Формальная верификация методом проверки модели

Захаров А.В.

Санкт-Петербургский государственный политехнический университет

9 апреля 2013

План

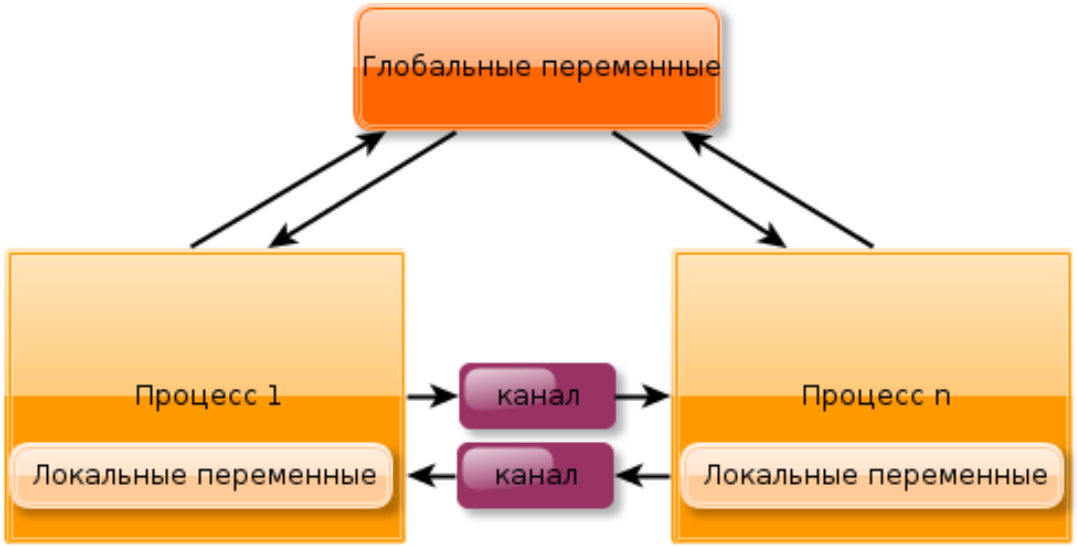
Общие сведения

- PROcess MEta LAnguage
- Входной язык верификатора SPIN
- Недетерминированный язык защищенных команд
- Позволяет задать конечную модель системы

Promela: основные элементы

Модель на языке Promela может включать

- процессы
- глобальные переменные и локальные переменные процессов
- каналы сообщений



Типы данных

- `bit | bool | byte | short | int | unsigned`
- массивы – только одномерные
- `typedef` – аналог структур
- `mtype` – перечисление
- каналы

Примеры

```
typedef Msg {
        byte a[3];
        int b;
        bit ena;
};

mtype = { ack, nak, err, next, accept };
...
Msg message;
mtype msg_type;
```

Каналы

- `chan` *name* = [*size*] `of` {*typename* (`,` *typename*)*}
- *size* $>$ 0 – буферный канал, хранит сообщения
- *size* $=$ 0 – рандеву, синхронный канал

Пример

chan a = [16] of mtype, short, byte ;

Типы процессов

```
active proctype foo(int bar)
{
    // process body
}

init
{
    // init process body
}
```

Основные операторы модели

- присваивание `a = b + c;`
- проверка `assert(x > 0)`
- выражение `x > 0;`
  - исполнимо только когда выполняется условие
  - не имеет side-эффектов
- `printf`
  - как в языке C
  - только во время симуляции
- `if ... fi;`
- `do ... od;`
- `break;`
- `goto LABEL;`
- `run proc();`

Каналы

- `channelname ! sendarg;` – запись в режиме FIFO
- `channelname !! sendarg;` – упорядоченная запись
- `channelname ? recvargs;` – считывает из канала первый элемент, если он подходит по типам
- `channelname ?? recvargs;` – считывает из канала произвольный элемент, который подходит по типам
- `full(channelname)`, `empty(channelname)`, `nempty(channelname)`, `nfull(channelname)` – предопределенные функции для поллинга

Пример модели

```
bool turn;
active proctype A() {
L: printf("A: turn=%d\n", turn);
   (turn == true); // wait for condition
   assert(turn == true);
   turn = false;
   goto L
}
proctype B() {
   do
   :: turn == false -> turn = true;
   od
}
init {
    turn = true; run B();
}
```

Вдохновляющие примеры

- Flood control
- PathStar call processing
- Selected algorithms for Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact
- Control software of the Toyota Camry
- Verification of medical device transmission protocols
- Transactional memory

План

Введение

- Для произвольной программы с неограниченным объемом сохраняемой и читаемой из памяти информации, нет алгоритмических методов для автоматического доказательства всех интересующих свойств. (Задача останова и т.д.)
- Если можно задать конечную границу на объем используемой памяти, то получится система с конечным числом возможных состояний (памяти), которые могут быть пронумерованы.
- Такой подход не используется на практике из-за объема пространства состояний.

Проблема экспоненциального роста пространства состояний

- Пусть имеется K процессов, каждый процесс имеет N состояний
- Тогда общее число состояний произведения получится N^K

Пример

Если каждый процесс имеет 1000 состояний
Если число процессов K=20
Общее число состояний 10^{60}, что соизмеримо с числом атомов во вселенной (10^{78}).

Абстракция

- Вычисления исходной системы содержат много деталей, которые не требуются для верификации
- Модель должна быть **абстрактной** и **адекватной** относительно верифицируемых свойств.

