Санкт-Петербургский государственный университет
информационных технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра «Компьютерные технологии»

Кирилл Николаев

Отчет по лабораторной работе «Верификация программ»

Санкт-Петербург
2011

Задание по верификатору SPIN #1

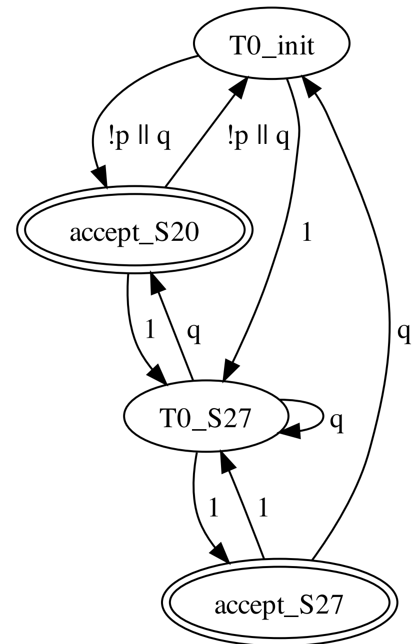
Необходимо сконвертировать формулы темпоральной логики в автомат Бюхи при помощи системы верификации SPIN.

Для преобразования формулы в автомат Бюхи необходимо запустить утилиту `spin` и передать ей в качестве параметров ключ `-f` и текст формулы. Выводом программы будет являться автомат Бюхи в специальном виде (never-clause), который пригоден для использования в текстах программ для SPIN.

G[p → F q]

Данная формула означает, что если на каком-либо шаге работы системы выполнено атомарное предусловие `p`, в будущем обязательно будет выполнено условие `q`.

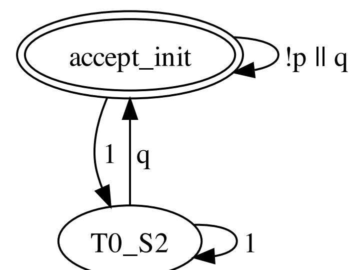
```
$ spin -f '[] (p -> <> q)'
never  {    /* [] (p -> <> q) */
T0_init:
        if
        :: (((! ((p))) || ((q)))) -> goto accept_S20
        :: (1) -> goto T0_S27
        fi;
accept_S20:
        if
        :: (((! ((p))) || ((q)))) -> goto T0_init
        :: (1) -> goto T0_S27
        fi;
accept_S27:
        if
        :: ((q)) -> goto T0_init
        :: (1) -> goto T0_S27
        fi;
T0_S27:
        if
        :: ((q)) -> goto accept_S20
        :: (1) -> goto T0_S27
        :: ((q)) -> goto accept_S27
        fi;
}
```



Для заданной формулы SPIN сгенерировал автомат из четырех состояний, два из которых допускающие. Heikki Tauriainen и Keijo Heljanko в работах [1] и [2] показали, что SPIN в значительной части случаев генерирует некорректный автомат. Для проверки корректности сравним результаты работы SPIN и LTL2BA при помощи программы lbtt [3]. В данном случае оказывается, что автоматы, сгенерированные обоими алгоритмами, эквивалентны, и можно сделать вывод об их корректности.

Возникает вопрос, является ли полученный автомат минимальным. Для получения минимального результата обратимся к программе LTL2BA [4]:

```
never { /* [] (p -> <> q) */
accept_init :    /* init */
        if
        :: (!p) || (q) -> goto accept_init
        :: (1) -> goto T0_S2
        fi;
T0_S2 :    /* 1 */
        if
        :: (1) -> goto T0_S2
        :: (q) -> goto accept_init
        fi;
}
```



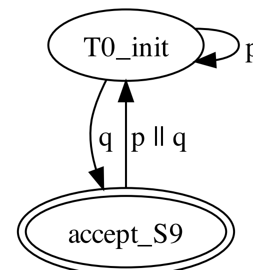
Находясь в состоянии accept_init, автомат ожидает выполнения условия p. Когда это условие выполенено, автомат переходит в состоянии T0_S2 и, находясь в нем, дожидается, когда выполнится требуемое формулой условия q. Затем он возвращается в начальное состояние, и все начинается сначала.

Очевидно, что у автомата должно быть как минимум одно недопускающее состояние и одно допускающее. Поэтому нижнюю границу для размера автомата, соответствующего заданной формуле, можно оценить как 2. Полученный автомат является минимальным, так как он содержит всего два состояния.

