s \subset q_c^s$, то значения атрибутов множества q_a^{s+1} должны совпадать с соответствующими значениями в q_c^{s+1}, т.е. $q_a^{s+1} \subset q_c^{s+1}$, что приводит к противоречию.

В случае ***Г*** необходимо показать, что $q_c^s \in \delta(q_c^v) \Rightarrow Abs(q_c^s) \in SCC_a$, т.е. в момент выполнения строки 39 (т.е. вызова процедуры `store_visited` (q_c^v)) не существует такого состояния q_c^x, достижимого из состояния q_c^v, абстракция которого еще не построена (рис. 7).

Поскольку по предположению базиса индукции вызов процедуры `check_visited` не мог стать причиной завершения абстрактного пути, а остальные случаи рассмотрены в вариантах ***Б*** и ***В***, то предполагается, что состояние q_c^x не достигнуто, поскольку процедура `traverse` завершила работу вследствие обнаружения абстрактного цикла в результате вызова процедуры `check_cycle` в строке 15 и последующего возврата в строке 20.

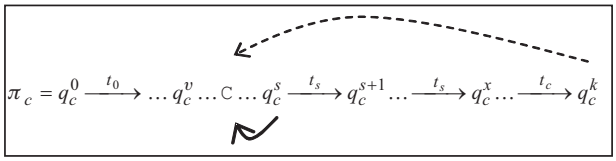

Рис. 7

Пусть процедура `traverse` завершила работу на участке пути $q_c^v \dots q_c^x$ (т.е. $v < x$), потому что `check_cycle` (q_c^s) обнаружила абстрактный цикл в состоянии `c`. Тогда согласно лемме 6 существует конкретный цикл на продолжении этого пути, т.е. существует $q_c^k = \mathtt{c}$. При этом, согласно условию в строке 34, $q_c^v \mathtt{->link} = q_c^v \mathtt{->deep}$, следовательно, циклов, указывающих на состояния, предшествующие состоянию q_c^v, не найдено. Значит, случай $k < x$ можно не рассматривать, потому что, во-первых, по предположению базиса индукции $v \le k$, во-вторых, согласно леммам 1 и 2 существует простой путь конечной длины, ведущий из q_c^v в q_c^x. Тогда без потери общности можно предположить, что состояние q_c^x есть непосредственно следующее за одним из состояний q_c^s, q_c^{s+z},..., $q_c^{s+n \cdot z}$ (здесь z — длина абстрактного цикла) после выполнения перехода t_s. Необходимо рассмотреть следующие варианты.

1. Переход t_s выполним из состояния q_c^s, а значит, и из `c`. А так как $v \le k$, то согласно строкам 6 и 10 вызов процедуры `precond` $[t_s]$ из состояния `c` был осуществлен до вызова `store_visited` (q_c^v), следовательно, существует такое состояние $q_a^u \in SCC_a$, что $\mathtt{c} \xrightarrow{t_s} q_c^u \wedge q_a^u = Abs(q_c^u)$. При этом необходимо отметить, что в силу детерминированности переходов $q_a^u \subset q_c^x$. Более того, так как строка 19 выполнена и множество `idem` состояния `c` содержит указатель на состояние q_c^s, то согласно строкам вызовов процедуры `form_abstract` множество q_a^s содержит историю всех атрибутов состояния q_a^u. Тогда должно выполняться $q_a^s \xrightarrow{t_s} q_a^u$, что приводит к противоречию, поскольку $q_a^u \subset q_c^x$.

2. Переход t_s не выполним из состояния c. Значит, он должен быть выполним из состояний $q_c^{s+z},\ldots, q_c^{s+n\cdot z}$. Согласно строке 11 множество abstract состояния c содержит все необходимые атрибуты для определения допустимости всех переходов, а значит, после выполнения строк 28 и 29 это множество будет содержать все необходимые атрибуты для выполнения процедуры precond$[t_s]$ в состоянии q_a^s.

