УДК 519.686.2

ОПТИМИЗАЦИЯ ПРОВЕРКИ ВЫПОЛНИМОСТИ ПЕРЕХОДОВ ПРИ ВЕРИФИКАЦИИ ФОРМАЛЬНЫХ МОДЕЛЕЙ

А.В. Колчин

Институт кибернетики им. В.М. Глушкова НАН Украины,
03680, Киев, проспект Академика Глушкова, 40.
E-mail: kolchin_av@yahoo.com

При проверке динамических свойств формальных моделей программ, в которых последовательность выполнения операторов выражена неявно, существенную часть операционного времени верификаторы тратят на анализ выполнимости переходов. В работе предложен метод повышения производительности проверки моделей, суть которого заключается в локализации причины невыполнимости перехода в некотором состоянии и ее использовании для анализа выполнимости в последующих состояниях.

Transitions feasibility analysis becomes a significant part of operation time of verification tools for formal models of programs, which control flow is expressed implicitly. This paper describes a new method of model checking performance improvement. The main idea is a localization of a transition's unsatisfiability reason in some state and it's usage for the satisfiability analysis in subsequent states.

Введение

Актуальной задачей для проверки правильности программных систем является повышение производительности автоматических средств верификации. Алгоритмы современных инструментов проверки формальных моделей оперируют множеством выполнимых переходов на данном состоянии (часто такое множество обозначают «enabled», или «outgoing transitions» [1–4]). Выполнение процедуры установления такого множества требуется на каждом достигнутом состоянии модели; причем, как правило, количество выполнимых (из некоторого состояния) переходов значительно меньше общего количества переходов модели. Для большого множества состояний и переходов эффективность такой процедуры будет критичной: наивный перебор затратит время $O(M_S \cdot M_T)$, где M_S – количество достигнутых состояний, M_T – общее количество переходов модели.

Типичные модели императивных программ состоят из параллельно работающих компонент, в каждой из которых явно выделен поток управления, определяющий последовательность выполнения операторов. Структура потока управления крайне важна для анализаторов моделей (например, [1, 5, 6]): его граф может быть построен статически, что дает возможность построения таблицы соответствия вида:

значение потока управления $\rightarrow$ *множество переходов*,

позволяющей эффективно осуществлять выбор подмножества потенциально выполнимых переходов модели, что существенно повышает производительность верификации, так как на практике множество выполнимых переходов (в некотором состоянии) в среднем состоит из 1–3 элементов, тогда как общее количество переходов модели может исчисляться тысячами. Рассмотрим пример программы 1 (рис. 1), выполнением которой является линейная последовательность операторов. Такая программа позволяет построить граф потока управления и таблицу соответствия для выбора переходов (см. рис. 2 и табл. 1) статически, что сделает процедуру установления выполнимых переходов эффективной. Однако последовательность выполнения операторов может быть задана неявно как, например, в моделях с неупорядоченным множеством переходов [7, 8]. Далее модель программы 1 приведена в виде такого множества переходов (табл. 2). Необходимо отметить, что порядок выполнения операторов `operator<i>` в программе 2 сохранен.

```
Operator1;
…
OperatorN;
```

Рис. 1. Программа 1

```
control_flow1:   Operator1;
…
control_flowN:   OperatorN;
```

Рис. 2. Программа 1 с явно обозначенными значениями потока управления

Таблица 1. Таблица соответствия вида:
значение потока управления $\rightarrow$ *множество переходов*

Значение потока управления	Множество потенциально выполнимых переходов (операторов) модели
control_flow1	Operator1
…	…
control_flowN	OperatorN

Таблица 2. Переходы модели программы 2

Переход	Предусловие	Постусловие
T1	`control_flow1==1`	`control_flow1 = 0; control_flow2 = 1; Operator1`
T2	`control_flow2==1`	`control_flow2 = 0; control_flow3 = 1; Operator2`
	...	
TN	`control_flowN==1`	`control_flowN = 0; OperatorN`

Программа 2 не содержит явно выделенного потока управления, и как следствие, статически построенная таблица соответствия будет неэффективна, так как потребует перебора всех переходов в каждом

достигнутом состоянии. Например, процесс достижения оператора `OperatorN` для программы 2 будет осуществляться так (рис. 3).

```
Начальное состояние: control_flow1 = 1, control_flow2 = 0,…, control_flowN = 0;
if(control_flow1 == 1){control_flow1 = 0; control_flow2 = 1; Operator1;}
if(control_flow1 == 1) -- не выполняется
if(control_flow2 == 1){control_flow2 = 0; control_flow3 = 1; Operator2;}
if(control_flow1 == 1) -- не выполняется
if(control_flow2 == 1) -- не выполняется
if(control_flow3 == 1){control_flow3 = 0; control_flow4 = 1; Operator3;}
...
if(control_flow1 == 1) -- не выполняется
if(control_flow2 == 1) -- не выполняется
...
if(control_flowN == 1){control_flowN = 0; OperatorN;}
```

Рис. 3. Процесс проверки модели с неявно заданной последовательностью выполнения операторов

То есть количество проверок атрибутов `control_flow<i>` будет равным `N·(N+1)/2`, и как следствие, сложность проверки программы так же становится квадратичной. Причем количество проверок невыполнимых условий `N·(N-1)/2`, следовательно, большую часть времени процесс проверки модели был занят поиском очередного выполнимого перехода, неэффективно перебирая невыполнимые условия.