G[p U q]

Заданная формула означает, что на каждом шаге верно, что атомарное предусловие p будет выполняться до тех пор, пока не будет выполнено предусловие q.

```
$ spin -f '[] (p U q)'
never  {    /* [] (p U q) */
T0_init:
        if
        :: ((q)) -> goto accept_S9
        :: ((p)) -> goto T0_init
        fi;
accept_S9:
        if
        :: (((p)) || ((q))) -> goto T0_init
        fi;
}
```



Вначале автомат находится в состоянии T0_init и принимает последовательность утверждений о выполнении условия p. Как только выполнено условие q, автомат переходит в допускающее состояние accept_S9. Если после этого q нарушится, автомат перейдет в начальное состояние и снова будет ожидать последовательность символов p.

По соображениям, аналогичным приведенным в первом задании, полученный автомат является минимальным, так как он содержит всего два состояния.

Задание по верификатору SPIN #2

Задание заключается в том, чтобы, используя SPIN, выполнить проверку следующей Promela-спецификации:

```
bit X, Y;
proctype C() {
        do
        :: true -> X = 0; Y = X
        :: true -> X = 1; Y = X
        od
}

proctype monitor() {
        assert(X==Y)
}

init{ atomic{ run C(); run monitor() } }
```

Краткое описание языка Promela можно найти в [5]. Заданная программа в начале исполнения запускает два процесса. Первый циклически меняет значения переменных `X` и `Y`. Он присваивает обоим значение 0, затем обоим же — значение 1, а потом все повторяется с начала. Второй процесс проверяет «согласованность» значений этих переменных, то есть то, что `X` = `Y`.

Для проверки спецификации запустим `spin` и передадим ему заданную программу и опцию `-a` для генерации тестирующей программы на C. Сгенерированная программа после компиляции и запуска выдает следующее заключение:

```
$ ./pan
<...>
pan:1: assertion violated (X==Y) (at depth 4)
pan: wrote spin3.promela.trail
```

Между процессами `C` и `monitor` есть гонка (*race condition*). Переменные `X` и `Y` обновляются не атомарно. При этом процесс `monitor` принимается исполняться как раз между этим обновлениями и наблюдает несогласованные значения переменных.

Можно поместить эти присваивания в *atomic* блок, дабы несогласованные значения не были бы наблюдаемы снаружи. При этом *atomic* блок в процессе `init` можно сделать неатомарным.

```
bit X, Y;

proctype C() {
    do
    :: true -> atomic { X = 0; Y = X }
    :: true -> atomic { X = 1; Y = X }
    od
}

proctype monitor() {
    assert(X==Y);
}

init { run C(); run monitor() }
```

Задание по верификатору NuSMV

Задание заключалось в том, чтобы смоделировать алгоритм взаимного исключения Деккера и проверить выполнение свойств взаимного исключения и отсутствия голодания.

Псевдокод для алгоритма Деккера для *i*-го процесса:

```
while true do
  begin
    bi := true;
    while bj do
      if k = j then
        begin
          bi := false;
          while k = j do skip;
          bi := true
        end;
     ⟨critical section⟩ ;
    k := j;
    bi := false
  end;
```

По аналогии с примером алгоритма взаимного исключения, рассматриваемым в руководстве [6], для моделирования этого алгоритма построим автомат с четырьмя состояниями: *IDLE* (работа вне критической секции), *ENTERING* (ожидание входа в критическую секцию), *CRITICAL* (выполнение критической секции), *EXITING* (выход из критической секции). Главный внутренний цикл алгоритма будет отражен в качестве переходов из состояния *ENTERING*. В зависимости от значений переменных *k* и *b* составного состояния, автомат переходит либо в состояние *CRITICAL*, либо продолжает ожидать выполнения соответствующих условий, оставаясь в том же состоянии.

Для исключения из рассмотрения выполнений программы, которые могут тривиально нарушить требования отсутствия голодания, выдвинем условия *FAIRNESS*, как рекомендуется в документации [7]. Во-первых, каждый процесс должен выполняться бесконечно часто (`FAIRNESS running`), а во-вторых, процесс не должен держать свою блокировку бесконечно (`FAIRNESS state = EXITING`).

