Санкт-Петербургский государственный политехнический университет
Институт информационных технологий и управления
Кафедра «Информационные и управляющие системы»

К У Р С О В А Я Р А Б О Т А

Разработка контроллера светофоров и его верификация
по дисциплине «Распределенные алгоритмы и протоколы»

Выполнил
студент гр.63504/12 И.С.Соловьев

Руководитель
ст. преподаватель И.В.Шошмина

«___» _____ 2013 г.

Санкт-Петербург
2013

СОДЕРЖАНИЕ

Введение

В данной курсовой работе ставится задача разработать контроллер светофоров для управления проездом через перекресток. При этом каждое направление движения транспорта на перекрестке регулируется своим светофором. На каждой полосе движения установлены датчики, которые фиксируют появление машины на данной полосе. В отсутствие автомобилей светофор в каждом направлении горит красным. При появлении автомобиля на полосе процесс, отвечающий за движение в данном направлении, пытается предоставить автомобилю возможность проехать через перекресток.

Разработка модели контроллера светофоров ведется на языке Promela в системе Spin.

1. Постановка задачи

Верификация – это проверка того, что продукт удовлетворяет сформулированным требованиям.

Формальная верификация программ – это приемы и методы формального доказательства (или опровержения) того, что модель программной системы удовлетворяет заданной формальной спецификации.

Один из методов верификации – model checking (проверка модели). Это метод проверки того, что на данной формальной модели системы заданная логическая формула выполняется (принимает истинное значение) [1].

В рамках курсовой работы необходимо провести верификацию системы и доказать ее корректность. Для этого необходимо проверить свойства *безопасности*, *живости* и *справедливости* построенной модели [2].

1.1. Вариант задания

Вариант: (2, 5, 13).

Пересечения: ({WN, NE}; {NS, SW}; {NE, ES}).

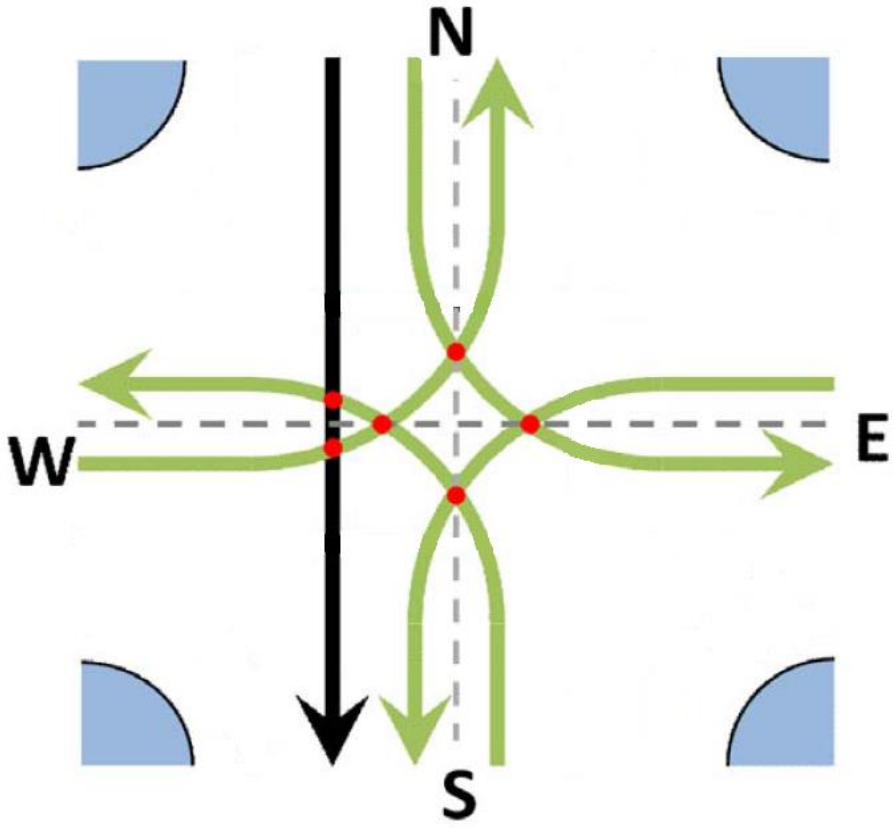

Рисунок 1. Вариант курсовой работы

2. Реализация

2.1. Основная идея реализации модели

Поставленная в данной курсовой работе задача является частным случаем задачи о синхронизации доступа к разделяемым ресурсам. В качестве ресурсов выступают области перекрестка, в которых возможны пересечения различных направлений движения.

