МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ИНДИВИДУАЛЬНОЕ ДОМАШНЕЕ ЗАДАНИЕ

по дисциплине «Распределенные алгоритмы»

Тема: Алгоритм Деккера для n процессов

Студент гр. 9381 _____ Колованов Р.А.

Преподаватель _____ Шошмина И.В.

Санкт-Петербург

2022

ЗАДАНИЕ
НА ИНДИВИДУАЛЬНУЮ ДОМАШНЮЮ РАБОТУ

Студент Колованов Р.А.

Группа 9381

Тема работы: Алгоритм Деккера для n процессов

Исходные данные:
Язык Promela, инструмент для верификации корректности распределенных программных моделей Spin.

Содержание пояснительной записки:
«Аннотация», «Содержание», «Введение», «Описание рассматриваемых алгоритмов», «Моделирование алгоритмов», «Верификация алгоритмов», «Заключение», «Список использованных источников», «Приложение А. Исходный код алгоритмов на языке Promela»

Предполагаемый объем пояснительной записки:
Не менее 10 страниц.

Дата выдачи задания: 07.04.2022

Дата сдачи отчета: 22.05.2022

Дата защиты отчета: 23.05.2022

Студент _____ Колованов Р.А.

Преподаватель _____ Шошмина И.В.

АННОТАЦИЯ

В данной работе рассматриваются алгоритм Деккера для двух процессов, а также обобщения данного алгоритма для n процессов: алгоритм Питерсона для n процессов (алгоритм фильтра) и алгоритм Дейкстры для n процессов, предназначенные для проблемы критической секции. Для рассматриваемых алгоритмов были реализованы программные модели на языке Promela, а также была выполнена проверка корректности работы этих алгоритмов и верификация их свойств, которая проводилась при помощи инструмента для верификации Spin.

Результатом данной работы являются реализованные программные модели алгоритмов на языке Promela, а также результаты верификации алгоритмов при помощи верификатора Spin.

SUMMARY

In this paper, we consider Dekker's algorithm for two processes, as well as generalizations of this algorithm for n processes: Peterson's algorithm for n processes (filter algorithm) and Dijkstra's algorithm for n processes, designed to solve the problem of critical section. For the algorithms under consideration, software models were implemented in the Promela language, as well as verification of the correctness of the operation of these algorithms and verification of their properties, which was carried out using the Spin verification tool.

The result of this work is implemented software models of algorithms in the Promela language, as well as the results of verification of algorithms using the Spin verifier.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является изучение обобщенных алгоритмов Деккера для n процессов, а также их моделирование и верификация свойств.

Для достижения поставленной цели были сформулированы следующие задачи:

- Изучение алгоритма Деккера для 2 процессов;
- Изучение обобщений алгоритма Деккера для n процессов: алгоритма Питерсона для n процессов и алгоритма Дейкстры для n процессов;
- Моделирование алгоритма Деккера для 2 процессов, алгоритма Питерсона для n процессов и алгоритма Дейкстры для n процессов на языке Promela;
- Верификация свойств алгоритма Деккера для 2 процессов, алгоритма Питерсона для n процессов и алгоритма Дейкстры для n процессов при помощи инструмента верификации Spin.

1. ОПИСАНИЕ РАССМАТРИВАЕМЫХ АЛГОРИТМОВ

Алгоритмы, которые будут рассматриваться ниже, в общем и целом, предназначены для решения *проблемы критической секции*. Формулируется она следующим образом:

- Каждый из N процессов выполняют в бесконечном цикле последовательность команд, которую можно разделить на две части: критическая секция и некритическая секция;
- Требуется любое решение, удовлетворяющее следующим свойствам:
  - Взаимоисключающий доступ. В критической секции может находиться только один из процессов;
  - Свобода от дедлоков. Если несколько процессов пытаются войти в критическую секцию, то только у одного из них это получится;
  - Свобода от голодания. Если какой-либо из процессов пытается войти в критическую секцию, то когда-нибудь у него это обязательно получится;
- Механизм синхронизации должен быть представлен в виде дополнительных команд до и после критической секции. Команды, расположенные до критической секции, являются *препротоколом*, а команды, расположенные после – *постпротоколом*;
- Протоколы могут использовать локальные или глобальные переменные, при этом предполагается, что никакие переменные, используемые в критической секции, не используются в протоколах;
- Критическая секция должна быть прогрессивна. Если один из процессов заходит в критическую секцию, то он когда-нибудь из нее выйдет;
- Некритическая секция непрогрессивна. Если процесс войдет в некритическую секцию, то он может не покидать ее.

Далее мы как раз будем рассматривать препротоколы и постпротоколы алгоритмов, решающих проблему критической секции.

1.1. Алгоритм Деккера для 2 процессов

Алгоритм Деккера, решающий проблему критической секции для 2 процессов, выглядит следующим образом (псевдокод алгоритма был взят из источника 1, глава 3.9):

Algorithm 3.10: Dekker's algorithm

```
boolean wantp ← false, wantq ← false
integer turn ← 1
```

	p		q
	loop forever		loop forever
p1	non-critical section	q1	non-critical section
p2	wantp ← true	q2	wantq ← true
p3	while wantq	q3	while wantp
p4	if turn = 2	q4	if turn = 1
p5	wantp ← false	q5	wantq ← false
p6	await turn = 1	q6	await turn = 2
p7	wantp ← true	q7	wantq ← true
p8	critical section	q8	critical section
p9	turn ← 2	q9	turn ← 1
p10	wantp ← false	q10	wantq ← false

Атомарные регистры *wantp* (SWMR) и *wantq* (SWMR) говорят нам о том, что процесс хочет или не хочет войти в критическую секцию. Если они равны *true* – то процесс хочет войти в критическую секцию, если они равны *false* – то не хочет. Атомарный регистр *turn* (MWMR) определяет, какой процесс войдет в критическую секцию, если оба процесса хотят войти в туда.

Как видно из алгоритма, если только один процесс хочет войти в критическую секцию, то он беспрепятственно это сделает. В случае, если два процесса захотят войти в критическую секцию, они войдут в цикл, и в зависимости от *turn* один из процессов пропустит другой процесс в критическую секцию. В случае, процесс, который пропускает, устанавливает флагу *want* значение *false*, говоря о том, что он более не претендует на получение доступа к

критической секции, после чего входит в цикл ожидания смены *turn*, после которого флагу *want* будет установлено значение *true*, сигнализируя о том, что процесс пропустил ход и теперь снова претендует на доступ к КС. Другой же процесс, после выхода из критической секции входит в постпротокол, где он говорит о том, что более не претендует на доступ к КС, и передает ход другому процессу, изменяя *turn*.

Рассмотренный алгоритм корректен: он обеспечивает взаимоисключающий доступ к критической секции, а также он является свободным от дедлоков и голодания. Его корректность и свойства будут подтверждены далее при моделировании и верификации в пунктах 2.1 и 3.1 соответственно.

1.2. Алгоритм Питерсона для n процессов (алгоритм фильтра)

Одним из обобщений алгоритма Деккера для n процессов является алгоритм Питерсона для n процессов или алгоритм фильтра, решающий проблему критической секции для n процессов, который выглядит следующим образом (псевдокод алгоритма был взят из источника 2, глава 3.9):

```
Shared Variables :
Q     :  array [1..n]    of integer;        (initially 0)
turn  :  array [1..n-1] of integer;        (initially 1)
Local Variables : i, n, j : integer;
Note : The construct wait until <cond.> means that wait
until condition is true and can be replaced by the standard
construct repeat until <cond.>. Here, i contains the process
number and N is the total number of processes.
Protocol for P_i is -
for j := 1 to N-1 do
begin
   Q[i] := j;
   turn[j] := i;
   wait until (∀k ≠ i, Q[k] < j) ∨ (turn[j] ≠ i)
end;
< CRITICAL SECTION >
Q[i] := 0;
```

Весь алгоритм основан на конструкции фильтра. Всего у нас имеется n уровней, на самом первом уровне 0 могут находиться *n* процессов, а на каждом следующем – на один процесс меньше. Таким образом на последнем уровне n - 1 у нас останется один процесс, которому и будет предоставлен доступ к критической секции.