Кроме того выдвинем три условия, которые необходимо верифицировать: взаимное исключение (оба процесса не могут одновременно находиться в состоянии *CRITICAL*), и отсутствие голодания каждого из процессов (если процесс когда-либо находится в состоянии *ENTERING*, со временем он попадет в состояние *CRITICAL*). Кроме того, для проверки корректности выполнения добавим условия того, что каждый процесс хоть раз входит в критическую секцию.

```
MODULE main
  VAR
    k : {1, 2};
    b : array 1 .. 2 of boolean;
    proc1 : process user(1, k, b);
    proc2 : process user(2, k, b);
  ASSIGN
    init(b[1]) := FALSE;
    init(b[2]) := FALSE;
  SPEC
    AG !(proc1.state = CRITICAL & proc2.state = CRITICAL);
  SPEC
    AG (proc1.state = ENTERING -> AF proc1.state = CRITICAL);
  SPEC
    AG (proc2.state = ENTERING -> AF proc2.state = CRITICAL);
  SPEC
    AF (proc1.state = CRITICAL)
  SPEC
    AF (proc2.state = CRITICAL)
```

```
MODULE user(i, k, b)
  VAR
    state : { IDLE, ENTERING, CRITICAL, EXITING };
  DEFINE
    j := 3 - i;
  ASSIGN
    init(state) := IDLE;
    next(state) := case
      state = IDLE : { IDLE, ENTERING };
      state = ENTERING & b[j] = FALSE : { CRITICAL };
      state = ENTERING & b[j] = TRUE : { ENTERING };
      state = CRITICAL : { CRITICAL, EXITING };
      state = EXITING : IDLE;
    esac;
    next(b[i]) := case
      state = ENTERING & b[j] = TRUE & k = j : FALSE;
      state = ENTERING : TRUE;
      state = EXITING : FALSE;
      TRUE : b[i];
    esac;
    next(k) := case
      state = EXITING : j;
      TRUE : k;
    esac;
  FAIRNESS
    running;
  FAIRNESS
    state = EXITING;
  FAIRNESS
    state = ENTERING;
```

Вывод NuSMV для данной модели:

```
$ NuSMV -r nusmv2.smv
*** This is NuSMV 2.5.3 (compiled on Mon Jul  4 07:18:04 UTC 2011)
WARNING *** Processes are still supported, but deprecated.      ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
-- specification AG !(proc1.state = CRITICAL & proc2.state = CRITICAL)  is true
-- specification AG (proc1.state = ENTERING -> AF proc1.state = CRITICAL)  is true
-- specification AG (proc2.state = ENTERING -> AF proc2.state = CRITICAL)  is true
-- specification AF proc1.state = CRITICAL  is true
-- specification AF proc2.state = CRITICAL  is true
system diameter: 8
reachable states: 32 (2^5) out of 128 (2^7)
```

Это сообщение подтверждает то, что на нашей модели выполняются все пять предъявленных условий.

Вывод

SPIN и NuSMV позволяют эффективно верифицировать работу распределенных систем, в том числе проверять наличие гонок и голодания. В работе был смоделирован алгоритм взаимного исключения Деккера и произведена проверка его корректности. Положительным моментом при проведении работы стало то, что для рассмотренных примеров системы NuSMV и SPIN производили автоматизированную верификацию меньше, чем за секунду.

Однако в процессе работы выяснилось, что процедуры преобразования LTL-формулы в автомат Бюхи из пакета SPIN иногда приводят к неправильным результатам. Для получения корректных автоматов использовалось средство LTL 2 BA [4] и Spot [8].

Источники

1. Heikki Tauriainen and Keijo Heljanko. Testing LTL Formula Translation into Büchi Automata. 2002
   http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.3280&rep=rep1&type=pdf
2. Heikki Tauriainen and Keijo Heljanko. Testing Spin's LTL Formula Conversion into Büchi Automata with Randomly Generated Input. 2000
   http://spinroot.com/spin/Workshops/ws00/18850055.pdf
3. H. Tauriainen and K. Heljanko. lbtt - an LTL-to-Büchi translator testbench.
   http://www.tcs.hut.fi/Software/lbtt/
4. Denis Oddoux, Paul Gastin. LTL 2 BA : fast translation from LTL formulae to Büchi automata.
   http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php
5. Gerard J. Holzmann. Basic Spin Manual. 2007
   http://spinroot.com/spin/Man/Manual.html
6. Roberto Cavada, Alessandro Cimatti, Gavin Keighren, Emanuele Olivetti, Marco Pistore and Marco Roveri. NuSMV 2.5 Tutorial. 2010.
   http://nusmv.fbk.eu/NuSMV/tutorial/v25/tutorial.pdf
7. Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, Andrei Tchaltsev. NuSMV 2.5 User Manual. 2010
   http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf
8. Alexandre Duret-Lutz, Denis Poitrenaud. Spot: an extensible model checking library using transition-based generalized Büchi automata.
   http://spot.lip6.fr/ltl2tgba.html