Выполнение условия строки 30 будет означать, что найден дополнительный значимый атрибут и множество q_a^s расширено, а также согласно строке 31 повлечет удаление q_c^s из q_a^v ->idem и вызов процедуры traverse(q_c^s) в строке 32. Следовательно, согласно строкам 6, 7, 12 переход t_s будет осуществлен из состояния q_c^s и его абстракция $Abs(q_c^s)$ согласно строке 11 будет построена, а так как значение q_c^v ->link в строке 33 было вычислено и по условию q_c^v ->link $= q_c^v$ ->deep в строке 34 циклов, указывающих на состояния, предшествующие состоянию q_c^v, не найдено, то $Abs(q_c^s) \in SCC_a$, что приводит к противоречию.

Если же условие в строке 30 не выполняется для всех $x \in q_a^v$ ->idem, то так как после выполнения строк 28 и 29 множество q_a^s содержит все необходимые атрибуты для определения допустимости t_s, должно выполняться $q_c^{s+n\cdot z} \not\models \alpha_{t_s}$ для всех $n \geq 0$ (согласно лемме 6 для всех $i = 0,\ldots, z$, $n \geq 0$ выполняется $q_a^{s+i} \subset q_c^{s+n\cdot z+i}$), т.е. переход t_s не выполним из состояний $q_c^{s+z},\ldots, q_c^{s+n\cdot z}$, что приводит к противоречию.

Индукционный переход. Необходимо показать, что утверждение теоремы справедливо после того, как процедура traverse(s) обнаружила пройденное ранее абстрактное состояние, поскольку вызов процедуры check_visited(s_new) в строке 21 приведет к возврату в строке 24. Для этого достаточно показать, что множество abstract состояния s будет содержать истории всех значимых атрибутов, определяющих допустимость всех переходов и выполнения свойств для всех состояний, достижимых из s. По предположению индукции все сформированные до сих пор абстрактные состояния удовлетворяют утверждению теоремы и согласно строкам 34, 37, 40 и процедуре store_visited принадлежат множеству visited. Процедура check_visited(s_new) возвращает множество значимых атрибутов состояния s_new, а строка 23 формирует их истории для состояния s.

Симметричность. В силу сюръективности отношения Abs (по лемме 7) выполняется

$$(q_a, q_a' \in \delta(q_a^0) \wedge t \in R) \Rightarrow$$

$$\Rightarrow (q_a \xrightarrow{t} q_a' \Leftrightarrow \forall q_c \in Abs^{-1}(q_a).\ \ \exists q_c' \in Abs^{-1}(q_a') \mid q_c \xrightarrow{t} q_c').$$

Таким образом, между абстрактной и конкретной АТС существует такое отношение B, что:

$$B(q_c, q_a) = \begin{Bmatrix} \mathbf{T}, \text{ если } q_a = Abs(q_c) \\ \mathbf{F}, \text{ если } q_a \neq Abs(q_c) \end{Bmatrix},\ B(q_a, q_c) = \begin{Bmatrix} \mathbf{T}, \text{ если } q_c = Abs^{-1}(q_a) \\ \mathbf{F}, \text{ если } q_c \neq Abs^{-1}(q_a) \end{Bmatrix}.$$

В силу теоремы 5.4 [31] последним состоянием, которое выступит параметром процедуры store_visited, будет абстракция начального состояния и по завершении работы алгоритма 4 выполняется visited $= Q_a$, таким образом, $B(q_c^0, q_a^0) = \mathbf{T}$. Следовательно, отношение B согласно определению 19 является отношением ослабленной бисимуляции.

Теорема доказана. □

Следствие 1 (тотальность абстракции). Алгоритм построит абстракцию каждого достижимого конкретного состояния: $q_c \in \delta(q_c^0) \Rightarrow \exists q_a \in Q_a \mid q_a = Abs(q_c)$.

Следствие 2 (трассовая эквивалентность). Множества трасс (а значит, и достижимых переходов) абстрактной и конкретной АТС совпадают: $\mathcal{L}(M_a) = \mathcal{L}(M_c)$.

Следствие 3 (правильность обнаружения недетерминизмов). Если конкретная модель содержит достижимый недетерминизм $q_c^n \in \delta(q_c^0)$, то алгоритм найдет $q_a^n \subset q_c^n$ и согласно строке 13 построит соответствующую трассу.

Следствие 4 (правильность обнаружения тупиков). Если конкретная модель содержит достижимое тупиковое состояние $q_c^d \in \delta(q_c^0)$, то алгоритм найдет тупиковое состояние $q_a^d \subset q_c^d$ и согласно строкам 14 и 35 построит соответствующую трассу.