Соответствие между моделью и системой

Стратегии

- Верификация абстрактной модели и синтез реализации путем уточнения модели
- Извлечение абстрактной модели из имеющейся реализации на основе методов абстракции

Методы снижения размерности задачи

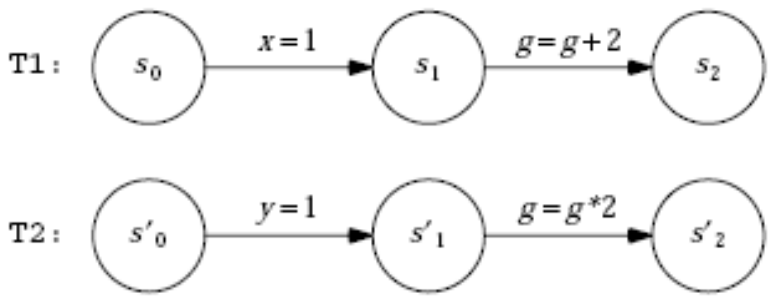
- слайсинг
- алгоритмы проверки модели
  - верификация на-лету
  - использование бинарных решающих диаграмм
  - редукция частичных порядков
- уточнение абстракции на основе контр-примеров

План

Редукция частичных порядков

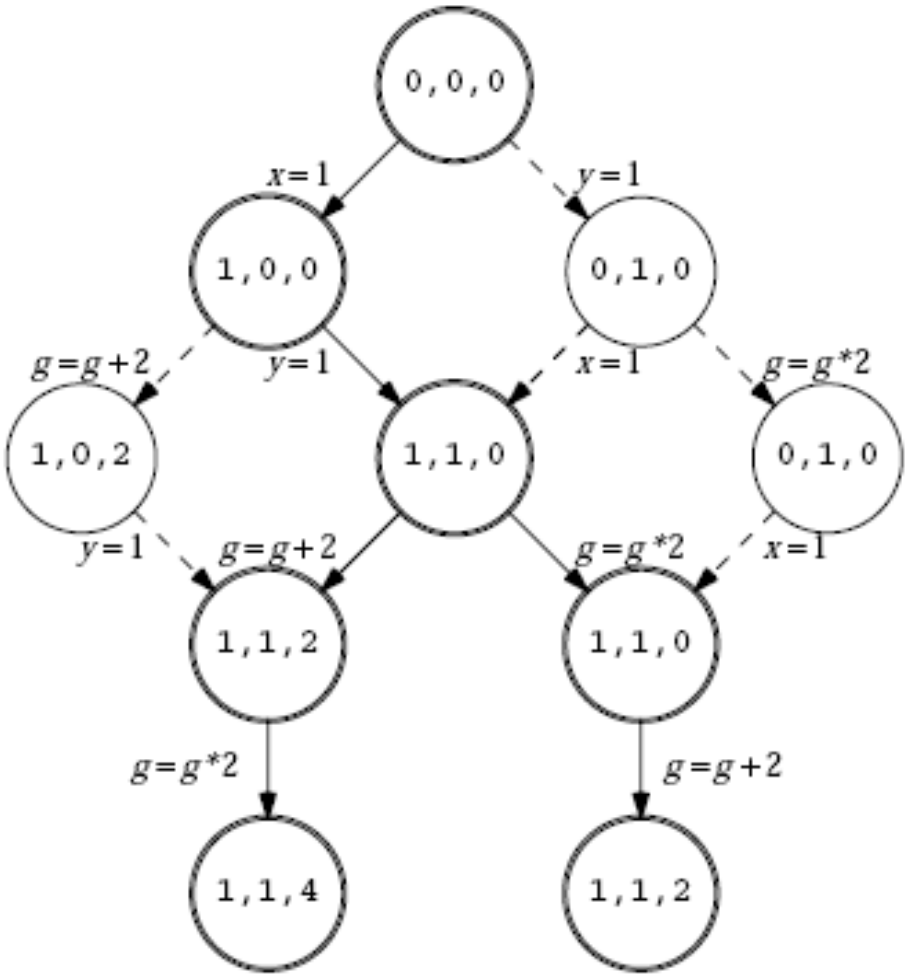
- Проверка на представительной модели
- Рассматриваемая модель выполнения параллельных процессов – интерливинг
- Вычисления в одном процессе и вычисления в разных процессах могут быть зависимыми или независимыми
- Для независимых вычислений результат выполнения не зависит от последовательности выполнения
  - `a = a + 1; b = b * 2`
  - `b = b * 2; a = a + 1`

Пример с двумя процессами



1: $x = 1; g = g+2; y = 1; g = g*2;$
2: $x = 1; y = 1; g = g+2; g = g*2;$
3: $x = 1; y = 1; g = g*2; g = g+2;$
4: $y = 1; g = g*2; x = 1; g = g+2;$
5: $y = 1; x = 1; g = g*2; g = g+2;$
6: $y = 1; x = 1; g = g+2; g = g*2;$

Полный и сокращенный поиски в глубину



Рассматриваемый подход

- Достаточные множества, Ample sets, D. Peled
- $ample(s) \subseteq enabled(s)$
- Что является «достаточным» (насколько можно сокращать множество переходов) – зависит от проверяемых свойств:
  - отсутствие deadlock
  - LTL_{-X}-свойства
  - LTL-свойства

Система переходов

Система переходов LTS=(S, T, S_0, L)

- S, S_0, L – те же, что и в модели Крипке
- T – множество отношений переходов
- $\forall \alpha \in T, \alpha \subseteq S \times S$
- В общем случае α – отношение, для детерминированных переходов α – функция

Допустимость переходов

- Переход α *допустим*, если $\exists s' : \alpha(s, s')$
- Множество допустимых переходов для состояния s обозначим $enabled(s)$

Поиск в глубину с редукцией

```
procedure expandState(s)
    workSet = ample(s);
    while workSet not empty do
        let a in workSet(s);
        workSet(s) = workSet(s) \ {a};
        s' = a(s);
        if new(s') then
            hash(s')
            set onStack(s');
            expandState(s');
        end;
        createEdge(s, a, s');
    end;
    set completed(s);
```