Массив атомарных регистров *Q* (SWMR) хранит уровни, на которые процессы претендуют. Индекс массива – это индекс процесса, а значение – это уровень процесса. Массив атомарных регистров *turn* (MWMR) определяет процессы, который в следствие конкурентной борьбы остаются на уровнях и не переходят на следующий. Индекс массива – это уровень, а значение – это индекс процесса, который “проиграл” на этом уровне.

В препротоколе мы видим цикл по уровням фильтра, процессы продвигаются от 0 до *n* - 1 уровня до критической секции. В данном алгоритме индекс *j* в цикле стоит понимать, как уровень, на который претендует процесс, но еще не находящийся на нем. Поэтому *j* изменяется от 1 до *n* - 1. В теле цикла для начала устанавливается уровень, на который претендует процесс, а также для уровня, на котором находится этот процесс, устанавливается turn как текущий процесс. В итоге на уровне будет оставаться всегда тот процесс, который начал претендовать на следующий уровень последним. Далее идет цикл ожидания, состоящий из двух условий. Если *turn* уровня равен текущему процессу, то он остается ждать до тех пор, пока turn не сменится, и если на претендуемый уровень и следующих за ним уровней есть какие-нибудь другие процессы, то процесс также остается ждать, пока это условие выполняется. Первое условие необходимо для того, чтобы пропускать на следующий уровень всегда на один процесс меньше, а второе – для случая, когда на следующие уровни еще нет претендентов, чтобы процесс не ожидал других процессов, пока они возьмут *turn* на себя.

Рассмотренный алгоритм корректен: он обеспечивает взаимоисключающий доступ к критической секции, а также он является свободным от дедлоков и голодания. Его корректность и свойства будут

подтверждены далее при моделировании и верификации в пунктах 2.2 и 3.2 соответственно.

1.3. Алгоритм Дейкстры для n процессов

Еще одним из обобщений алгоритма Деккера для n процессов является алгоритм Дейкстры для n процессов, решающий проблему критической секции для n процессов. Он выглядит следующим образом (псевдокод алгоритма был взят из источника 2, глава 3.5):

```
Shared Variables :

Boolean array b, c [1: N];   (initialized to true)

integer k;

Note : 1 ≤ k ≤ N. b[i] and c[i] are set by P_i only, where as all
other processes can only read them. Here, i contains the process
number, and N is the total number of processes.

Local Variables :  integer j;

Protocol for Process P_i (1 ≤ i ≤ N) is -

Li0:   b[i] := false;
Li1:   if k ≠ i then
Li2:   begin  c[i] := true;
Li3:          if b[k] then k := i;
              goto Li1;
       end
Li4:   else begin
              c[i] := false;
              for j := 1 step 1 until N do
                if (j ≠ i) and (not c[j]) then goto Li1;
       end;
       < CRITICAL SECTION >
       c[i] := true;
       b[i] := true;
       < NON-CRITICAL SECTION >
       goto Li0;
```

Массив атомарных регистров *c* (SWMR) определяют возможность доступа процесса к критической секции. Алгоритм позволяет процессу получить доступ к критической секции, если его собственный флаг *c* равен *false*, а флаг *c* всех других процессов равен *true*. Массив атомарных регистров *b* (SWMR) определяют местонахождение процессов: если флаг b процесса равен false, то он находится в препротоколе, в критической секции или в постпротоколе, если *true* – то он находится в некритической секции. Атомарный регистр *k* (MWMR) определяет процесс, которому предоставлено право попасть в критическую секцию после проверки условия с флагами *c*.

Препротокол состоит из двух частей – первая для процессов, которым право на доступ к критической секции не предоставлено (они определяются регистром *k*), а вторая – для процесса, у которого это право есть.

В первом части процессы проверяют, не вышел ли процесс, соответствующий регистру *k*, в некритическую секцию (при помощи флага *b* для процесса *k*). Если он вошел в некритическую секцию, то это значит, что теперь право доступа к КС должно перейти к другому процессу. В этом случае это право забирает случайный процесс, который последним успел изменить значение *k*. Если же процесс k все еще находится в препротоколе, КС или постпротоколе – то процессы ждут изменений.

Во второй части для процесса, имеющего право на вход в КС, проверяются условие флагов *c* для процессов. Если оно выполняется, то процесс может спокойно входить в КС, если же нет – это значит, что существуют некоторые другие процессы, которые находятся во второй части и еще не перешли в первую часть.

Рассмотренный алгоритм корректен: он обеспечивает взаимоисключающий доступ к критической секции, а также он является свободным от дедлоков и голодания. Его корректность и свойства будут подтверждены далее при моделировании и верификации в пунктах 2.3 и 3.3 соответственно.

2. МОДЕЛИРОВАНИЕ АЛГОРИТМОВ

В этой части работы для рассмотренных выше алгоритмов будет реализована программная модель на языке Promela с реализацией проблемы критической секции.

2.1. Алгоритм Деккера для 2 процессов

По псевдокоду алгоритма Деккера для 2 процессов из пункта 1.1 была реализована следующая модель:

Файл 2-processes-dekker.pml

```
//  ====  LTL  ====  //

int critical = 0;
bool inCS[2] = {false, false};

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { []<> NO_STARVATION(0) };

//  ====  Algorithm  ====  //

int turn = 0;
bool want[2] = {false, false};

inline acquire(i)
{
    want[i] = true;

    do
    :: (want[1 - i] == false) -> break;
    :: (want[1 - i] == true && turn == i) -> skip;
    :: else ->
       want[i] = false;
       (turn == i);
       want[i] = true;
    od;
}

inline release(i)
{
    turn = 1 - i;
    want[i] = false;
}

//  ====  Processes  ====  //

active [2] proctype P()
{
do
:: acquire(_pid);
```

```
   critical++;
   inCS[_pid] = true;
   printf("Process #%d enter CS.\n", _pid);
   printf("Process #%d leave CS.\n", _pid);
   inCS[_pid] = false;
   critical--;
   release(_pid);
od;
}
```

Алгоритм представлен в виде двух процедур препротокола (acquire) и постпротокола (release), объяснение работы которых приведено в пункте 1.1, с общими регистрами *want* и *turn*. Переменная *want* является массивом регистров, его первая ячейка аналогична *wantp*, а вторая - *wantq*. Они принимают в качестве аргумента *i* – индекс процесса, который исполняет данную процедуру.

```
int turn = 0;
bool want[2] = {false, false};

inline acquire(i)
{
    want[i] = true;

    do
    :: (want[1 - i] == false) -> break;
    :: (want[1 - i] == true && turn == i) -> skip;
    :: else ->
       want[i] = false;
       (turn == i);
       want[i] = true;
    od;
}

inline release(i)
{
    turn = 1 - i;
    want[i] = false;
}
```

Для верификации алгоритма также были объявлены ltl операторы, хранящие LTL-выражения для проверки. Здесь регистр *critical* – это количество процессов, которые в данным момент находятся в критической секции, а массив регистров *inCS* каждому процессу сопоставляет флаг, который говорит нам о том, находится ли процесс на данным момент в КС. Подробнее о них будет рассказано в пункте 3.1.

```
int critical = 0;
bool inCS[2] = {false, false};

#define MUTEX (critical <= 1)
```