Можно выделить следующие ключевые особенности модели:

1. Роль разделяемых ресурсов в модели играют каналы емкостью 1. Такой подход позволяет исключить одновременный проезд автомобилей по пересекающимся направлениям, но, в то же время, позволяет одновременное движение по независимым направлениям.
2. На каждое направление создается свой процесс.
3. Траффик генерируется дополнительным процессом (trafficManager), который имитирует появление машин на разных направлениях случайным образом.
4. Взаимодействие между процессами происходит за счет сигналов.

2.2. Исходный код модели на языке Promela

```
/*
   Task numbers:
   2 - WN, NE;
   5 - NS, SW;
   13 - NE, ES.
*/

/* Traffic lights states */
mtype = {RED, GREEN};

/* Directions mtype */
mtype = {WN, NE, NS, SW, ES};

/* Channels */
chan channel_WN_NS_SW = [1] of {bit};
chan channel_WN_NE = [1] of {bit};
chan channel_NE_ES = [1] of {bit};
chan channel_ES_SW = [1] of {bit};

/* Channel for sensors' states */
```

```
chan sensorChannel = [5] of {mtype, bit};

/* Channels for car passes */
chan carsPassWN = [1] of {bit};
chan carsPassNE = [1] of {bit};
chan carsPassNS = [1] of {bit};
chan carsPassSW = [1] of {bit};
chan carsPassES = [1] of {bit};

/* Traffic manager process */
proctype trafficManager () {
   bool sensorWN = false;
   bool sensorNE = false;
   bool sensorNS = false;
   bool sensorSW = false;
   bool sensorES = false;

end:
   do
      :: (sensorWN && (len (carsPassWN) != 0))->
         carsPassWN ? 1;
wnFalse:
         sensorWN = false;
         sensorChannel ! WN, 0;
      :: (sensorNE && (len (carsPassNE) != 0)) ->
         carsPassNE ? 1;
neFalse:
         sensorNE = false;
         sensorChannel ! NE, 0;
      :: (sensorNS && (len (carsPassNS) != 0)) ->
         carsPassNS ? 1;
nsFalse:
         sensorNS = false;
         sensorChannel ! NS, 0;
      :: (sensorSW && (len (carsPassSW) != 0)) ->
         carsPassSW ? 1;
swFalse:
         sensorSW = false;
         sensorChannel ! SW, 0;
      :: (sensorES && (len (carsPassES) != 0)) ->
         carsPassES ? 1;
esFalse:
         sensorES = false;
         sensorChannel ! ES, 0;
      :: !sensorWN ->
wnTrue:
         sensorWN = true;
         sensorChannel ! WN, 1;
      :: !sensorNE ->
neTrue:
         sensorNE = true;
         sensorChannel ! NE, 1;
```

```
        :: !sensorNS ->
nsTrue:
            sensorNS = true;
            sensorChannel ! NS, 1;
        :: !sensorSW ->
swTrue:
            sensorSW = true;
            sensorChannel ! SW, 1;
        :: !sensorES ->
esTrue:
            sensorES = true;
            sensorChannel ! ES, 1;
    od;
}

/* WN direction process */
proctype dirWN () {
    mtype lightWN = RED;
end:
    do
        ::
        /* Wait for the resources */
        sensorChannel ? WN, 1;
        channel_WN_NS_SW ? 1; channel_WN_NE ? 1;
        /* We can go! */
green:
        lightWN = GREEN;
        carsPassWN ! 1;
        sensorChannel ? WN, 0;
red:
        lightWN = RED;
        /* Return resources */
        channel_WN_NS_SW ! 1; channel_WN_NE ! 1;
    od;
}

/* NE direction process */
proctype dirNE () {
    mtype lightNE = RED;
end:
    do
        ::
        /* Wait for the resources */
        sensorChannel ? NE, 1;
        channel_NE_ES ? 1; channel_WN_NE ? 1;
        /* We can go! */
green:
        lightNE = GREEN;
        carsPassNE ! 1;
        sensorChannel ? NE, 0;
red:
        lightNE = RED;
```

```
        /* Return resources */
        channel_NE_ES ! 1; channel_WN_NE ! 1;
    od;
}

/* NS direction process */
proctype dirNS () {
    mtype lightNS = RED;
end:
    do
        ::
        /* Wait for the resources */
        sensorChannel ? NS, 1;
        channel_WN_NS_SW ? 1;
        /* We can go! */
green:
        lightNS = GREEN;
        carsPassNS ! 1;
        sensorChannel ? NS, 0;
red:
        lightNS = RED;
        /* Return resources */
        channel_WN_NS_SW ! 1;
    od;
}

/* SW direction process */
proctype dirSW () {
    mtype lightSW = RED;
end:
    do
        ::
        /* Wait for the resources */
        sensorChannel ? SW, 1;
        channel_ES_SW ? 1; channel_WN_NS_SW ? 1;
        /* We can go! */
green:
        lightSW = GREEN;
        carsPassSW ! 1;
        sensorChannel ? SW, 0;
red:
        lightSW = RED;
        /* Return resources */
        channel_ES_SW ! 1; channel_WN_NS_SW ! 1;
    od;
}

/* ES direction process */
proctype dirES () {
    mtype lightES = RED;
end:
    do
```

```
        ::
        /* Wait for the resources */
        sensorChannel ? ES, 1;
        channel_ES_SW ? 1; channel_NE_ES ? 1;
        /* We can go! */
green:
        lightES = GREEN;
        carsPassES ! 1;
        sensorChannel ? ES, 0;
red:
        lightES = RED;
        /* Return resources */
        channel_ES_SW ! 1; channel_NE_ES ! 1;
    od;
}

/* Init process */
init {
    /* All resources are freed at start */
    atomic {
        channel_WN_NS_SW ! 1;
        channel_WN_NE ! 1;
        channel_NE_ES ! 1;
        channel_ES_SW ! 1;

        run trafficManager ();

        run dirWN ();
        run dirNE ();
        run dirNS ();
        run dirSW ();
        run dirES ();
    }
}
```