Независимость

Отношение независимости $I \subseteq T \times T$

- $\forall s \in S, \forall(\alpha, \beta) \in I$
- *Допустимость:* Ни один переход не отменяет допустимость другого.
- $\forall \alpha, \beta \in \mathit{enabled}(s) \Rightarrow \alpha \in \mathit{enabled}(\beta(s))$
- *Коммутативность:* Одинаковый результат при разных порядках выполнения независимых переходов
- $\forall \alpha, \beta \in \mathit{enabled}(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$

Невидимость переходов

Переход называется *невидимым* относительно $AP' \subseteq AP$, если

- $\forall (s, s'), s' = \alpha(s) : L(s) \cap AP' = L(s') \cap AP'$

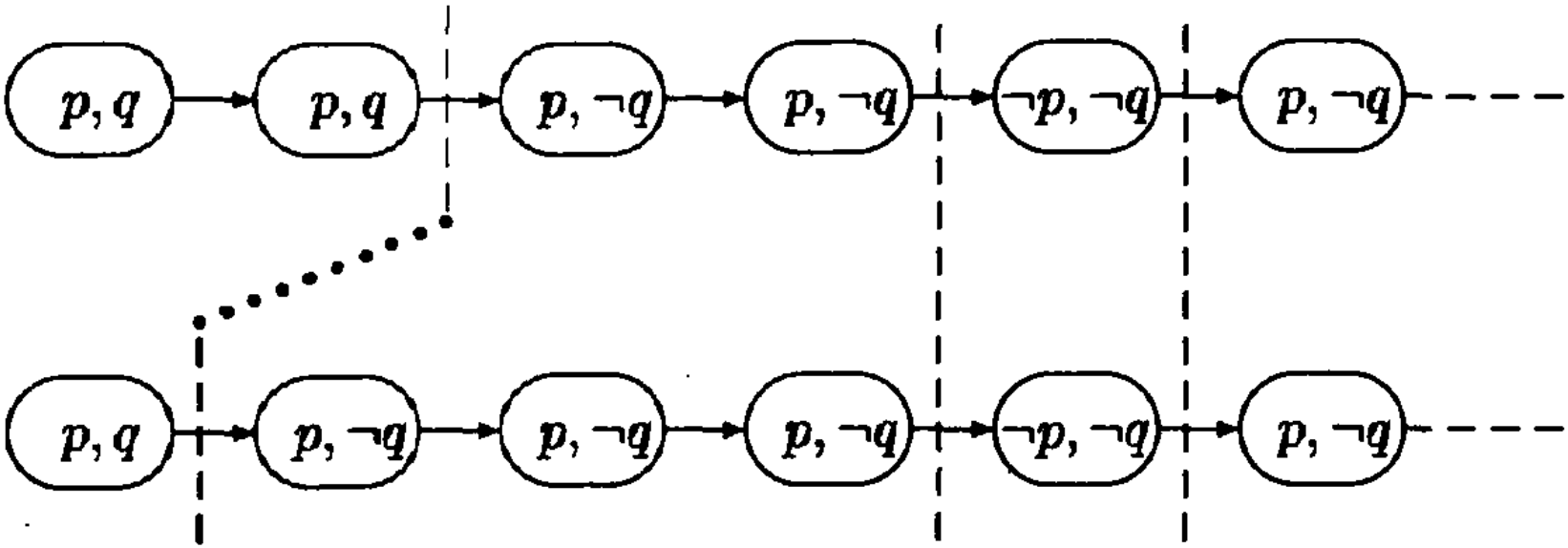
Два пути $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$ и $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} r_2 \xrightarrow{\beta_2} \ldots$ *эквивалентны относительно прореживания*, если есть две последовательности целых чисел

- $0 = i_0 < i_1 < i_2 < \ldots$
- $0 = j_0 < j_1 < j_2 < \ldots$

такие, что

$$\begin{array}{rlll} \forall k \geq 0 \; L(s_{i_k}) = & L(s_{i_k+1}) = & \ldots = & L(s_{i_{k+1}-1}) = \\ L(r_{j_k}) = & L(r_{j_k+1}) = & \ldots = & L(r_{j_{k+1}-1}) \end{array}$$

Пример эквивалентных по прореживанию путей



LTL-формула ϕ *инвариантна относительно прореживания*, если

- $\forall \pi, \pi' : \pi \sim_{st} \pi'$ выполняется $\pi \models \phi \Leftrightarrow \pi' \models \phi$

Теорема

Всякое LTL_{-X} свойство инвариантно относительно прореживания

Достаточные множества

Для каждого состояния используется подмножество $ample(s)$ допустимых переходов, для которого выполнены свойства:

1. $ample(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$
2. на каждом пути из состояния s всякий переход, зависящий от перехода из $ample(s)$, не может быть выполнен прежде, чем переход из $ample(s)$
3. если для состояния s не выполняется $enabled(s) = ample(s)$, то каждый переход $\alpha \in ample(s)$ невидим
4. цикл запрещен, если в нем содержится состояние, в котором переход α допустим, но не содержится в $ample(s)$ ни для одного состояния из цикла

Свойства 1-2 сохраняют deadlock в редуцированном графе переходов; свойства 1-4 сохраняют свойства LTL_{-X}