Описание метода

Рассмотрим пример пропозициональной формулы: $X \wedge Y \vee Z$. Непосредственно из ее таблицы истинности следует, что если значение атомарной формулы X ложно, то значение Y не влияет на результат, т. е. значение Y можно не вычислять; аналогично, можно не вычислять значение Z, если X и Y истинны.

Таблица 3. Пример таблицы истинности логической формулы

#	X	Y	Z	$X \wedge Y \vee Z$
1	T	T	T	T
2	F	T	T	T
3	T	F	T	T
4	F	F	T	T
5	T	T	F	T
6	F	T	F	F
7	T	F	F	F
8	F	F	F	F

Относительно невыполнимости, эта таблица (см. табл. 3) позволяет сделать такие выводы: для случая 6 формула будет гарантированно невыполнима до тех пор, пока не изменится либо *X,* либо *Z*; для случая 7 – либо *Y,* либо *Z*. Случай 8 требует изменения атрибутов *X*, *Y* либо атрибута *Z*.

На практике во многих языках программирования (например, в Си), бинарные логические операции 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.

Описываемый метод создает и поддерживает таблицу соответствия «причина невыполнимости $\rightarrow$ множество переходов» динамически, в процессе проверки свойств модели. В случае, если переход невыполним, локализируются предикаты, явившиеся причиной невыполнения, и в таблицу соответствия добавляется элемент *{атрибуты, входящие в предикаты* $\rightarrow$ *имя перехода}*. Далее переход будет гарантировано невыполнимым до тех пор, пока хотя бы один из этих атрибутов не изменит своего значения.

Далее приведены основные формальные определения необходимые для описания алгоритмов.

Пусть задано конечное множество атрибутов $A = v_1, v_2, ..., v_n$ 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 $\{\bigcup_{i=0..|A|}(v_i = d_i) \mid v_i \in A,\ d_i \in D(v_i)\}$.

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

***Определение* 3.** Постусловием $\beta(s, s')$ называется конечное множество присваиваний вида $a := F(s, expr)$, где $a \in A$ атрибут состояния s', $expr$ – выражение над константами D и атрибутами состояния s, F – функция, вычисляющая значение выражения $expr$ в состоянии s.

Не теряя общности, предполагается, что ни пред- ни постусловия: не допускают деления на ноль, выходов за пределы допустимых значений, переполнений и потерь значимости, а так же использования неинициализированных атрибутов. Анализ таких свойств потребует введения вспомогательных атрибутов и проверок; требуемая модификация не является критичной для доказательства основных свойств алгоритмов.

***Определение* 4.** Переходом называется тройка вида (α, t, β), где α – предусловие, t – метка (имя перехода), β – постусловие.

В описании алгоритмов для интерпретации атомарных формул предусловий и функции F будет использоваться функция `interpret(<состояние>, <выражение>)`.

***Определение* 5.** Истинность формулы φ на состоянии s, записывается s|=φ, определяется индуктивно по структуре формулы φ:

s |= a ≡ interpret(s,a)= **T** – *если атомарная формула a истинна в состоянии s*;

s |= ~φ ≡ s |≠ φ – *если формула φ не выполняется в s*;

s |= $\varphi_1 \wedge \varphi_2$ ≡ s |= $\varphi_1 \wedge$ s |= φ_2 – *если в s выполняется формула φ_1 и φ_2*;

s |= $\varphi_1 \vee \varphi_2$ ≡ s |= $\varphi_1 \vee$ s |= φ_2 – *если в s выполняется формула φ_1 или φ_2*.

***Определение* 6.** Переход t выполним из состояния s тогда и только тогда когда $s \models \alpha_t$.

***Определение* 7.** Выполнением перехода t из состояния s в состояние s' называется проверка его выполнимости из s согласно определению 6 и модификация s' согласно определению 3.

Для описания алгоритмов будет использоваться язык высокого уровня, состоящий из Упрощенного Алгола [9], и правил переписывания термов языка алгебраического программирования APLAN [10]. Из соображений простоты описания и доказательств, приведенные алгоритмы не претендуют на оптимальность. Далее описан алгоритм, определяющий множество атрибутов, необходимое для вычисления выполнимости перехода.