2.3. Пояснения к модели

Процесс **trafficManager** симулирует появления автомобилей на разных направлениях. Он содержит 5 логических переменных, отвечающих за состояние датчиков на каждом из направлений. Метки **wnFalse**, **neFalse**, **nsFalse**, **swFalse** и **esFalse** обозначают состояния процесса, в которых датчик состояния **WN**, **NE**, **NS**, **SW** или **ES** соответственно устанавливается в состояние ЛОЖЬ. Аналогично, метки **wnTrue**, **neTrue**, **nsTrue**, **swTrue** и **esTrue** обозначают состояния процесса, в которых датчики соответствующих состояний устанавливаются в состояние ИСТИНА.

Процессы **dirWN**, **dirNE**, **dirNS**, **dirSW** и **dirES** отвечают за направления **WN**, **NE**, **NS**, **SW** и **ES** соответственно. Логика их работы одинакова и может быть описана следующим образом: сначала процесс ожидает сигнала о том, что его направлении появился автомобиль; затем процесс пытается захватить все необходимые разделяемые ресурсы; когда все ресурсы захвачены, процесс переходит в состояние, отмеченное меткой **green**, и разрешает движение автомобилям; после того, как проедут все автомобили, процесс переходит в состояние, отмеченное меткой **red**, – движение автомобилей в данном направлении запрещено.

Таким образом, алгоритм проезда автомобиля через перекресток можно представить следующим образом (на примере направления WN):

- Процесс trafficManager генерирует сигнал о том, что на направлении WN появился автомобиль.
- Процесс dirWN, отвечающий за заданное направление, принимает сигнал и начинает борьбу за ресурсы.
- Когда все необходимые ресурсы захвачены, процесс dirWN сигнализирует о том, что машины могут проезжать.
- Получив этот сигнал, процесс trafficManager в случайный момент времени имитирует ситуацию, при которой все автомобили в данном направлении проехали через перекресток. Сигнал об этом поступает обратно процессу dirWN.
- Получив последний сигнал, процесс dirWN освобождает ресурсы и снова ожидает появления автомобилей на заданном направлении.

3. LTL-формулы для верификации модели

3.1. Safety

Для данной модели свойства safety на естественном языке звучат следующим образом: «Не должно возникнуть ситуации, при которой зеленый свет зажегся одновременно на двух или более пересекающихся направлениях».