Эффект от редукции частичных порядков

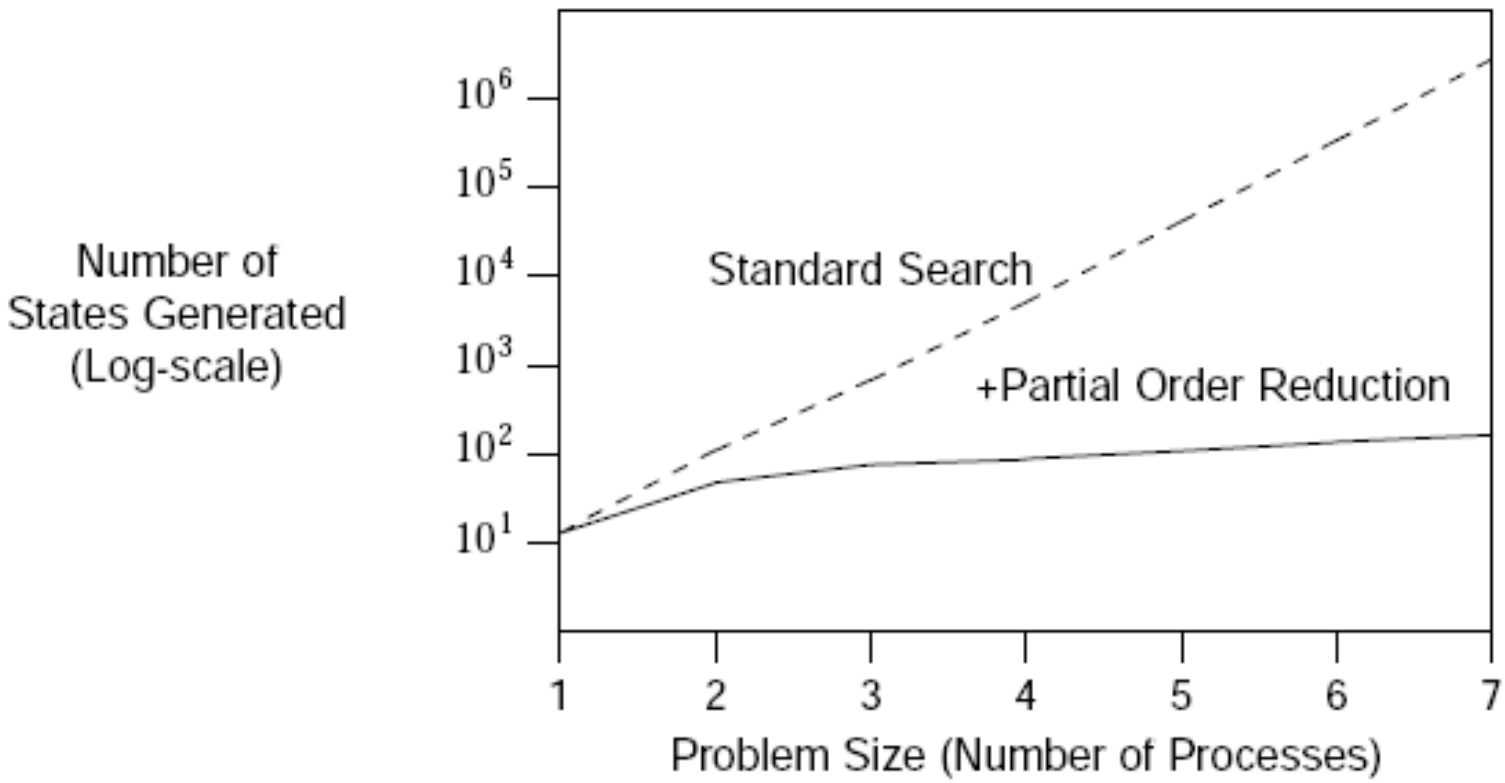


Figure: Leader Election Protocol

План

Идеи

- Множество состояний конечно, отношение переходов конечно
- Элементы конечного множества можно пронумеровать и представить в двоичном виде
- Тогда характристическая функция представима в виде булевой функции
- Отношение переходов - это подмножество декартового произведения, его характеристическую функцию также можно представить в виде булевской функции
- Эффективным мат. аппаратом для представления булевских функций и операций над ними являются *бинарные решающие диаграммы*