Алгоритм интерпретации предусловий переходов

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. Текст процедуры precond[t], интерпретирующей предусловие Pre.

```
interpret_pre := procedure(t, Pre) begin
   return `precond[' t `] := procedure(S)local(R_ARR, result, r_idx, i1, i2) begin
              r_idx ← 0;
              ' ! truth_table(Pre, T) ! `
              return (result; {R_ARR[0..r_idx]}) end'
end
```

Кавычки `` ' ' `` используются для выделения строк, знак «!» обозначает операцию конкатенации. Оператор truth_table представляет собой систему переписывающих правил [10], которая сопоставляет левую часть (до знака «=») каждого правила входному терму сверху вниз до первого совпадения, и на выходе строит терм согласно правой части правила; **T**, **F** обозначают соответственно True, False; оператор push(x) означает заталкивание аргумента x в стек; оператор pop(x) означает выталкивание элемента из вершины стека и запись его значения в аргумент x; функция interpret вычисляет значение атомарных формул:

```
truth_table := rewrite_system(a, b, p)begin
1.  (a ∨ b, p) =
1a.              `push(r_idx);
1b.              ' ! truth_table(a, p) ! `
1c.              if (result = ~(' ! p ! `)) then
1d.              begin
1e.                   push(r_idx);
1f.                   ' ! truth_table(b, p) ! `
1g.                   pop(i2);
1h.                   pop(i1);
1i.                   if (result =' ! p ! `) then
1j.                        r_idx ← replace (i1, i2, r_idx)
1k.              end
1l.              else pop(i1);' ,
2.  (a ∧ b, p) =
2a.              `push(r_idx);
2b.              ' ! truth_table(a, p) ! `
2c.              if (result = ' ! p ! `) then
2d.              begin
2e.                   push(r_idx);
2f.                   ' ! truth_table(b, p) ! `
2g.                   pop(i2);
2h.                   pop(i1);
2i.                   if (result = ~(' ! p ! `)) then
2j.                        r_idx ← replace (i1, i2, r_idx)
```

```
2k.                 end
2l.                 else pop(i1);' ,
3.   (~(a), p) = truth_table(a, ~(p)),
4.    (a, T)    = read(a) ! 'result ← interpret (S, ' ! a ! ')',
5.    (a, F)    = read(a) ! 'result ← ~( interpret (S, ' ! a ! '))',
end
```

Процедура `read` строит операторы для добавления в массив `R_ARR` атрибутов, входящих в формулу:

```
read := procedure(atomic_formula)local(r, attr)begin
   r ← '';
   for attr ∈ atomic_formula begin
        r ← r ! ' R_ARR[r_idx] ← ' ! attr;
        r ← r ! ' r_idx ← r_idx + 1;'
   end
   return r
end
```

Процедура `replace` удаляет элементы с индексами `idx1..idx2-1` из массива `R_ARR` путем сдвига элементов влево на (`idx2-idx1`) шагов начиная с позиции `idx2`.

```
replace :=procedure(idx1, idx2, idx)local(i) begin
    for(i ← idx2; i < idx; i ← i + 1) do
        R_ARR[i – (idx2 - idx1)] ← R_ARR[i];
    return (idx - (idx2 - idx1))
end
```

□

На рис. 4 и 5 показаны примеры построения процедуры для переходов `t1` и `t2`.

Вход:
`interpret_pre(t1, x=a+b ∧ y=0)`
Выход:

```
precond[t1] := procedure(S)
      local(R_ARR,result,r_idx,i1,i2)
begin
   r_idx ← 0;
   push(r_idx);
   R_ARR[r_idx] ← x;
   r_idx ← r_idx + 1;
   R_ARR[r_idx] ← a;
   r_idx ← r_idx + 1;
   R_ARR[r_idx] ← b;
   r_idx ← r_idx + 1;
   result ← interpret(S, x=a+b);
   if (result = T) then
   begin
      push(r_idx);
      R_ARR[r_idx] ← y;
      r_idx ← r_idx + 1;
      pop(i2);
      pop(i1);
      result← ~(interpret(S, y=0));
      if(result = ~(T))then
         r_idx←replace(i1,i2,r_idx)
   end
   else pop(i1);
   return (result; {R_ARR[0..r_idx]})
end
```