Следствие 5 (правильность обнаружения компонент сильной связности). Пусть $q_c^l \in \delta(q_c^0)$ — произвольное состояние любой компоненты сильной связности графа достижимости конкретной АТС, а $SCC_c \in \delta(q_c^l)$ — множество состояний этой компоненты. Тогда алгоритм 4 найдет такое множество состояний $SCC_a \subset$ `visited`, образующее компоненту сильной связности абстрактной АТС, что $q_c \in SCC_c \Rightarrow$ $\Rightarrow \exists q_a \in SCC_a \,|\, q_a \subset q_c$. Пусть, далее, $q_a^x \in \delta(q_a^0)$ — любое достижимое состояние, не входящее в данную компоненту (т.е. $q_a^x \notin SCC_a$). Тогда справедливо следующее утверждение:

$$q_a \in SCC_a \wedge q_a^x \notin SCC_a \Rightarrow \exists q_c, q_c^x \,|\, q_a \subset q_c \wedge q_a^x \subset q_c^x \wedge q_c^x \notin \delta(q_c).$$

Следствие 6 (правильность обнаружения ливлоков). Если конкретная модель содержит достижимый ливлок $q_c^l \in \delta(q_c^0)$, то алгоритм найдет $q_a^l \subset q_c^l$ и согласно строке 36 построит соответствующую трассу.

Теорема 2. Алгоритм 4 построит точную абстракцию M_a конкретной АТС M_c по отношению к проверяемым свойствам Ψ за время $O(|Q_a| \cdot \max(|\Psi|, |Q_a| \cdot |R| \cdot |A|))$.

Доказательство. Следует из строк 2 и 3 процедуры `traverse`, следствия 2 леммы 3 и теоремы 1. Оценка времени приведена из леммы 9. □

Следовательно, алгоритм не найдет ложных недетерминизмов, тупиков, ливлоков и нарушений заданных для проверки свойств. Таким образом, результаты работы алгоритма достоверны и не нуждаются в уточнении. Важно отметить, что проверка темпоральных формул может быть выполнена [36, 37] путем обнаружения сильно связных компонент поведения модели, что согласно теореме 1 обеспечивается алгоритмом 4.

7. ПРИМЕРЫ И СРАВНИТЕЛЬНЫЙ АНАЛИЗ

Пример 1. Пусть дана программа на языке C++ для CBMC [38, 39] (спецификации на других языках опущены).

```
      int scan = 0; bool key0; … bool keyN;
  scr:
      if(scan==0 && key0==1){scan = scan + 1; goto scr;}…
      if(scan==N && keyN==1){scan = scan + 1; goto scr;}
      if((scan == N+1 && (key0 == 0 ||… keyN == 0)) ||
         (scan < N+1 && (key0 == 1 &&… keyN == 1))
      )goto ERROR;
```

Необходимо проверить, что метка ERROR недостижима.

Результаты экспериментов с некоторыми верификаторами (табл. 1) демонстрируют экспоненциальный рост количества состояний (время работы и объем памяти растет пропорционально состояниям) верификаторов NuSMV, CBMC, SMV, VCEGAR, BLAST и SPIN, и квадратичный для разработанного алгоритма. Более того, экспоненциально увеличивается количество итераций уточнений абстракций в BLAST и VCEGAR, при этом построенные абстракции не улучшают результат — количество состояний растет экспоненциально.

На рис. 8 представлено множество абстрактных состояний примера 1, построенных алгоритмом 4. Атрибут `cf` используется в качестве потока управления.

Таблица 1. Сравнительный анализ для примера 1

Размер задачи	Результаты экспериментов с верификаторами						Алгоритм 4
	NuSMV [5, 40]	CBMC [38, 39]	SMV [7, 41]	VCEGAR [12, 42]	BLAST [43, 44]	SPIN [26, 45]	
	Объем памяти, Мбайты		Число состояний (BDD узлов и т.п.)				
9	9	59	55.740	14.691	8.847.360	5.623	81
10	18	118	165.308	23.155	18.284.544	11.255	100
11	41	231	494.122	39.284	38.449.152	22.519	121
12	87	451	1.479.610	70.854	—	45.047	144
13	220	—	4.434.197	133.187	—	90.103	169
14	616	—	13.295.426	257.025	—	180.215	196

```
N=1: cf=[cf0]            x (key0=0,                    scan=0)
N=2: cf=[cf0,cf1]        x (key0=1,key1=0,             scan=1)
  ...
N=N-1: cf=[cf0,cf1,…]    x (key0=1,key1=1, …keyN=0,    scan=N)
N=N: cf=[cf0,cf1,…,cfN] x (key0=1,key1=1,…,keyN=1,
```