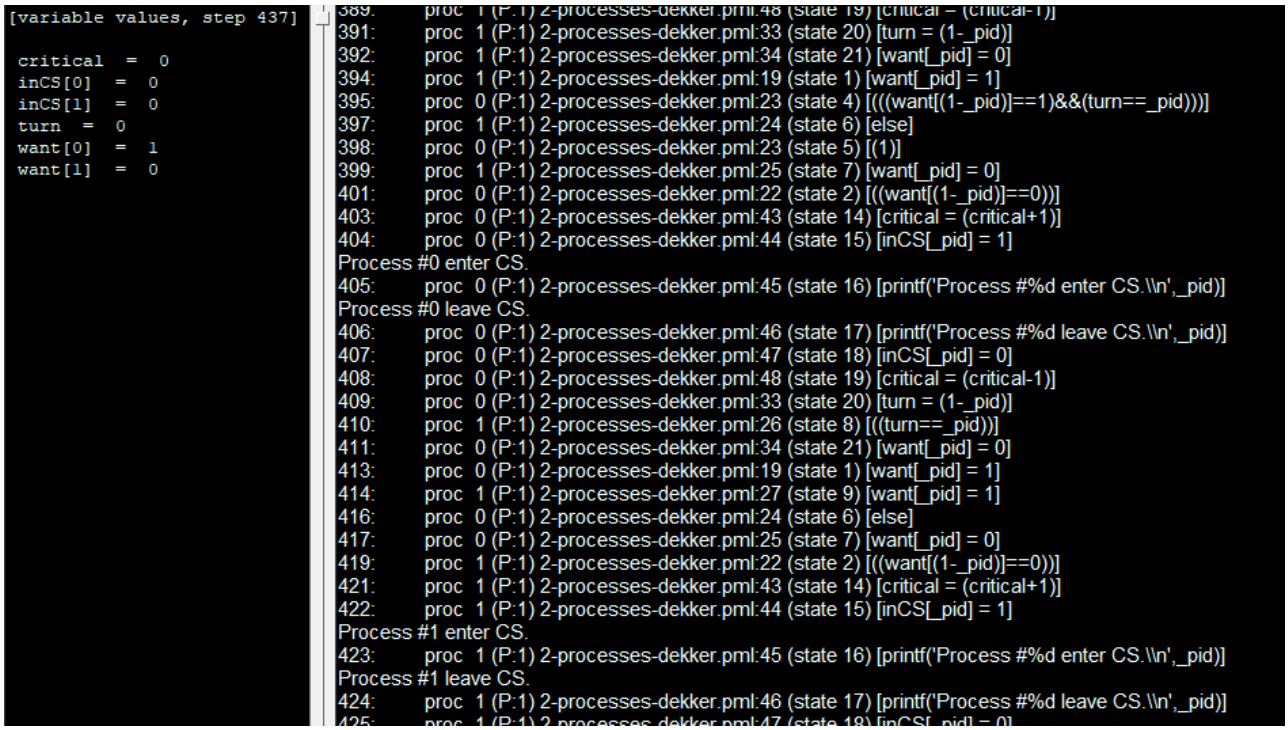
```
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { []<> NO_STARVATION(0) };
```

Сам алгоритм проверяется на решении задачи критической секции, которая реализована следующим образом:

```
active [2] proctype P()
{
do
:: acquire(_pid);
   critical++;
   inCS[_pid] = true;
   printf("Process #%d enter CS.\n", _pid);
   printf("Process #%d leave CS.\n", _pid);
   inCS[_pid] = false;
   critical--;
   release(_pid);
od;
}
```

Внутри критической секции (между acquire и release) происходит обновление регистров для LTL-выражений, а также вывод сообщений о посещении критической секции процессом.

Алгоритм был запущен на случайном сценарии для проверки работоспособности:



2.2. Алгоритм Питерсона для n процессов (алгоритм фильтра)

По псевдокоду алгоритма фильтра из пункта 1.2 была реализована следующая модель:

Файл n-processes-peterson-filter.pml

```
#define PROC_NUM 4

//  ====  LTL  ====  //

int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };

//  ====  Algorithm  ====  //

int level[PROC_NUM];
int turn[PROC_NUM];

inline acquire(i)
{
    int l = 1;
    do
    :: (l >= PROC_NUM) -> break;
    :: else ->
       level[i] = l;
       turn[l] = i;

       do
       :: (turn[l] != i) -> break;
       :: else ->
          int k = 0;
          do
          :: (k >= PROC_NUM) -> goto NL;
          :: (k < PROC_NUM && k != i && level[k] >= l) -> break;
          :: else -> k++;
          od;
       od;

NL:  l++;
    od;
}

inline release(i)
{
    level[i] = 0;
}

//  ====  Processes  ====  //

proctype P(int i)
{
    do
    :: acquire(i);
       critical++;
```

```
        inCS[i] = true;
        printf("Process %d enter CS.\n", i);
        printf("Process %d leave CS.\n", i);
        inCS[i] = false;
        critical--;
        release(i);
    od;
}

init
{
    atomic
    {
        int i = 0;
        do
        :: (i >= PROC_NUM) -> break;
        :: else ->
           level[i] = 0;
           turn[i] = 0;
           inCS[i] = false;
           run P(i);
           i++;
        od;
    }
}
```

Количество процессов было выбрано равным 4. Алгоритм представлен в виде двух процедур препротокола (acquire) и постпротокола (release), объяснение работы которых приведено в пункте 1.2, с общими массивами регистров *level* и *turn*. Они принимают в качестве аргумента *i* – индекс процесса, который исполняет данную процедуру.

```
int level[PROC_NUM];
int turn[PROC_NUM];

inline acquire(i)
{
    int l = 1;
    do
    :: (l >= PROC_NUM) -> break;
    :: else ->
       level[i] = l;
       turn[l] = i;

       do
       :: (turn[l] != i) -> break;
       :: else ->
          int k = 0;
          do
          :: (k >= PROC_NUM) -> goto NL;
          :: (k < PROC_NUM && k != i && level[k] >= l) -> break;
          :: else -> k++;
          od;
       od;

NL:  l++;
    od;
```

```
}

inline release(i)
{
    level[i] = 0;
}
```

Для верификации алгоритма также были объявлены ltl операторы, хранящие LTL-выражения для проверки. Здесь регистр *critical* – это количество процессов, которые в данным момент находятся в критической секции, а массив регистров *inCS* каждому процессу сопоставляет флаг, который говорит нам о том, находится ли процесс на данным момент в КС. Подробнее о них будет рассказано в пункте 3.2.

```
int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };
```

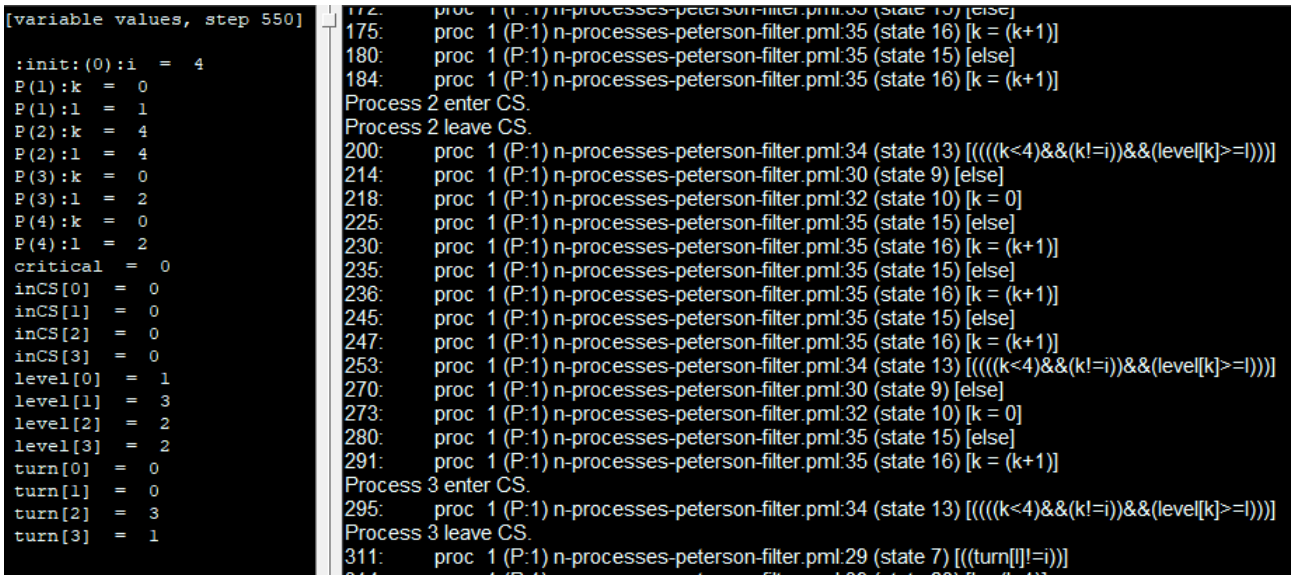
Сам алгоритм проверяется на решении задачи критической секции, которая реализована следующим образом:

```
proctype P(int i)
{
    do
    :: acquire(i);
       critical++;
       inCS[i] = true;
       printf("Process %d enter CS.\n", i);
       printf("Process %d leave CS.\n", i);
       inCS[i] = false;
       critical--;
       release(i);
    od;
}

init
{
    atomic
    {
        int i = 0;
        do
        :: (i >= PROC_NUM) -> break;
        :: else ->
           level[i] = 0;
           turn[i] = 0;
           inCS[i] = false;
           run P(i);
           i++;
```

```
        od;
    }
}
```

Внутри критической секции (между acquire и release) происходит обновление регистров для LTL-выражений, а также вывод сообщений о посещении критической секции процессом. При помощи init для общих регистров задаются начальные значения до начала работы основных процессов.