Рис. 4. Пример 1 работы алгоритма 1

Вход:
`interpret_pre(t2, x=0 ∨ ~(y=0 ∨ z=0))`
Выход:

```
precond[t2] := procedure(S)
        local(R_ARR,result,r_idx,i1,i2)
begin
   r_idx ← 0;
   push(r_idx);
   R_ARR[r_idx] ← x;
   r_idx ← r_idx + 1;
   result ← interpret(S, x=0);
   if (result = ~(T)) then
   begin
      push(r_idx);
      R_ARR[r_idx] ← y;
      r_idx ← r_idx + 1;
      pop(i2);
      pop(i1);
      result ← ~(interpret(S, y=0));
      if(result = ~(F))then
      begin
         push(r_idx);
         R_ARR[r_idx] ← z;
         r_idx ← r_idx + 1;
         pop(i2);
         pop(i1);
         result ← ~(interpret(S, z=0));
         if(result = ~(F))then
            r_idx←replace(i1,i2,r_idx)
       end
       else pop(i1);
   end
   else pop(i1);
   return (result; {R_ARR[0..r_idx]})
end
```

Рис. 5. Пример 2 работы алгоритма 1

В табл. 4, 5 приведены примеры работы процедуры интерпретации предусловий переходов `t1` и `t2`.

Таблица 4. Примеры работы `precond[t1]`

Предусловие `t1`: `x=a+b ∧ y=0`	
Вход - состояние S	**Выход**
`a=0, b=0, x=0, y=0`	`(T, {x,a,b,y})`
`a=0, b=0, x=1, y=0`	`(F, {x,a,b})`
`a=0, b=0, x=0, y=1`	`(F, {y})`

Таблица 5. Примеры работы `precond[t2]`

Предусловие `t2`: `x=0 ∨ ~(y=0 ∨ z=0)`	
Вход - состояние S	**Выход**
`x=0,y=0,z=0`	`(T, {x})`
`x=1,y=0,z=0`	`(F, {x,y})`
`x=1,y=1,z=0`	`(F, {x,z})`

Лемма 1. Алгоритм 1 строит процедуру `precond`, которая:

а) интерпретирует согласно определению 5 логическую формулу `Pre` (удовлетворяющую определению 2), и имеет значение `result` 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 элементов массива `R_ARR[0..r_idx]` после окончания работы процедуры содержит атрибуты, значений которых достаточно для интерпретации формулы `Pre`.

Доказательство:

а) преобразование пропозициональных связок (`truth_table`, правила 1–3) и вычисление результата интерпретации атомарных формул, т. е. вызов функции `interpret` (правила 4, 5) производятся с учетом полярности (атрибут `p` в правилах 1–3 и **T**, **F** – в 4,5) согласно таблицам истинности для логических выражений;

б) согласно семантике операторов `if-then` (в строках `1c`, `2c`), вычисление второго операнда (строки `1f`, `2f`) осуществляется только в случае, когда значения первого операнда (вычисленного в строках `1b`, `2b`) не достаточно для определения результата операции;

в) базис рекурсии (правила 4, 5) 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 добавляет элементы в массив `R_ARR`, то элементы `R_ARR[0..r_idx]` будут содержать только атрибуты, входящие в операнды, значения которых вычислялись; этих атрибутов, очевидно, достаточно для вычисления результата. Создаваемая процедура не имеет циклов и не содержит рекурсивных вызовов, следовательно, время ее работы имеет линейную зависимость от количества атомарных предикатов и атрибутов, входящих в формулу.

Осталось показать, что все элементы `R_ARR`, удаленные процедурой `replace`, были добавлены в массив только в процессе избыточных вычислений (т. е. соответствующие вызовы процедуры `read` можно было бы не осуществлять). Рассмотрим вызовы процедуры `replace`. В момент вызова в строке `1j`, элементы массива `R_ARR[i1..i2-1]` (согласно строкам `1a`, `1b`, `1e`, `1g`, `1h`, `1l`) содержат только атрибуты, необходимые для вычисления первого дизъюнкта (подформулы `a`), а элементы `R_ARR[i2..r_idx]` (согласно строкам `1e`, `1f`, `1g`) для вычисления второго (подформулы `b`). Для определения результата `a∨b` оказалось (согласно строкам `1f`, `1i`) достаточно вычисления второго дизъюнкта, следовательно, достаточно значений атрибутов, входящих в подформулу `b`; вычисление подформулы `a`, согласно таблице истинности формулы `a∨b`, в этом случае можно считать избыточным, а значит, элементы `R_ARR[i1..i2-1]` можно удалить.

Аналогично, в момент вызова в строке `2j`, элементы `R_ARR[i1..i2-1]` (согласно строкам `2a`, `2b`, `2e`, `2g`, `2h`, `2l`) содержат только атрибуты, необходимые для вычисления первого конъюнкта (подформулы `a`), а элементы `R_ARR[i2..r_idx]` (согласно строкам `2e`, `2f`, `2g`) для вычисления второго (подформулы `b`). Для определения результата `a∧b` оказалось (согласно строкам `2f`, `2i`) достаточно вычисления второго конъюнкта, следовательно, достаточно значений атрибутов подформулы `b`; вычисление подформулы `a`, согласно таблице истинности `a∧b`, в этом случае можно считать избыточным, а элементы `R_ARR[i1..i2-1]` можно удалить. □

***Следствие* 1.** Результат выполнения `precond[t](S)` соответствует определению 6:

$$S \models \alpha_t \Leftrightarrow \texttt{result=}\mathbf{T} \ \wedge \ S \not\models \alpha_t \Leftrightarrow \texttt{result=}\mathbf{F}.$$

***Следствие* 2.** В случае невыполнимости перехода, никакое изменение состояния `S`, не включающее атрибутов из `R_ARR[0..r_idx]`, не приведет к его выполнимости.