Рис. 8

Пример 2. Пусть дана программа на языке Си (спецификация на языках верификаторов опущена).

```
 int max = 4, c = 1, d = 0, z = 1;
cf1: while(true){
cf2:          if(c < max) c = c + 1;
cf3:          if(c > max + d) c = c + z; }
```

Помимо достижимости переходов, тупиков, ливлоков и недетерминизмов, требуется проверить, что всегда выполняется свойство $c-1<max$. На рис. 9 представлено множество абстрактных состояний, построенных алгоритмом 4. Атрибут cf используется в качестве потока управления; повторяющиеся значения вынесены за скобки.

```
      (max=4) X (d=0) X (
1..3:    cf=[cf1,cf2,cf3] X (c=4);
4..6:    cf=[cf1,cf2,cf3] X (c=3);
7..9:    cf=[cf1,cf2,cf3] X (c=2);
10..11:  cf=[cf1,cf2]     X (c=1) )
```

Рис. 9

Необходимо отметить, что атрибут z не входит ни в одно из 11 состояний, построенных алгоритмом 4. Однако, несмотря на недостижимость последнего оператора, методы типа [6, 9–14] не имеют приемлемого критерия для завершения построения контрпримеров (генерируются предикаты $c>max+d$, $c+z>max+d$, $c+2z>max+d$,…); при ограничении $(c,d,z=0\ldots255)$ NuSMV сообщает об ошибке переполнения; число состояний (и соответственно время выполнения) в экспериментах с BLAST, VCEGAR и SMV заметно уменьшится, если недостижимое присваивание удалить (табл. 2). В отличие от упомянутых верификаторов, алгоритм 4 не имеет зависимости от постусловий недостижимых переходов.

Таблица 2. Сравнительный анализ для примера 2

Модели	Результаты экспериментов с верификаторами			Алгоритм 4
	SMV	VCEGAR	BLAST	
С недостижимым переходом	106.705	104.805	3.330.194	11
Без недостижимого перехода	9	9	404.928	11

8. ДЕКОМПОЗИЦИЯ

Не теряя справедливости основных свойств алгоритмов, можно добиться существенного усовершенствования метода. Для этого следует реализовать разбиение множества значимых атрибутов на состояниях путей (и как следствие, общего хранилища

пройденных состояний) на подмножества по следующему принципу: каждый атрибут параметризируется метками переходов (свойств), которые порождали необходимость его добавления во множество abstract. Так, состояние будет считаться ранее пройденным, если оно пройдено «с точки зрения» всех переходов (свойств), т.е. для всех параметров; аналогично модифицируется обнаружение циклов. Функция H_a^s переопределяется соответственно:

$$H^s_{a(t)} = \begin{cases} \mathtt{a}, & \text{если } \mathtt{a} \notin \mathtt{s->W\_SET} \\ \{\mathtt{s->V\_SET[a]} \cup \mathtt{s->abstract(t)}\}, & \text{если } \mathtt{a} \in \mathtt{s->W\_SET} \end{cases}.$$

Такая декомпозиция обеспечивает эффект редукции метода частичного порядка «на лету».

ЗАКЛЮЧЕНИЕ

В отличие от [4, 6, 9–14], описанный метод всегда выдает достоверные результаты, не нуждается в повторных запусках экспериментов для уточнений и не требует построения абстрактных переходов. Метод полностью автоматический, не подразумевает вмешательства со стороны пользователя, как например в [4, 46]. Основное отличие от существующих систем и методов верификации заключается в том, что абстракции строятся «на лету» в процессе проверки свойств модели, таким образом, метод нечувствителен к зависимостям атрибутов, которые обусловлены недостижимыми переходами и состояниями, как, например, [4–6, 10–14, 23, 32].