Алгоритм был запущен на случайном сценарии для проверки работоспособности:



2.3. Алгоритм Дейкстры для n процессов

По псевдокоду алгоритма Дейкстры для n процессов из пункта 1.3 была реализована следующая модель:

Файл n-processes-deijkstra.pml

```
#define PROC_NUM 4

//  ====  LTL  ====  //

int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };

//  ====  Algorithm  ====  //
```

```
bool b[PROC_NUM];
bool c[PROC_NUM];
int k = 0;

inline acquire(i)
{
    b[i] = true;

L0: do
    :: (k == i) -> break;
    :: else ->
       c[i] = true;
       if
       :: (b[k]) -> k = i;
       :: else -> skip;
       fi;
    od;

    c[i] = false;
    int j = 0;
    do
    :: (j >= PROC_NUM) -> break;
    :: (j < PROC_NUM && i != j && !c[j]) -> goto L0;
    :: else -> j++;
    od;
}

inline release(i)
{
    c[i] = true;
    b[i] = true;
}

//  ====  Processes  ====  //

proctype P(int i)
{
do
:: acquire(i);
   critical++;
   inCS[i] = true;
   printf("Process #%d enter CS.\n", i);
   printf("Process #%d leave CS.\n", i);
   inCS[i] = false;
   critical--;
   release(i);
od;
}

init
{
    atomic
    {
        int i = 0;
        do
        :: (i >= PROC_NUM) -> break;
        :: else ->
           b[i] = true;
           c[i] = true;
           inCS[i] = false;
           run P(i);
           i++;
```

```
        od;
    }
}
```

Количество процессов было выбрано равным 4. Алгоритм представлен в виде двух процедур препротокола (acquire) и постпротокола (release), объяснение работы которых приведено в пункте 1.2, с общими массивами регистров *b* и *c*, а также регистром *k*. Они принимают в качестве аргумента *i* – индекс процесса, который исполняет данную процедуру.

```
bool b[PROC_NUM];
bool c[PROC_NUM];
int k = 0;

inline acquire(i)
{
    b[i] = true;

L0: do
    :: (k == i) -> break;
    :: else ->
       c[i] = true;
       if
       :: (b[k]) -> k = i;
       :: else -> skip;
       fi;
    od;

    c[i] = false;
    int j = 0;
    do
    :: (j >= PROC_NUM) -> break;
    :: (j < PROC_NUM && i != j && !c[j]) -> goto L0;
    :: else -> j++;
    od;
}

inline release(i)
{
    c[i] = true;
    b[i] = true;
}
```

Для верификации алгоритма также были объявлены ltl операторы, хранящие LTL-выражения для проверки. Здесь регистр *critical* – это количество процессов, которые в данным момент находятся в критической секции, а массив регистров *inCS* каждому процессу сопоставляет флаг, который говорит нам о том, находится ли процесс на данным момент в КС. Подробнее о них будет рассказано в пункте 3.2.

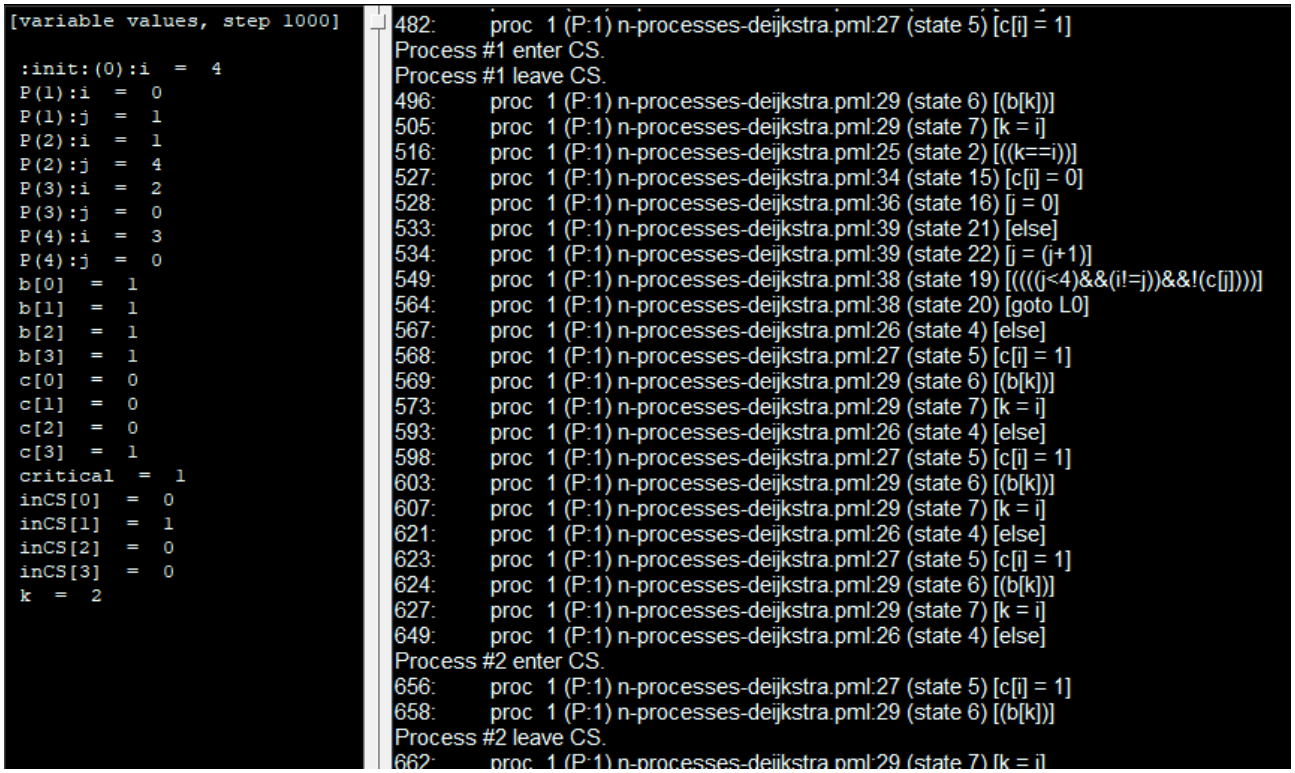
```
int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };
```

Сам алгоритм проверяется на решении задачи критической секции, которая реализована следующим образом:

```
proctype P(int i)
{
do
:: acquire(i);
   critical++;
   inCS[i] = true;
   printf("Process #%d enter CS.\n", i);
   printf("Process #%d leave CS.\n", i);
   inCS[i] = false;
   critical--;
   release(i);
od;
}

init
{
    atomic
    {
        int i = 0;
        do
        :: (i >= PROC_NUM) -> break;
        :: else ->
           b[i] = true;
           c[i] = true;
           inCS[i] = false;
           run P(i);
           i++;
        od;
    }
}
```

Внутри критической секции (между acquire и release) происходит обновление регистров для LTL-выражений, а также вывод сообщений о посещении критической секции процессом. При помощи init для общих регистров задаются начальные значения до начала работы основных процессов.