Для краткости, элементы массива `R_ARR[0..r_idx]` будут называться `R`-атрибутами. В случае если переход невыполним, эти элементы представляют собой причину невыполнения. Необходимо отметить, что множество `R`-атрибутов будет содержать не обязательно все атрибуты, входящие в предусловие. Например, для вычисления $f_1 \wedge f_2 \wedge \ldots f_n$ при истинных $f_1, \ldots, f_{n-1}$ и ложном f_n, результатом работы процедуры `precond` будут только атрибуты, входящие в f_n; аналогично, только атрибуты, входящие в f_n, будут результатом для вычисления $\sim(f_1 \vee f_2 \vee \ldots f_n)$ при ложных $f_1, \ldots, f_{n-1}$ и истинном f_n.

Алгоритм 1 будет использоваться для определения причины невыполнимости переходов. Его так же можно использовать для проверяемых свойств модели, выраженных согласно определению 2 (при этом вместо имени перехода первым параметром `precond` нужно указать идентификатор проверяемого свойства).

Постусловие переходов интерпретируются процедурой `postcond[t]`. Она должна вычислять новое состояние модели, а так же формировать множество атрибутов, изменивших свое значение.

Алгоритм 2. Интерпретация постусловий.
Вход. Переход t, множество присваиваний постусловия Post, так же исходное состояние Src, и целевое Dst.
Выход. Процедура postcond[t] для интерпретации присваиваний Post, а так же множество W.

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

Лемма 2. Алгоритм 2 строит процедуру postcond, которая:

а) выполнит все присваивания Post(Src, Dst) согласно определению 3;

б) формирует множество атрибутов W, которым осуществлялось присваивание новых значений.

Доказательство:

а) строка 3 предполагает выполнение строки 5 для каждого присваивания; в строке 5 присваивание значения, вычисляемого из (предыдущего) состояния Src, выполняется для атрибута (нового) состояния Dst;

б) согласно строке 2, множество W изначально пустое; атрибуты добавляются во множество W только в строке 4 при условии, что присваиваемое значение новое; строка 3 предполагает выполнение строки 4 для каждого присваивания. □

Алгоритм 2 будет использоваться для определения атрибутов, изменивших свое значение (далее W-атрибутов), а так же для вычисления нового состояния модели. Далее описана процедура для выполнения переходов transit, удовлетворяющая определениям 4 и 7, которая на основании имени перехода Transition и состояния Src формирует множество R-атрибутов, и, если переход допустим, новое состояние Dst, а так же множество W-атрибутов.

```
transit := procedure(Src, Transition, Dst) local(result, R, W) begin
   W ← ∅;
   (result; R) ← precond[Transition](Src);
   if(result = T) then (Dst; W) ← postcond[Transition](Src, Dst);
   return (result; Dst; R; W)
end                                                    □
```

На рис. 6 показаны примеры работы процедуры transit.

Предусловие перехода t1 : (x = a + b ∧ y = 0) **Постусловие перехода t1 :** (a := a - 1; x := 0)	
Вход: transit((a=2,b=0,x=2,y=0), t1, (a=2,b=0,x=2,y=0)) **Выход:** (**T;**(a=1,b=0,x=0,y=0); {x,a,b,y}; {a,x})	**Вход:** transit((a=2,b=0,x=0,y=1), t1, (a=2,b=0,x=0,y=1)) **Выход:** (**F;**(a=2,b=0,x=0,y=1); {y}; ∅)

Рис. 6. Примеры работы процедуры transit

Абстрактный алгоритм верификации

Перед тем, как приступить к описанию алгоритма выбора подмножества потенциально выполнимых переходов, необходимо абстрактно определить алгоритм верификации, оперирующий множеством Enabled_Set. Допустим, он построен на алгоритме поиска в глубину (например, [2]).

```
traverse := procedure(S)local(Enabled_Set, t, explored, S_new, res, R, W) begin
a1.   if (visited(S) = T) then return; else visited(S) ← T;
a2.   check_properties(S);
a3.   Enabled_Set ← Get_Enabled(S);
a4.   for t ∈ (Enabled_Set \ explored) do begin
a5.      explored ← explored ∪ t;
a6.      S_new ← copy(S);
a7.      (res, S_new, R, W) ← transit(S, t, S_new);
a8.      if (res = T) then traverse(S_new)
      end
a9.   return
   end                                                 □
```

Отношение visited:S→{**T**,**F**} используется для отметки пройденных состояний. Процедура check_properties(S) осуществляет проверку свойств на состоянии S. Процедура Get_Enabled(S) выполняет формирование множества выполнимых переходов из состояния S.