В терминах системы SPIN формула записывается в следующем виде (описание процессов и меток см. в п. 2.3):

```
ltl s {[] !((dirNS@green && dirSW@green && dirWN@green)
|| (dirWN@green && dirSW@green && dirNS@green && dirNE@green)
|| (dirNE@green && dirES@green && dirWN@green)
|| (dirES@green && dirSW@green && dirNE@green)
|| (dirSW@green && dirES@green && dirWN@green &&
dirNS@green))};
```

3.2. Fairness & Liveness

Модель системы, разработанная в данной курсовой работе, является несправедливой, то есть, построена таким образом, что процессы в ней будут выполняться случайным образом. Это может привести к ситуации, при которой один или несколько процессов будут выполняться всегда, в то время как все остальные процессы будут простаивать [4, 5]. Ясно, что в такой ситуации свойства fairness и liveness для простаивающих процессов выполняться не будут.

Таким образом, свойства fairness и liveness необходимо доказывать совместно. На естественном языке данные свойства звучат следующим образом: «При появлении автомобиля на каком-либо направлении ему обязательно представится возможность проехать (возможно, через какое-то время), при ограничении, что в каждом направлении не движется непрерывный поток автомобилей».

В терминах системы Spin ltl формулы для всех направлений записываются следующим образом (описание процессов и меток см. в п. 2.3):

ltl wn {[]<> (([]<> ! (dirWN@green && trafficManager@wnTrue)) -> ([] ((trafficManager@wnTrue && dirWN@red) -> <> dirWN@green)))};

ltl ne {[]<> (([]<> ! (dirNE@green && trafficManager@neTrue)) -> ([] ((trafficManager@neTrue && dirNE@red) -> <> dirNE@green)))};

ltl ns {[]<> (([]<> ! (dirNS@green && trafficManager@nsTrue)) -> ([] ((trafficManager@nsTrue && dirNS@red) -> <> dirNS@green)))};

ltl sw {[]<> (([]<> ! (dirSW@green && trafficManager@swTrue)) -> ([] ((trafficManager@swTrue && dirSW@red) -> <> dirSW@green)))};

ltl es {[]<> (([]<> ! (dirES@green && trafficManager@esTrue)) -> ([] ((trafficManager@esTrue && dirES@red) -> <> dirES@green)))};

Здесь, на примере формулы для направления WN:

- []<> ! (dirWN@green && trafficManager@wnTrue) – свойство fairness, говорящее о том, что поток автомобилей в направлении WN не будет непрерывным.
- [] ((trafficManager@wnTrue && dirWN@red) -> <> dirWN@green – свойство liveness, говорящее о том, что если на направлении WN появится автомобиль, то рано или поздно ему представится возможность для проезда через перекресток.

4. Результаты моделирования

Моделирование показало, что проверка свойств safety и fairness & liveness прошла успешно.

Рассмотрим вывод системы Spin при верификации свойства ne:

```
gcc -DMEMLIM=2048 -O2 -DXUSAFE -w -o pan pan.c
./pan -m200000  -a -N ne
Pid: 2654
Depth=   55866 States=    1e+06 Transitions= 3.73e+06 Memory=   138.249      t=     3.1 R=
3e+05
Depth=   57340 States=    2e+06 Transitions= 7.66e+06 Memory=   212.273      t=    6.34 R=
3e+05
Depth=   57340 States=    3e+06 Transitions= 1.15e+07 Memory=   282.878      t=    9.71 R=
3e+05
Depth=   57340 States=    4e+06 Transitions= 1.53e+07 Memory=   354.363      t=    12.9 R=
3e+05
Depth=   57340 States=    5e+06 Transitions= 1.92e+07 Memory=   421.062      t=    40.1 R=
1e+05
Depth=   57340 States=    6e+06 Transitions= 2.29e+07 Memory=   494.792      t=    43.2 R=
1e+05
Depth=   57340 States=    7e+06 Transitions= 2.63e+07 Memory=   585.808      t=    46.1 R=
2e+05
Depth=   57340 States=    8e+06 Transitions= 2.96e+07 Memory=   676.238      t=    73.1 R=
1e+05
Depth=   57340 States=    9e+06 Transitions= 3.33e+07 Memory=   748.113      t=    76.4 R=
1e+05
Depth=   57340 States=    1e+07 Transitions= 3.71e+07 Memory=   819.988      t=    79.8 R=
1e+05
Depth=   57340 States=  1.1e+07 Transitions=  4.1e+07 Memory=   889.519      t=    83.3 R=
1e+05
Depth=   57340 States=  1.2e+07 Transitions= 4.48e+07 Memory=   959.343      t=    86.7 R=
1e+05
Depth=   57340 States=  1.3e+07 Transitions= 4.89e+07 Memory=  1052.019      t=    93.7 R=
1e+05
Depth=   57340 States=  1.4e+07 Transitions= 5.53e+07 Memory=  1156.023      t=    99.7 R=
1e+05
Depth=   57340 States=  1.5e+07 Transitions= 6.15e+07 Memory=  1260.808      t=     106 R=
1e+05
Depth=   57340 States=  1.6e+07 Transitions= 6.76e+07 Memory=  1367.253      t=     112 R=
1e+05
Depth=   57340 States=  1.7e+07 Transitions= 7.38e+07 Memory=  1470.867      t=     118 R=
1e+05

(Spin Version 6.2.5 -- 3 May 2013)
+ Partial Order Reduction

Full statespace search for:
never claim             + (ne)
assertion violations    + (if within scope of claim)
```

```
acceptance   cycles     + (fairness disabled)
invalid end states      - (disabled by never claim)

State-vector 120 byte, depth reached 57340, errors: 0
 13818309 states, stored (1.77336e+07 visited)
 60479930 states, matched
 78213547 transitions (= visited+matched)
       18 atomic steps
hash conflicts:  13053628 (resolved)

Stats on memory usage (in Megabytes):
 1792.231       equivalent memory usage for states (stored*(State-vector + overhead))
 1477.181       actual memory usage for states (compression: 82.42%)
          state-vector as stored = 96 byte + 16 byte overhead
   64.000       memory used for hash table (-w24)
    6.867       memory used for DFS stack (-m200000)
 1547.527       total actual memory usage
pan: elapsed time 122 seconds
No errors found -- did you verify all claims?
```