Алгоритм был запущен на случайном сценарии для проверки работоспособности:



3. ВЕРИФИКАЦИЯ АЛГОРИТМОВ

В этой части работы для реализованных выше программных моделей была проведена верификация свойств алгоритмов, а именно:

- Свойства взаимоисключающего доступа;
- Свойства свободы от deadlock-ов;
- Свойства свободы от голодания;

3.1. Алгоритм Деккера для 2 процессов

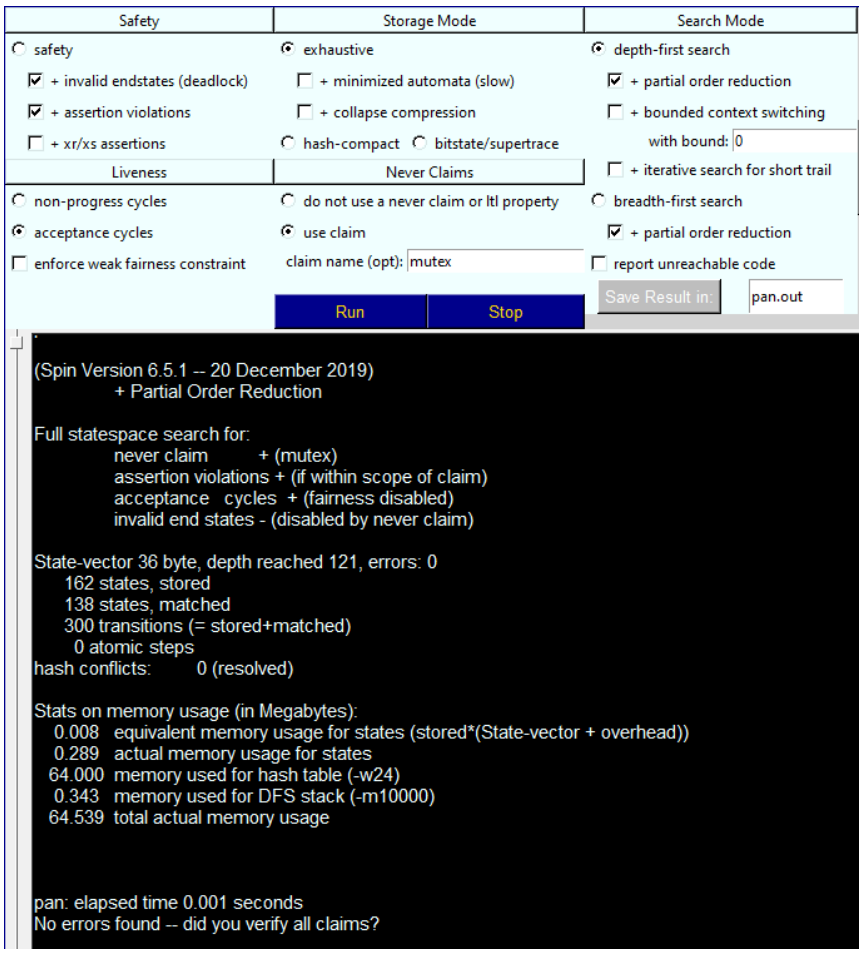
Для верификации алгоритма были объявлены следующие LTL-выражения:

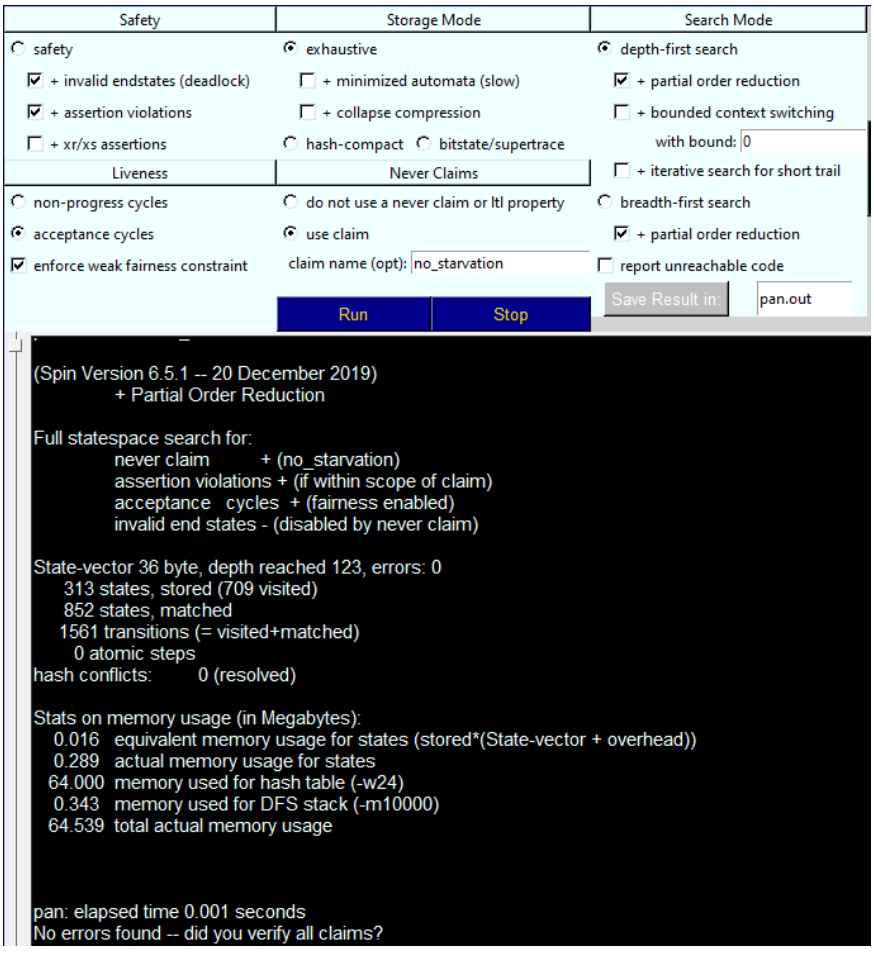
- LTL-выражение *mutex* отражает свойство взаимоисключающего доступа: всегда в будущем количество процессов в критической секции меньше или равно единице ([] *critical* <= 1);
- LTL-выражение *no_starvation* отражает свойство свободы от голодания: всегда в будущем процесс i когда-нибудь войдет в критическую секцию ([]<> inCS[i] == true). Поскольку алгоритм Деккера симметричный, достаточно проверить свободу от голодания для одного из процессов. Также для верификации данного свойства в Spin требуется отключить проверку для несправедливых вычислений.

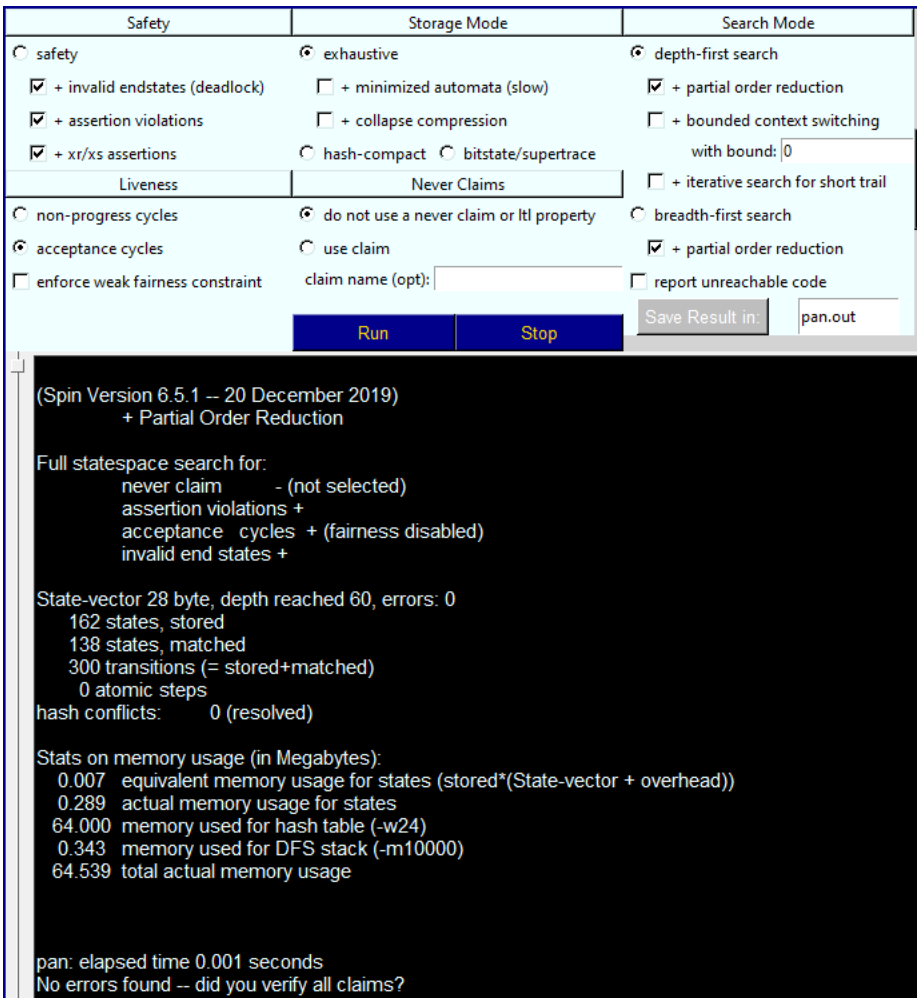
```
int critical = 0;
bool inCS[2] = {false, false};

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { []<> NO_STARVATION(0) };
```

Свойство свободы от deadlock-ов было проверено при помощи готовых LTL для Spin.