Гарантия трассовой эквивалентности абстракции дает возможность использовать описываемый метод совместно с методами частичного порядка и методами, использующими симметрии; сохранение реальных значений значимых атрибутов позволяет совместно применять символьные методы абстракции предикатов и многие другие методы редукции пространства поиска.

Метод применим для проверки распространенных ошибок в моделях большинства программных систем, таких как выход за пределы допустимых значений, переполнение и потеря значимости, деление на ноль, обращение к неинициализированным атрибутам, неоднозначная реакция на воздействующие сигналы, недостижимость функциональности, тупики, а также нарушение условий безопасности и живости поведения системы, сформулированных для проверки пользователем в виде формул темпоральной логики.

Результаты экспериментов продемонстрировали на некоторых примерах понижение сложности выполнения проверки свойств модели с экспоненциальной до полиномиальной. Метод показал существенное сокращение вычислительной сложности верификации — времени и памяти, а также в некоторых случаях показал большую эффективность по сравнению с современными популярными системами верификации, такими как SMV, NuSMV, VCEGAR, BLAST, SPIN, CBMC.

Очевидно, худший случай для метода — модель, в которой для интерпретации переходов и условий безопасности на каждом состоянии пути потребуются все атрибуты. Однако на практике проверки программ предусловия переходов, как правило, небольшие, а в первом конъюнкте предусловий проверяется атрибут потока управления, отсекая тем самым потребность в проверке остальной части предусловий недопустимых переходов.

СПИСОК ЛИТЕРАТУРЫ

1. Godefroid P. Partial-order methods for the verification of concurrent systems — an approach to the state-explosion problem. — Berlin: Springer-Verlag, 1996. — **1032.** — 143 P.
2. Godefroid P. Software model checking: the VeriSoft approach // Formal methods in system design. — Springer Scie., Netherlands. — 2005. — **26**. — P. 77–101.
3. Peled D. Combining partial order reductions with on the fly model checking // J. of Formal Methods in System Design. — 1996. — **8,** N 1. — P. 39–64.
4. Symbolic model checking: 10^20 states and beyond / J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang // Inform. and Comput. — 1992. — **98,** N 2. — P. 142–170.
5. Cimatti A., Clarke E. M., Giunchiglia E. and others. NuSMV 2: An OpenSource tool for symbolic model checking // Proc. of Int. Conf. on Computer-Aided Verification, Copenhagen, Denmark. — 2002. — P. 359–364.
6. Counterexample-guided abstraction refinement for symbolic model checking / E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith // J. of the ACM. — 2003. — **50,** N 5. — P. 752–794.
7. McMillan K. Symbolic model checking. —New York: Kluwer Academ. Publ., 1993. — 216 p.