Представление состояний

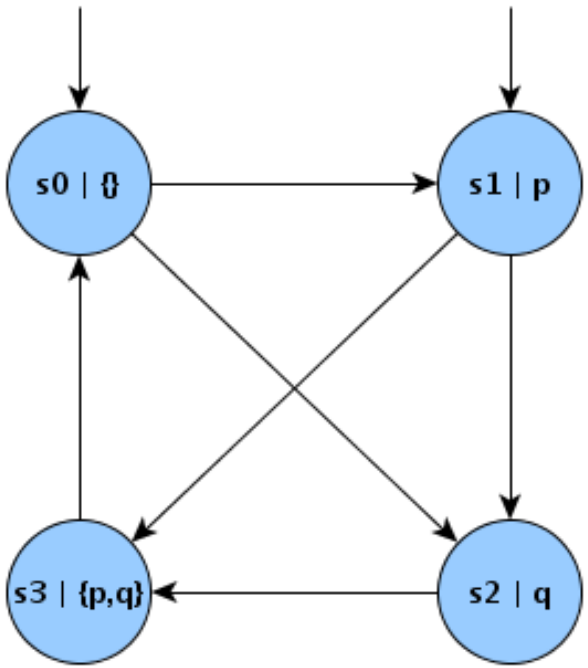


Figure: Пример Модели

Table: Кодирование состояний

x_1, x_2	f_S	f_{S_0}	Состояние
00	1	1	s0
01	1	1	s1
10	1	0	s2
11	1	0	s3

Представление состояний

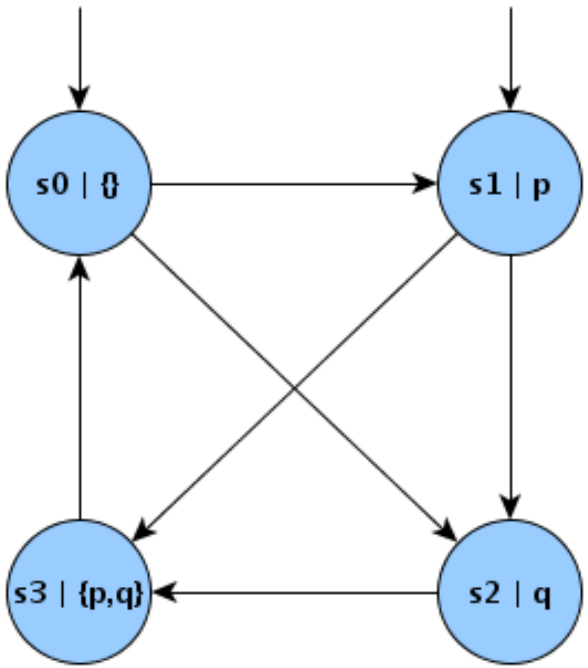


Figure: Пример Модели

Table: Кодирование состояний

x_1, x_2	f_p	f_q
00	0	0
01	1	0
10	0	1
11	1	1

Представление переходов

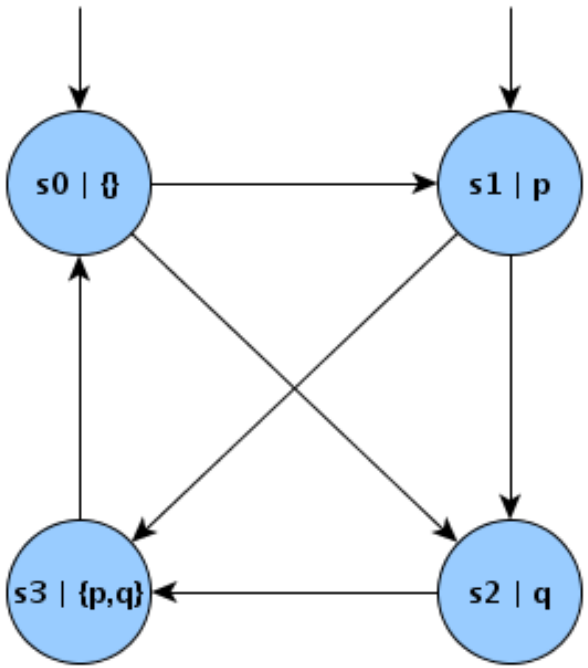


Figure: Пример Модели

Table: Кодирование переходов

x_1, x_2, x_1', x_2'	f	Переход
0000	0	-
0001	1	s0 s1
0010	1	s0 s2
0011	0	-
...		
1111	0	-

Двоичное разрешающее дерево

$$f(x_1, x_2, x_3) = x_1 x_2 \vee x_3$$

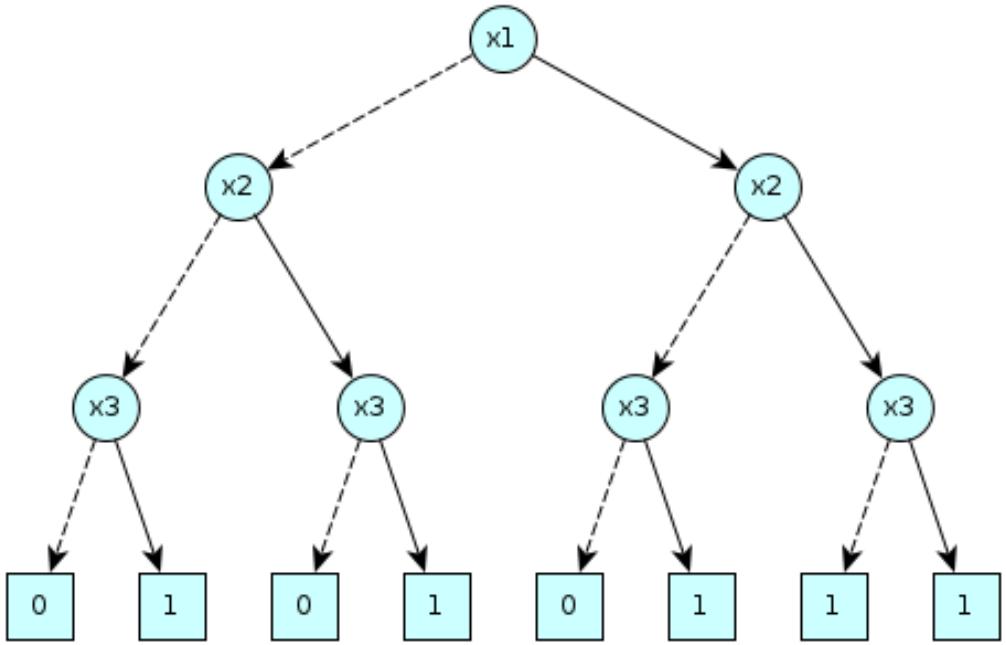


Figure: Двоичное разрешающее дерево

Двоичная разрешающая диаграмма

- $f(x_1, x_2, x_3) = x_1 x_2 \vee x_3$
- Задан порядок переменных $x_1 < x_2 < x_3$

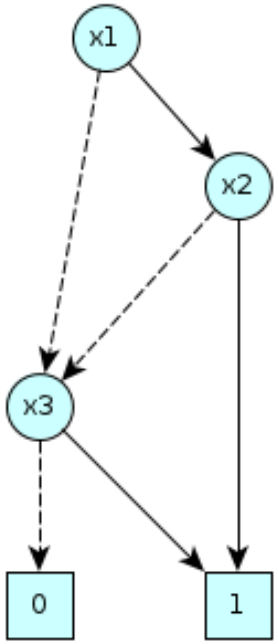


Figure: Двоичная разрешающая диаграмма

Упорядоченные двоичные разрешающие диаграммы

Построение OBDD из семантического дерева

- Удаление повторных вхождений терминалов: остаются 0 и 1
- Удаление повторных вхождений нетерминалов: $var(u) = var(v), low(u) = low(v), high(u) = high(v)$
- Удаление избыточных проверок: $low(v) = high(v)$

Упорядоченные двоичные разрешающие диаграммы

- Каноническое представление булевской функции
- Размер BDD существенно зависит от порядка переменных
- Проверка, является ли порядок оптимальным – NP-полная задача