Все рассматриваемые свойства выполняются для рассматриваемого алгоритма. Результаты верификации свойств:


Safety
Storage Mode
Search Mode
safety
exhaustive
depth-first search
+ invalid endstates (deadlock)
+ minimized automata (slow)
+ partial order reduction
+ assertion violations
+ collapse compression
+ bounded context switching
+ xr/xs assertions
hash-compact
bitstate/supertrace
with bound: 0
Liveness
Never Claims
+ iterative search for short trail
non-progress cycles
do not use a never claim or ltl property
breadth-first search
acceptance cycles
use claim
+ partial order reduction
enforce weak fairness constraint
claim name (opt): mutex
report unreachable code
Run
Stop
Save Result in:
pan.out
(Spin Version 6.5.1 -- 20 December 2019)
+ Partial Order Reduction
Full statespace search for:
never claim + (mutex)
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)
State-vector 36 byte, depth reached 121, errors: 0
162 states, stored
138 states, matched
300 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)
Stats on memory usage (in Megabytes):
0.008 equivalent memory usage for states (stored*(State-vector + overhead))
0.289 actual memory usage for states
64.000 memory used for hash table (-w24)
0.343 memory used for DFS stack (-m10000)
64.539 total actual memory usage
pan: elapsed time 0.001 seconds
No errors found -- did you verify all claims?


Safety
Storage Mode
Search Mode
safety
exhaustive
depth-first search
+ invalid endstates (deadlock)
+ minimized automata (slow)
+ partial order reduction
+ assertion violations
+ collapse compression
+ bounded context switching
+ xr/xs assertions
hash-compact
bitstate/supertrace
with bound: 0
Liveness
Never Claims
+ iterative search for short trail
non-progress cycles
do not use a never claim or ltl property
breadth-first search
acceptance cycles
use claim
+ partial order reduction
enforce weak fairness constraint
claim name (opt): no_starvation
report unreachable code
Run
Stop
Save Result in:
pan.out
(Spin Version 6.5.1 -- 20 December 2019)
+ Partial Order Reduction
Full statespace search for:
never claim + (no_starvation)
assertion violations + (if within scope of claim)
acceptance cycles + (fairness enabled)
invalid end states - (disabled by never claim)
State-vector 36 byte, depth reached 123, errors: 0
313 states, stored (709 visited)
852 states, matched
1561 transitions (= visited+matched)
0 atomic steps
hash conflicts: 0 (resolved)
Stats on memory usage (in Megabytes):
0.016 equivalent memory usage for states (stored*(State-vector + overhead))
0.289 actual memory usage for states
64.000 memory used for hash table (-w24)
0.343 memory used for DFS stack (-m10000)
64.539 total actual memory usage
pan: elapsed time 0.001 seconds
No errors found -- did you verify all claims?



3.2. Алгоритм Питерсона для n процессов (алгоритм фильтра)

Для верификации алгоритма были объявлены следующие LTL-выражения:

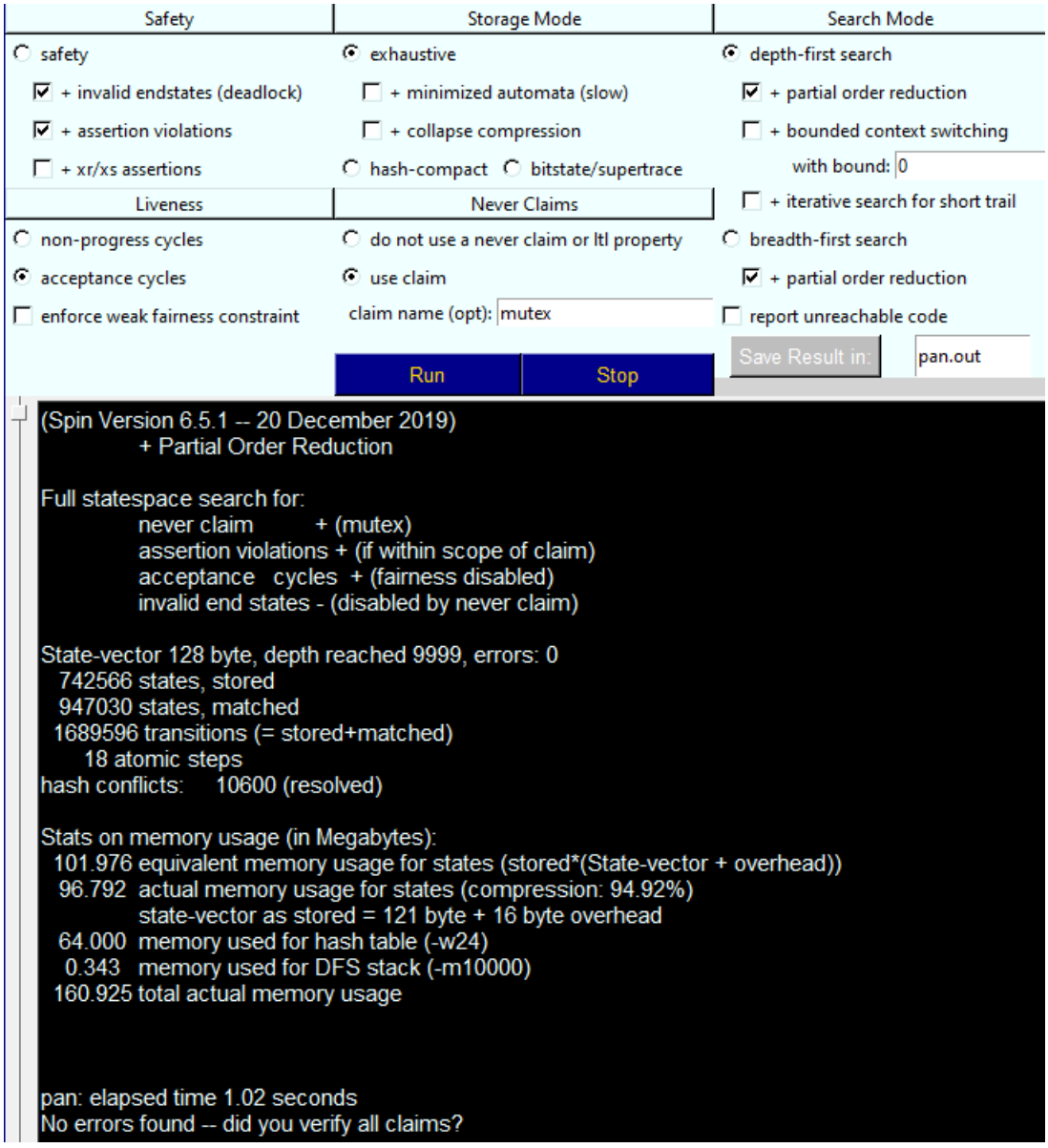
- LTL-выражение *mutex* отражает свойство взаимоисключающего доступа: всегда в будущем количество процессов в критической секции меньше или равно единице ([] *critical* <= 1);
- LTL-выражение *no_starvation* отражает свойство свободы от голодания: всегда в будущем процесс i когда-нибудь войдет в критическую секцию ([]<> inCS[i] == true). Также для верификации данного свойства в Spin требуется отключить проверку для несправедливых вычислений.