Алгоритм выбора выполнимых переходов

Далее описан алгоритм выбора подмножества потенциально выполнимых переходов. Данный алгоритм подразумевает такие модификации абстрактного алгоритма верификации traverse.

- Вызов процедуры:

1) в начальном состоянии все переходы включаются в ENABLED, а множество DISABLED объявляется пустым; процедура traverse дополняется параметром для множества W-атрибутов (изначально пустого):

```
ENABLED ← All transitions;
DISABLED ← ∅;
new_traverse(initial, ∅).
```

- Во вновь вычисленном новом состоянии S перед инициализацией множества Enabled_Set:

2) все переходы из множества DISABLED, причина невыполнения которых входит во множество W (т. е. атрибутов, изменивших свое значение), заносятся во множество ENABLED, так же все элементы с этим переходом (у одного перехода может быть более чем одна причина невыполнимости) удаляются из DISABLED.

- В процессе работы с множеством Enabled_Set (при выполнении переходов):

3) если переход из множества Enabled_Set оказывается невыполнимым, то он удаляется из ENABLED, заносится во множество DISABLED с указанием причины невыполнимости.

4) для восстановления исходных значений множеств ENABLED, DISABLED используются множества tmp_enabled и tmp_disabled.

Модификации алгоритма содержатся в строках m<i>, строки a<i> соответствуют исходной процедуре traverse.

```
new_traverse := procedure(S, W)local(tmp_enabled, tmp_disabled, r, t, Enabled_Set,
                                     explored, S_new, res, R, W_new, x)begin
a1.   if (visited(S) = T) then return; else visited(S) ← T;
a2.   check_properties(S);
m1.   tmp_enabled ← ∅; tmp_disabled ← ∅;
m2.   for r ∈ W do
m3.      while ∃t|(r,t) ∈ DISABLED do begin
m4.         ENABLED ← ENABLED ∪ t;
m6.         while ∃x|(x,t) ∈ DISABLED do begin
m7.            tmp_enabled ← tmp_enabled ∪ (r,t);
m8.            DISABLED ← DISABLED \ (r,t)
            end
         end
m9.   Enabled_Set ← ENABLED;
a4.   for t ∈ (Enabled_Set \ explored) do begin
a5.      explored ← explored ∪ t;
a6.      S_new ← copy(S);
a7.      (res, S_new, R, W_new) ← transit(S, t, S_new);
a8.      if (res = T) then new_traverse(S_new, W_new);
         else begin
m10.        ENABLED ← ENABLED \ t;
m11.        for r ∈ R do begin
m12.           tmp_disabled ← tmp_disabled ∪ (r,t);
m13.           DISABLED ← DISABLED ∪ (r,t)
            end
         end
      end
m14.  for (r,t) ∈ tmp_enabled do begin
m15.     DISABLED ← DISABLED ∪ (r,t);
m16.     ENABLED ← ENABLED \ t
      end
m17.  for (r,t) ∈ tmp_disabled do begin
m18.     ENABLED ← ENABLED ∪ t;
m19.     DISABLED ← DISABLED \ (r,t)
      end
a9.   return
   end                                                    □
```

Вместо строки a3 множество Enabled_Set теперь формируется в строке m9.

Лемма 3. Для любого вызова new_traverse(S,W), множество ENABLED (DISABLED) в начале работы процедуры равно множеству ENABLED (DISABLED) в конце ее работы.

Доказательство. Можно показать индукцией по числу завершенных вызовов процедуры new_traverse(S,W), что каждое ее выполнение удовлетворяет утверждению леммы.

Базис индукции. Рассмотрим первое завершение работы процедуры в строке a9, когда все переходы из множества Enabled_Set оказались, согласно строкам a4–a8, невыполнимы (случай завершения в строке a1 тривиален). До выполнения строки m14, множества ENABLED и DISABLED могли изменяться в строках m4, m8 и m10, m13. Рассмотрим строки m4, m8: множество tmp_enabled будет содержать, согласно строкам m3, m6, m7, все элементы, удаленные из DISABLED в строке m8, и переходы, добавленные во множество ENABLED строке m4, и только их, так как в строке m1 оно объявляется пустым. Рассмотрим строки m10, m13: множество tmp_disabled будет содержать, согласно строкам m11, m12, все элементы, добавленные во множество DISABLED в строке m13, а так же переход, удаленный из ENABLED в строке m10, и только их, так как в строке m1 оно объявляется пустым. Тогда после выполнения строк m14–m19, множества ENABLED и DISABLED вернутся в исходное состояние, так как, согласно строкам m14, m15, m16 все элементы из множества tmp_enabled будут добавлены в DISABLED и каждый переход удален из ENABLED, а согласно строкам m17, m18, m19 все элементы из множества tmp_disabled будут удалены из DISABLED, и каждый переход добавлен в ENABLED.