Упорядоченные двоичные разрешающие диаграммы

$f(x_1, x_2, x_3) = x_1 x_2 \vee x_3$

$x_1 < x_2 < x_3$

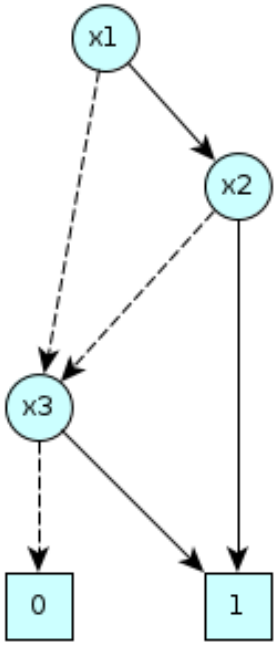


Figure: Двоичная разрешающая диаграмма

$x_1 < x_3 < x_2$

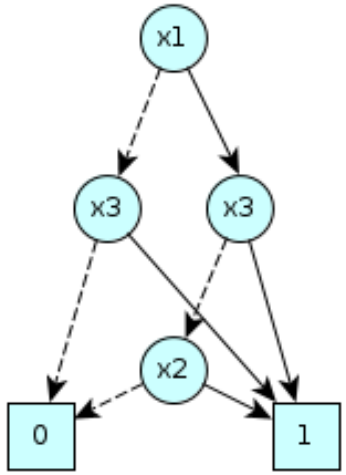


Figure: Двоичная разрешающая диаграмма

Операции над BDD

- Отрицание
- Бинарная булевская операция $f \otimes g$
  - Используем разложение Шеннона
  - $f \otimes g = \neg x(f \otimes g)|_{x=0} \vee x(f \otimes g)|_{x=1}$
  - $f \otimes g = \neg x(f|_{x=0} \otimes g|_{x=0}) \vee x(f|_{x=1} \otimes g|_{x=1})$
  - Операции выполняются рекурсивно, в соответствии с правилами

Операции над BDD

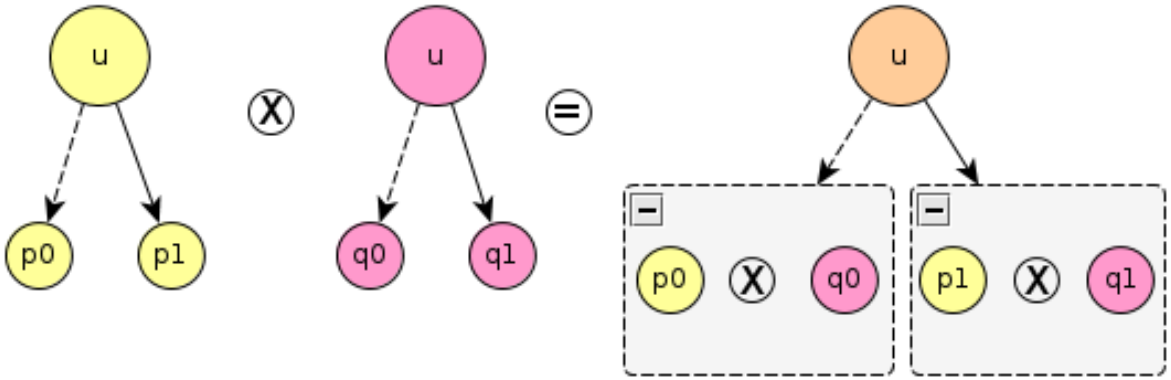


Figure: Правило 1

Операции над BDD

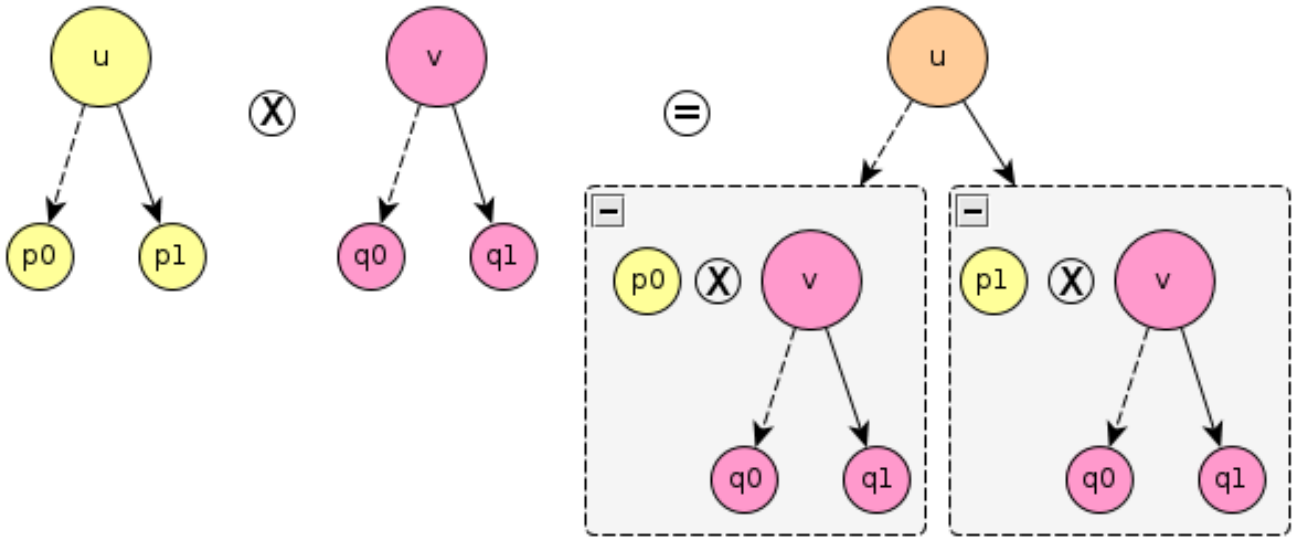


Figure: Правило 2 (u < v)