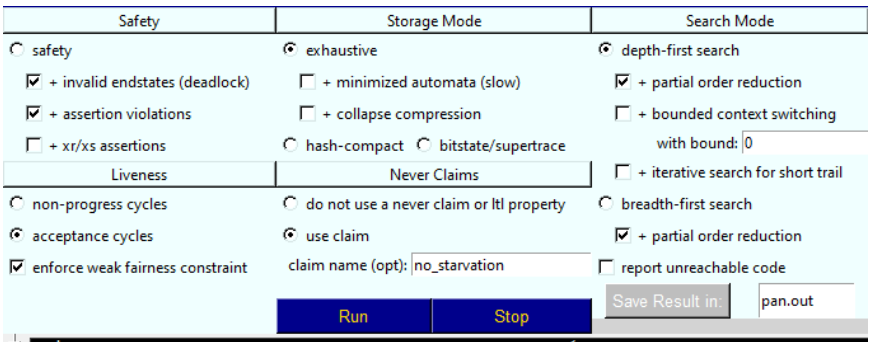
```
int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
```

```
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };
```

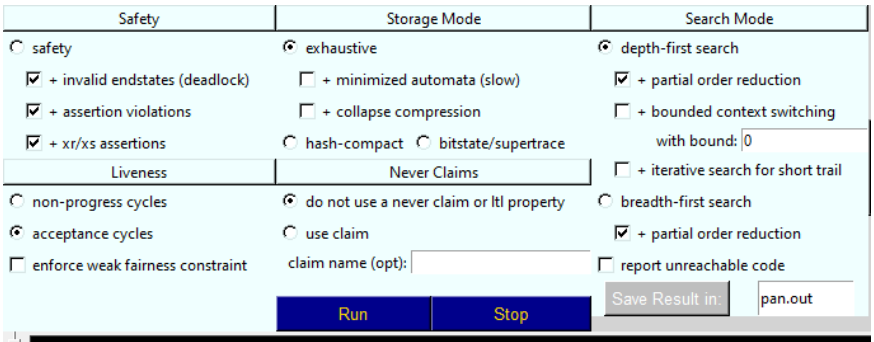
Свойство свободы от deadlock-ов было проверено при помощи готовых LTL для Spin.

Все рассматриваемые свойства выполняются для рассматриваемого алгоритма. Результаты верификации свойств:






```
(Spin Version 6.5.1 -- 20 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim         + (no_starvation)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness enabled)
        invalid end states - (disabled by never claim)

State-vector 128 byte, depth reached 9999, errors: 0
  4941677 states, stored (3.79148e+007 visited)
 68016415 states, matched
1.0593118e+008 transitions (= visited+matched)
      90 atomic steps
hash conflicts:   5132737 (resolved)

Stats on memory usage (in Megabytes):
  678.636 equivalent memory usage for states (stored*(State-vector + overhead))
  690.766 actual memory usage for states
  256.000 memory used for hash table (-w26)
    0.343   memory used for DFS stack (-m10000)
  946.222 total actual memory usage



pan: elapsed time 72.1 seconds
No errors found -- did you verify all claims?
```




```
(Spin Version 6.5.1 -- 20 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim         - (not selected)
        assertion violations +
        acceptance   cycles  + (fairness disabled)
        invalid end states +

State-vector 120 byte, depth reached 364652, errors: 0
  1389387 states, stored
  2160551 states, matched
  3549938 transitions (= stored+matched)
       18 atomic steps
hash conflicts:     57214 (resolved)

Stats on memory usage (in Megabytes):
  180.203 equivalent memory usage for states (stored*(State-vector + overhead))
  169.816 actual memory usage for states (compression: 94.24%)
          state-vector as stored = 112 byte + 16 byte overhead
   64.000  memory used for hash table (-w24)
   34.332  memory used for DFS stack (-m1000000)
  268.059 total actual memory usage



pan: elapsed time 2.18 seconds
No errors found -- did you verify all claims?
```

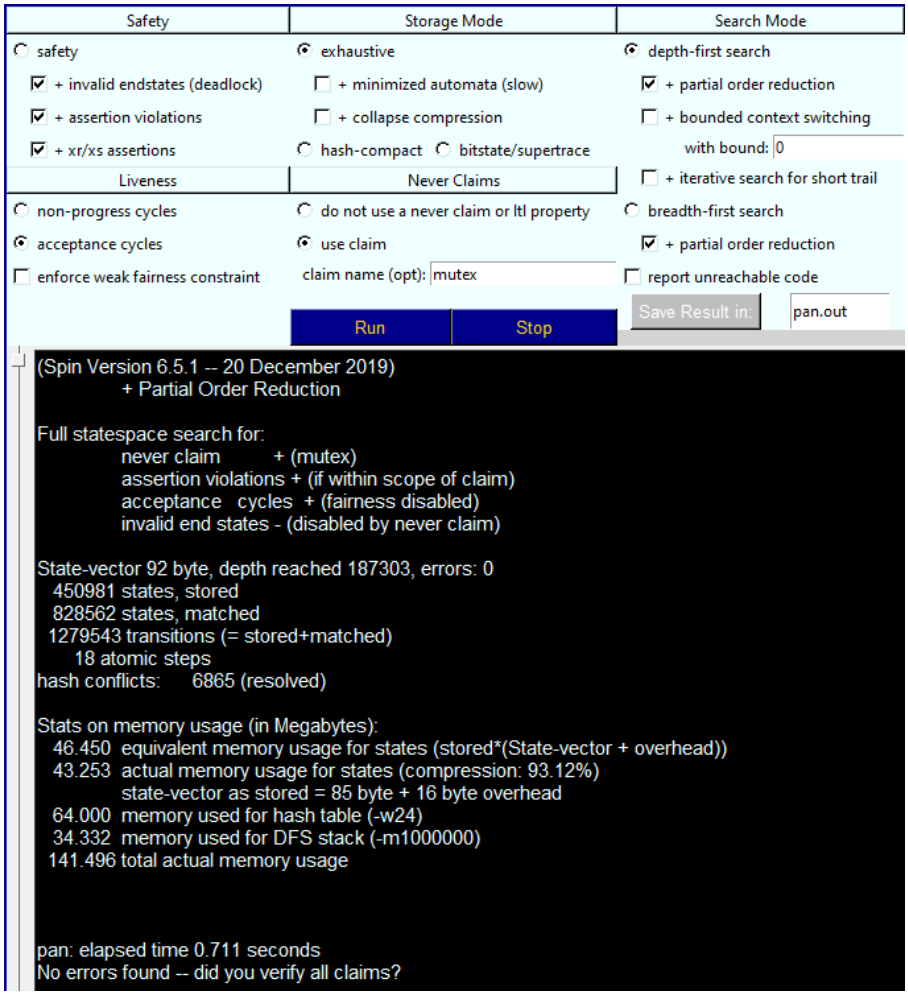
3.3. Алгоритм Дейкстры для n процессов

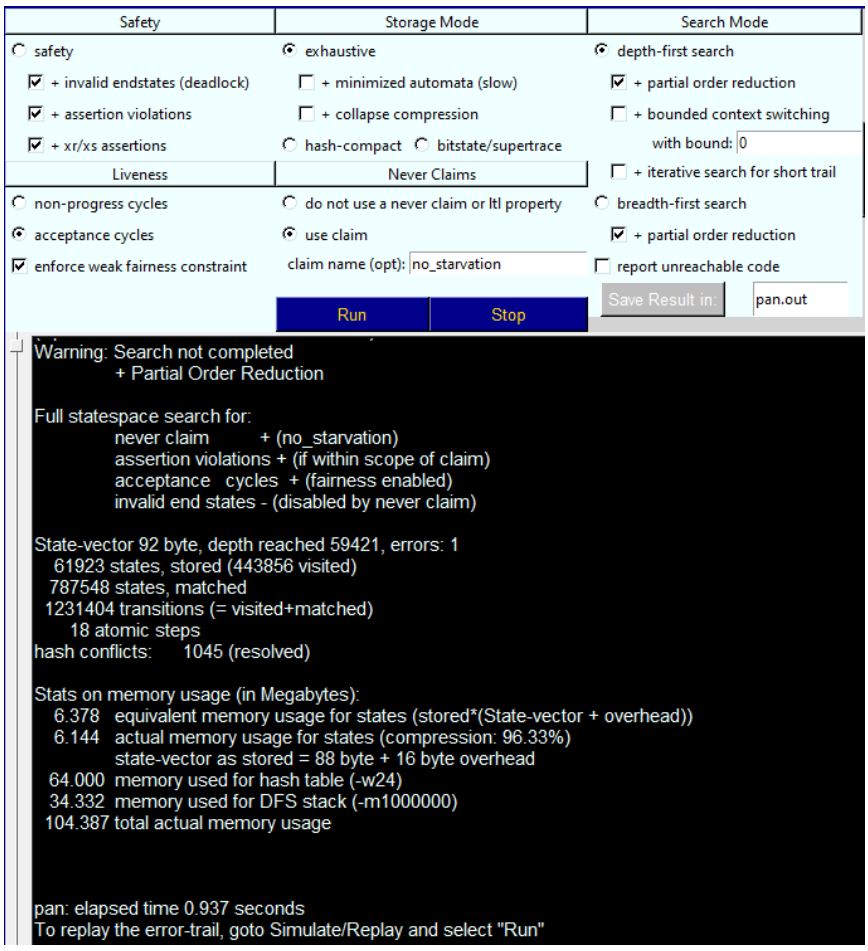
Для верификации алгоритма были объявлены LTL-выражения, аналогичные LTL-выражениям из пункта 3.2.