Индукционный переход. Достаточно показать, что множества tmp_enabled и tmp_disabled после выполнения соответственно строк m7 и m12 не будут модифицированы где-либо еще до выполнения строк m12–m17. Это справедливо, так как множества tmp_enabled и tmp_disabled являются локальными для процедуры new_traverse, всякий ее рекурсивный вызов порождает новые экземпляры этих множеств, и следовательно, экземпляры любого вызова модифицируются только в указанных строках того же вызова. □

Лемма 4. Для любого вызова new_traverse(S,W) в строке m9 справедливо утверждение: для любого перехода, либо он является элементом множества ENABLED, либо он невыполним в состоянии S и множество DISABLED содержит непустое множество пар (причина, переход) таких, что никакое изменение атрибута, не являющегося причиной, не приведет к выполнимости этого перехода.

Доказательство. Достаточно показать индукцией по числу вызовов процедуры new_traverse(S,W). Случай завершенных вызовов, согласно лемме 3, тривиален. Рассмотрим случай незавершенных вызовов.

Базис индукции. Случай первого вызова тривиален, так как в начальном состоянии множество ENABLED содержит все переходы, а множество DISABLED пустое.

Индукционный переход. По предположению индукции, утверждение леммы выполняется для S и требуется доказать справедливость утверждения в следующем рекурсивном вызове new_traverse(S_new,W_new). Вначале рассмотрим возможные изменения множества ENABLED. Необходимо рассмотреть такие варианты обработки очередного перехода t, выбранного из множества Enabled_Set в строке a4:

1) переход t выполним в состоянии S. Тогда, согласно строке a8, произойдет рекурсивный вызов процедуры new_traverse(S_new, W_new), до выполнения строки m9 которого никакие элементы из ENABLED не удаляются и в DISABLED не добавляются, т. е. истинность утверждения леммы сохраняется при следующем вызове;

2) переход t невыполним в состоянии S (тогда рекурсивный вызов new_traverse обеспечивается наличием во множестве Enabled_Set другого, выполнимого, перехода). Отметим, что это повлечет, согласно строке a8, удаление t из ENABLED (в строке m10) и добавление всего множества пар (r∈R, t) в DISABLED (в строках m11, m12, m13). Этот случай рассмотрен ниже в изменении множества DISABLED.

Далее рассмотрим возможные изменения множества DISABLED. Условием для выполнимости любого перехода, содержащегося в DISABLED, по предположению индукции, может быть только изменение значения как минимум одного атрибута a, являющегося причиной невыполнения, т. е. множество DISABLED содержит пару (a, t). Тогда в следующем рекурсивном вызове new_traverse(S_new, W_new) переход t либо становится выполнимым (а значит, согласно строке a7 и лемме 2, существует a∈W_new), и, согласно строкам m4, m8 добавляется в ENABLED и удаляется из DISABLED, либо, в противном случае, согласно следствию 2 леммы 1, остается невыполнимым, и как следствие, не добавляется в ENABLED и не удаляется из DISABLED.
В обоих случаях индукционный шаг сохраняет истинность утверждения леммы. □

Теорема 1 (о непротиворечивости абстрактному алгоритму верификации). Для любого вызова процедуры new_traverse(S,W), множество Enabled_Set будет содержать все выполнимые из состояния S переходы.

Доказательство. Непосредственно из леммы 4 следует, что в строке m9 все выполнимые в состоянии S переходы принадлежат множеству ENABLED. Следовательно, множество Enabled_Set будет содержать все выполнимые в S переходы, так как оно формируется из множества ENABLED. □

Теорема 2 (об оптимизации абстрактного алгоритма верификации). Для любого рекурсивного вызова new_traverse(S_new, W_new), справедливо утверждение: если некоторый переход невыполним в предыдущем состоянии, и множество W_new не содержит атрибутов, являющихся причиной невыполнения перехода, то этот переход невыполним и в состоянии S_new, а множество Enabled_Set не будет его содержать.

Доказательство. По индукции аналогично доказательству леммы 4: при обработке вызова new_traverse(S_new, W_new) переход t может стать выполнимым, согласно следствию 2 леммы 1, только если существует такое a∈W_new, что (a,t)∈DISABLED. В противном случае, переход не добавляется в ENABLED, и как следствие, не будет принадлежать множеству Enabled_Set. □

Следствие. Пусть E_1 – множество переходов, возвращаемых функцией Get_Enabled(S) в абстрактном алгоритме верификации, E_2 – множество переходов Enabled_Set в строке m9 процедуры new_traverse. Тогда $E_2 \subseteq E_1$.