Рассмотрим этот вывод более подробно.

Опции компиляции и запуска [3]:

- Флаг *DMEMLIM* задает ограничение памяти в мегабайтах, выделяемой приложению Spin.
- Флаг *DXUSAFE* означает, что мы не проверяем channel assertions – свойство эксклюзивного доступа процессов к каналам (только процесс, в котором описан канал с использованием оператора xr/xs имеет право на чтение/запись в этот канал).
- Флаг *–m200000* устанавливает максимальную глубину посика на отметке в 200000 шагов.
- Флаг *–a* означает, что включена проверка отсутствия циклов с бесконечно частым выполнением операторов с меткой *accept*.
- Флаг *–N ne* означает, что проверяется ltl-формула с именем ne.

Заключение

Язык Promela существенно отличается от всех функциональных и объектно-ориентированных языков программирования. В данном языке нет поддержки функций или указателей, зато он предоставляет простые механизмы обеспечения недетерминизма, а также передачу сообщений между процессами по каналам (в C++ такая функциональность реализована только дополнительными библиотеками Qt и Boost).

Эти особенности делают язык Promela удобным для моделирования достаточно сложных распределенных систем. Код получается лаконичным и понятным.

Однако модели, построенные на языке Promela, не всегда строго соответствуют действительности. Построенная в данной курсовой работе модель контроллера светофоров была несправедливой. На практике это приводило к тому, что при моделировании возникали ситуации, когда контроллер траффика всегда генерировал автомобили только с одного направления. Все остальные направления при этом простаивали. Понятно, что на реальном перекрестке такая ситуация невозможна.

В результате было установлено, что свойства fairness и liveness в таких системах надо доказывать совместно. После этого верификация прошла успешно.

Таким образом, в ходе выполнения данной курсовой работы были получены практические навыки построения моделей распределенных систем на языке Promela и их верификации в системе Spin.

Список использованных источников

1. Карпов Ю.Г. Model Checking. Верификация параллельных и распределенных программных систем – СПб.: БХВ-Петербург, 2009. – 560 с.

2. Карпов Ю.Г., Шошмина И.В. Верификация распределенных систем – СПб.: Издательство Политехнического университета, 2011. – 212 с.

3. Holzmann G.J. The SPIN Model Checker: Primer and Reference Manual – Addison Wesley, 2004. – 610 с.

4. Holzmann G.J. The Model Checker Spin // IEEE Trans. on Software Engineering, Vol. 23, No. 5, май 1997, с. 279-295.

5. Wahl T. Fairness and Liveness // http://www.ccs.neu.edu – Northeastern University official website – URL: http://www.ccs.neu.edu/home/wahl/Publications/fairness.pdf (дата обращения: 06.12.2013).