Достоинства и недостатки BDD

Достоинства:

- Эффективное представление ЛФ в памяти
- Позволяют быстро вычислять значения ЛФ и операции над ними
- Решение задачи SAT на BDD – проверка есть ли терминальная вершина 1

Недостатки:

- Порядок переменных влияет на размер
- Построение BDD для ЛФ - затратная операция

Идеи алгоритма верификации

- Используется поиск в ширину
- Состояния, отношения переходов и разметка задаются с помощью BDD
- Темпоральные формулы выражаются в виде операций над BDD
- Для некоторых формул используется поиск наименьшей неподвижной точки
- Вычисляются состояния, в которых выполняется требуемая формула
- Если в начальном состояние формула выполнима, то свойство выполняется

Эффект от применения BDD

- BDD позволяют выполнять верификацию систем до 10^{20} - 10^{30} состояний
- Есть работы, в которых успешно верифицированы системы с числом состояний $\mathbf{10^{130}}$!

План

Абстракция данных

- Дана программа P с переменными $x_1, x_2, ..., x_n$, определенными на множестве D
- Требуется выбрать абстрактный домен A, и отображение $h : D \to A$
- Пример: x – целочисленная переменная, $h(d)$ может быть определено так:
  - $h(d) = a_0$, если $d = 0$
  - $h(d) = a_+$, если $d > 0$
  - $h(d) = a_-$, если $d < 0$

Абстракция предикатов

- Отслеживаем только значения предикатов $p_1(s), p_2(s), ..., p_n(s)$ над состоянием программы s
- $h(s) = (p_1(s), p_2(s), ..., p_n(s))$

Пример

```
int main() {
  int i;

  i=0;

  while(even(i))
    i++;
}
```

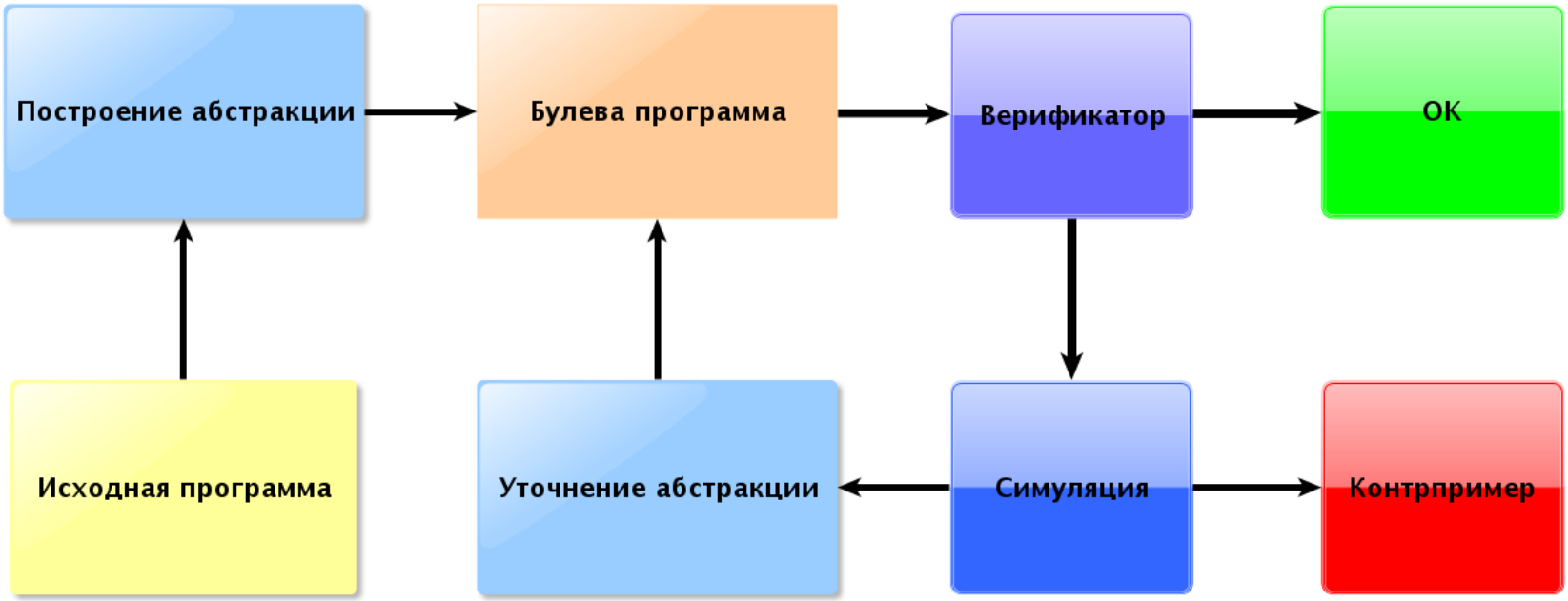
```
p1 = (i = 0)
p2 = (even(i))
```

```
void main() {
  bool p1, p2;

  p1=TRUE;
  p2=TRUE;

  while(p2)
  {
    p1=p1?FALSE:nondet();
    p2=!p2;
  }
}
```

Автоматическое вычисление абстракции: CEGAR



Применения CEGAR

- BLAST
- SLAM
- MAGIC

План

Идеи

- Исходная программа содержит много переменных и операторов, не относящихся к верифицируемому свойству
- Удаление лишних переменных и операторов возможно методом статического анализа – слайсинг.

Постановка задачи

Дана программа P, её CFG и критерий слайсинга $C = \{(n_1, V_1), ..., (n_k, V_k)\}$, где

- n_i - номер вершины
- V_i - множество переменных, интересующих в вершине n_i

Требуется построить выполнимую программу P', которая ведет себя также как P с точки зрения изменения значений указанных переменных.

NB!

Задача построения минимального слайса не разрешима, поэтому пытаются строить аппроксимации минимального слайса.