8. Летичевский А., Капитонова Ю., Волков В. и др. Спецификация систем с помощью базовых протоколов // Кибернетика и системный анализ. — 2005. — № 4. — С. 3–21.
9. Chaki S. A counterexample guided abstraction refinement framework for verifying concurrent C programs // PhD thesis. Carnegie Mellon University, USA. — 2005. — P. 253.
10. Gupta A., Wang C., Kim H. Hybrid CEGAR: combining {upsilon} ariable hiding and predicate abstraction // Proc. of the 2007 IEEE // ACM Int. Conf. on Computer-aided design. — 2007. — P. 310–317.
11. Henzinger T., Jhala R., Majumdar R., Sutre G. Lazy abstraction // Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Program. Languages. — 2002. — **37.** — P. 58–70.
12. Jain H., Kroening D., Sharygina N., Clarke E. VCEGAR: Verilog counterexample guided abstraction refinement // Tools and Algorithms for the Construction and Analysis of Systems. — 2007. — 4 P.
13. Pasareanu C., Pelanek R., Visser W. Predicate abstraction with under–approximation refinement // Logical Methods in Comput. Scie. — 2007. — **3.** — P. 1–22.
14. Yuan Lu. Automatic abstraction in model checking // PhD thesis, Carnegie Mellon Univ. — 2000. — 157 p.
15. Ip C., Dill D. Better verification through symmetry // Formal Methods in System Design. — 1996. — **9.** — P. 41–75.
16. Miller A., Donaldson A., Calder M. Symmetry in temporal logic model checking // ACM Comput. Surv. — 2006. — **38.** — 37 p.
17. Nilsson M. Structural symmetry and model checking // PhD thesis, Uppsala Un-t. — 2005. — 157 p.
18. Ball T., Horwitz S. Slicing programs with arbitrary control frow // Proc. of the First Int. Workshop on Automat. and Algorithmic Debugging. — 1993. — P. 206–222.
19. Graf S., Saidi H. Construction of abstract state graphs with PVS // Proceedings of the 9th Intern. Conf. on Comput. Aided Verificat., LNCS 1254. — 1997. — P. 72–83.
20. Hatcliff J., Dwyer M., Zheng H. Slicing software for model construction // Higher-Order and Symbolic Comput. — 2000. — **13**(4). — P. 315–353.
21. Kesten Y., Pnueli A. Control and data abstraction: The cornerstones of practical formal {upsilon} erification // Int. J. on Software Tools for Technology Transfer. — 2000. — **2,** N 4. — P. 328–342.
22. Kurshan R. Computer-aided verification of coordinating processes. — Princeton Univ., 1994. — 270 p.
23. Lind-Nielsen J., Andersen H., Behrmann G. and oth. Verification of large state/event systems using compositionality and dependency analysis // J. of Formal Methods in System Design. — 2001. — **18,** N 1. — P. 5–23.
24. Millet L., Tietelbaum T. Slicing Promela and its applications to model checking, simulation, and protocol understanding // Proc. of the 4th Intern. SPIN Workshop. — 1998. — 9 p.
25. Wilander J. Modeling and visualizing security properties of code using dependence graphs // Proc. of the 5th Conf. on Software Engineering Research and Practice in Sweden. — 2007. — P. 65–74.
26. Ben-Ari M. Principles of Spin. — N.Y.: Springer-Verlag, 2008. — 216 p.
27. Holzmann G. An analysis of bitstate hashing // Formal Methods in Systems Design. — 1998. — P. 301–314.
28. Керниган Б., Ритчи Д. Язык программирования Си. — 2-е изд. — М.: Вильямс, 2007. — 304 с.
29. Dijkstra E. Guarded commands, nondeterminacy and formal derivation of programs // Com. of the ACM. — 1975. — **18,** N 8. — P. 453–457.
30. Park D. Concurrency and automata on infinite sequences // Proc. of the 5th GI-Conf. on Theoret. Comput. Scie. — London, UK: Springer-Verlag, 1981. — P. 167–183.
31. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979. — 536 с.
32. Letichevsky A., Kapitonova J., Konozenko S. Computations in APS // Theoret. Comput. Scie. — 1993. — **119.** — P. 145–171.
33. Tarjan R. Depth first search and linear graph algorithms // SIAM J. on Comput. — 1972. — **1,** N 2. — P. 146–160.
34. Bloem R., Gabow H., Somenzi F. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps // Formal Methods in System Design. — 2006. — **28.** — P. 37–56.
35. Gerth R., Peled D., Vardi M., Wolper P. Simple on the fly automatic verification of linear temporal logic // Protocol Specificat. Testing and Verificat. — 1995. — P. 3–18.
36. Sistla A., Vardi M., Wolper P. The complementation problem for Buchi automata with application to temporal logic // Theoretical Comput. Scie. — 1987. — N 49. — P. 217–237.
37. Wolfgang T. Automata on infinite objects // Ibid. — 1990. — P. 133–191.
38. Clarke E., Kroening D., Lerda F. A tool for checking ANSI-C programs // Tools and Algorithms for the Construction and Analysis of Systems. LNCS 2988. — 2004. — P. 168–176.
39. http://www.cs.cmu.edu/~modelcheck/cbmc
40. http://nusmv.irst.itc.it
41. http://www.kenmcmil.com/smv/linux
42. http://www.cs.cmu.edu/~modelcheck/vcegar
43. Beyer D., Henzinger T., Jhala R., Majumdar R. The software model checker BLAST // Int. J. Softw. Tools Technol. Transfer. — 2007. — N 9. — P. 505–525.
44. http://mtc.epfl.ch/software–tools/blast
45. http://spinroot.com/spin/whatispin.html
46. Bloem R., Ravi K., Somezi F. Symbolic guided search for CTL model checking // Design Automat. Conf. — 2004. — P. 29–34.

Поступила 15.04.2010