Оценка дополнительных затрат времени и памяти для построения множеств ENABLED и DISABLED существенным образом зависит от верифицируемой модели. Конечно, можно построить такую модель, в которой в предусловиях переходов будут громоздкие формулы, включающие все атрибуты модели, а постусловия будут так же изменять значения всех возможных атрибутов, тогда в худшем случае время дополнительных затрат будет $O(M_A^2 \cdot M_T)$, где M_T – количество переходов модели, M_A – атрибутов, тогда как прямой перебор потребует время $O(M_T)$. Однако на практике переходы совсем не такие. Назовем модель реалистичной, если она удовлетворяет таким предположениям, наиболее приближенным к реальным моделям:

1) количество элементов множества W при каждом вызове new_traverse – маленькая константа, в реальных моделях в среднем переход изменяет значения 3–4 атрибутов, следовательно, W состоит из 3–4 элементов;

2) количество элементов множества R причин невыполнения переходов – маленькая константа. В реальных моделях в среднем это значение равно 1–2 элементам для каждого перехода;

3) количество переходов, соответствующих одному атрибуту–причине невыполнения – M_T/M_A. Чаще всего это значение распределения не превосходит 3. Однако в моделях с явно выделенным потоком управления ситуация иная; в случаях, когда можно статически выделить атрибут, являющийся потоком управления, его следует обрабатывать отдельно, используя его значение для дополнительной индексации соответствий {*причина невыполнения* → *множество переходов*}, что позволит многократно улучшить распределение.

Теорема 3. Дополнительное время и память, затраченные на вычисление множества Enabled_Set при каждом вызове процедуры new_traverse для реалистичных моделей $O(M_T/M_A)$.

Доказательство. Оценка следует из предположения о реалистичных моделях. Зависимость от распределения M_T/M_A порождается в строке m3, остальные затраты времени и памяти согласно предположению – маленькая константа. □

Можно показать, что метод пригоден и для поиска в ширину, однако недостатком будет неэффективное использование памяти: для каждого состояния нужно хранить различные экземпляры таблиц ENABLED и DISABLED.

Выводы

В лучшем случае метод позволит уменьшить сложность установления множества выполнимых переходов с $O(M_S \cdot M_T^2)$ до $O(M_S \cdot M_T/M_A)$, где M_S – количество сгенерированных состояний, M_T – переходов модели, M_A – атрибутов. На практике, безусловно, лучше применять описанный метод в комбинации со статическим анализом потока управления.

Необходимо отметить, что алгоритм 1 не требует построения нормальных форм для предусловий переходов, а так же улучшает результаты [7], существенно сокращая множество атрибутов, достаточных для определения результата выполнения переходов и заданных свойств.

Описанный метод может быть использован совместно с существующими оптимизациями и методами редукции [1–7] пространства поиска. Для эффективной работы с типами данных, которые имеют большой диапазон допустимых значений, процедура установления множества допустимых переходов может быть усовершенствована путем применения адаптированного метода абстракции предикатов.

1. *Holzmann G.* The model checker SPIN // IEEE transactions on software engineering. – 1997. – Vol. 23. – N 5. – P. 279–295.
2. *Holzmann G., Peled D.* An improvement in formal verification // FORTE 1994 Conference. – 1994. – P. 197–211.
3. *Zhang Y., Rodriguez E., Zheng H., Myers C.* A Behavioral Analysis Approach for Efficient Partial Order Reduction // In proc. of the 13th IEEE International Symposium on High-Assurance Systems Engineering. – 2011. – P. 49–56.

4. *Jussila T.* On Bounded Model Checking of Asynchronous Systems // Doctoral Thesis. Helsinki University of Technology, Laboratory for Theoretical Computer Science. Research report A97. – 2005. – P. 136.
5. *Beyer D., Henzinger T., Jhala R., Majumdar R.* The software model checker BLAST // International Journal Software Tools Technology Transfer. – 2007. – N 9. – P. 505–525.
6. *Zheng H.* Compositional reachability analysis for efficient modular verification of asynchronous designs // IEEE Trans. on CAD of Integrated Circuits and Systems. – 2010. – Vol. 29. – P. 329–340.
7. *Колчин А.В.* Автоматический метод динамического построения абстракций состояний формальной модели // Кибернетика и системный анализ. – 2010. – № 4. – С. 70–90.
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. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979. – С. 536.
10. *Letichevsky A., Kapitonova J., Konozenko S.* Computations in APS // Theoretical Computer Science. – 1993. – Vol. 119. – P. 145–171.