Обозначения

- $ref(n)$ – множество переменных, используемых в вершине n
- $def(n)$ – множество переменных, определяемых в n
- $infl(b)$ – множество операторов, которые зависят по управлению от оператора ветвления b

Непосредственно релевантные переменные

Множество *непосредственно релевантных переменных* R^0_C - множество всех переменных v, таких что:

1. $(n = n_i) \wedge (v \in V_i)$ для $i \in 1, ..., k$, или
2. вершина n - непосредственный предшественник вершины m, и n удовлетворяет одному из условий:
   1. v сохраняет релевантность в предшественнике
      - $v \notin def(n) \wedge v \in R^0_C(m)$, или
   2. v используется для вычисления релевантной переменной
      - $v \in ref(n)$ и $def(n) \cap R^0_C(m) \neq \emptyset$

.

Непосредственно релевантные операторы

Множество S_C^0 непосредственно релевантных операторов определяется на основе R_C^0 по правилу:
$S_C^0 = \{i | def(i) \cap R_C^0(j) \neq \emptyset, (i,j) \in E\}$.

Неявно релевантные операторы

- $B_C^k = \{b | i \in S_C^k, i \in infl(b)\}$
- $R_C^{k+1}(i) = R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{(b, ref(b))}(i)$
- $S_C^{k+1} = B_C^k \cup \{i | def(i) \cap R_C^{k+1}(j) \neq \emptyset, (i,j) \in E\}$

Пример

```
// C = (10, product)
1:  read(n);
2:  i := 1;
3:  sum := 0;
4:  product := 1;
5:  while i <= n do
    begin
6:    sum := sum + i;
7:    product := product * i;
8:    i := i + 1;
    end;
9:  write(sum);
10: write(product);
```

Пример

```
1:  read(n);                          // def(1) = {n}, ref(1) = {}
2:  i := 1;                           // def(2) = {i}, ref(2) = {}
3:  sum := 0;                         // def(3) = {sum}, ref(3) = {}
4:  product := 1;                     // def(4) = {product}, ref(4) = {
5:  while i <= n do                   // def(5) = {}, ref(5) = {i,n}
    begin
6:    sum := sum + i;                 // def(6) = {sum}, ref(6) = {sum,
7:    product := product * i;         // def(7) = {product}, ref(7) = {
8:    i := i + 1;                     // def(8) = {i}, ref(8) = {i}
    end;
9:  write(sum);                       // def(9) = {}, ref(9) = {sum}
10: write(product);                   // def(10) = {}, ref(10) = {produ
```

Пример

```
1:  read(n);                          // R_C^0 = {}
2:  i := 1;                           // R_C^0 = {}
3:  sum := 0;                         // R_C^0 = {i}
4:  product := 1;                     // R_C^0 = {i}
5:  while i <= n do                   // R_C^0 = {product,i}, infl(5) =
    begin
6:    sum := sum + i;                 // R_C^0 = {product,i}
7:    product := product * i;         // R_C^0 = {product,i}
8:    i := i + 1;                     // R_C^0 = {product,i}
    end;
9:  write(sum);                       // R_C^0 = {product}
10: write(product);                   // R_C^0 = {product}
```

- $S_C^0 = 2, 4, 7, 8, 10$

Пример

```
1:  read(n);                          // R_C^1 = {}
2:  i := 1;                           // R_C^1 = {n}
3:  sum := 0;                         // R_C^1 = {i} U {i,n} = {i,n}
4:  product := 1;                     // R_C^1 = {i} U {i,n} = {i,n}
5:  while i <= n do                   // R_C^1 = {product,i} U {i,n}
    begin
6:    sum := sum + i;                 // R_C^1 = {product,i} U {i,n}
7:    product := product * i;         // R_C^1 = {product,i} U {i,n}
8:    i := i + 1;                     // R_C^1 = {product,i} U {i,n}
    end;
9:  write(sum);                       // R_C^1 = {product}
10: write(product);                   // R_C^1 = {product}
```

- $S_C^1 = S_C^2 = \ldots = 1, 2, 4, 7, 8, 10$

Результаты алгоритма

Критерий слайсинга

$C = (10, product)$

Node #	Def	Ref	Infl	R_C^0	R_C^1
1	{ n }	∅	∅	∅	∅
2	{ i }	∅	∅	∅	{ n }
3	{ sum }	∅	∅	{ i }	{ i, n }
4	{ product }	∅	∅	{ i }	{ i, n }
5	∅	{ i, n }	{ 6, 7, 8 }	{ product, i }	{ product, i, n }
6	{ sum }	{ sum, i }	∅	{ product, i }	{ product, i, n }
7	{ product }	{ product, i }	∅	{ product, i }	{ product, i, n }
8	{ i }	{ i }	∅	{ product, i }	{ product, i, n }
9	∅	{ sum }	∅	{ product }	{ product }
10	∅	{ product }	∅	{ product }	{ product }

Результат слайсинга

```
1:  read(n);
2:  i := 1;

4:  product := 1;
5:  while i <= n do
    begin

7:    product := product * i;
8:    i := i + 1;
    end;
```

Источники

1. Верификация моделей программ: Model Checking [Текст] / Кларк Э.М. Мл., Гра мберг О., Пелед Д. - М. : МЦНМО, 2002. - 416 с. - ISBN 5-94057-054-2
2. Ю.Г. Карпов Model Checking. Верификация параллельных и распределенных прог раммных систем [Текст]. - СПб.: БХВ-Петербург, 2010. - 552 с. - ISBN 978-5-9775- 0404-1
3. G. Holzmann Software Model Checking // In proc. forte. – 1999. – vol 27. – pp. 481-497
4. F. Tip A survey of program slicing techniques // Journal of programming languages. – 1995. – vol. 3. – pp. 121-189
5. `http://www.spinroot.com/spin`