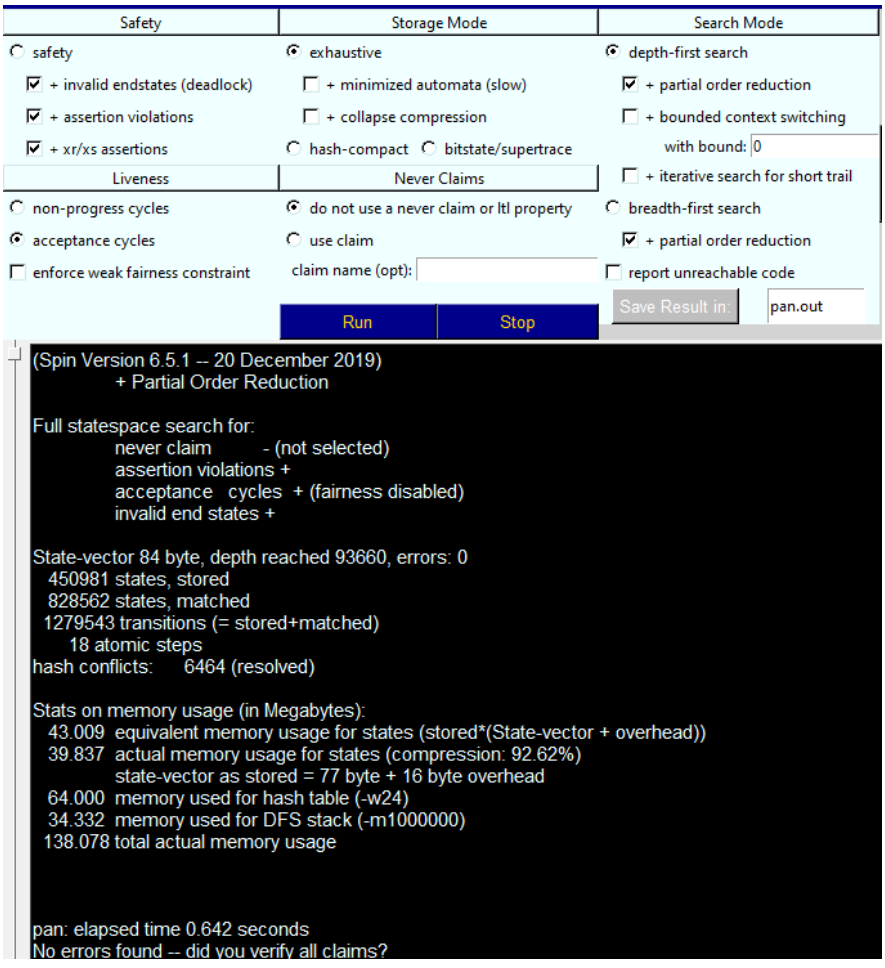
```
int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };
```

Свойство свободы от deadlock-ов было проверено при помощи готовых LTL для Spin.

Для алгоритма Дейкстры выполняются только свойства взаимоисключающего доступа и свободы от deadlock-ов. Свойства свободы от голодания не выполняется. Результаты верификации свойств:




Safety
Storage Mode
Search Mode
safety
exhaustive
depth-first search
+ invalid endstates (deadlock)
+ minimized automata (slow)
+ partial order reduction
+ assertion violations
+ collapse compression
+ bounded context switching
+ xr/xs assertions
hash-compact
bitstate/supertrace
with bound: 0
Liveness
Never Claims
+ iterative search for short trail
non-progress cycles
do not use a never claim or ltl property
breadth-first search
acceptance cycles
use claim
+ partial order reduction
enforce weak fairness constraint
claim name (opt): no_starvation
report unreachable code
Run
Stop
Save Result in:
pan.out
Warning: Search not completed
+ Partial Order Reduction
Full statespace search for:
never claim + (no_starvation)
assertion violations + (if within scope of claim)
acceptance cycles + (fairness enabled)
invalid end states - (disabled by never claim)
State-vector 92 byte, depth reached 59421, errors: 1
61923 states, stored (443856 visited)
787548 states, matched
1231404 transitions (= visited+matched)
18 atomic steps
hash conflicts: 1045 (resolved)
Stats on memory usage (in Megabytes):
6.378 equivalent memory usage for states (stored*(State-vector + overhead))
6.144 actual memory usage for states (compression: 96.33%)
state-vector as stored = 88 byte + 16 byte overhead
64.000 memory used for hash table (-w24)
34.332 memory used for DFS stack (-m1000000)
104.387 total actual memory usage
pan: elapsed time 0.937 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"


Safety
Storage Mode
Search Mode
safety
exhaustive
depth-first search
+ invalid endstates (deadlock)
+ minimized automata (slow)
+ partial order reduction
+ assertion violations
+ collapse compression
+ bounded context switching
+ xr/xs assertions
hash-compact
bitstate/supertrace
with bound: 0
Liveness
Never Claims
+ iterative search for short trail
non-progress cycles
do not use a never claim or ltl property
breadth-first search
acceptance cycles
use claim
+ partial order reduction
enforce weak fairness constraint
claim name (opt):
report unreachable code
Run
Stop
Save Result in:
pan.out
(Spin Version 6.5.1 -- 20 December 2019)
+ Partial Order Reduction
Full statespace search for:
never claim - (not selected)
assertion violations +
acceptance cycles + (fairness disabled)
invalid end states +
State-vector 84 byte, depth reached 93660, errors: 0
450981 states, stored
828562 states, matched
1279543 transitions (= stored+matched)
18 atomic steps
hash conflicts: 6464 (resolved)
Stats on memory usage (in Megabytes):
43.009 equivalent memory usage for states (stored*(State-vector + overhead))
39.837 actual memory usage for states (compression: 92.62%)
state-vector as stored = 77 byte + 16 byte overhead
64.000 memory used for hash table (-w24)
34.332 memory used for DFS stack (-m1000000)
138.078 total actual memory usage
pan: elapsed time 0.642 seconds
No errors found -- did you verify all claims?

ЗАКЛЮЧЕНИЕ

В ходе работы были рассмотрены алгоритм Деккера для 2 процессов и обобщенные алгоритмы Деккера для 2 процессов: алгоритм фильтра и алгоритм Дейкстры для n процессов. Для рассмотренных алгоритмов были реализованы программные модели на языке Promela, а также верифицированы свойства взаимоисключающего доступа, свободы от deadlock-ов и свободы от голодания.

Были получены следующие результаты:

- Для алгоритма Деккера для 2 процессов выполняются все свойства;
- Для алгоритма Питерсона для n процессов выполняются все свойства;
- Для алгоритма Дейкстры для n процессов выполняются все свойства, кроме свойства свободы от голодания.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. M. Ben-Ari. Principles of Concurrent and Distributed Programming. Second Edition. Addison-Wesley, 2006. 360 с.

2. Rajeev Chawla. The Problem of Mutual Exclusion: A New Distributed Solution. Virginia Commonwealth University, 1991. 181 с.

3. Basic Spin Manual // Spin. URL: https://spinroot.com/spin/Man/Manual.html (дата обращения: 22.05.2022).

ПРИЛОЖЕНИЕ А
ИСХОДНЫЕ КОДЫ АЛГОРИТМОВ НА ЯЗЫКЕ PROMELA

Алгоритм Деккера для 2 процессов:

Файл 2-processes-dekker.pml

```
//  ====  LTL  ====  //

int critical = 0;
bool inCS[2] = {false, false};

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { []<> NO_STARVATION(0) };

//  ====  Algorithm  ====  //

int turn = 0;
bool want[2] = {false, false};

inline acquire(i)
{
    want[i] = true;

    do
    :: (want[1 - i] == false) -> break;
    :: (want[1 - i] == true && turn == i) -> skip;
    :: else ->
       want[i] = false;
       (turn == i);
       want[i] = true;
    od;
}

inline release(i)
{
    turn = 1 - i;
    want[i] = false;
}

//  ====  Processes  ====  //

active [2] proctype P()
{
do
:: acquire(_pid);
   critical++;
   inCS[_pid] = true;
   printf("Process #%d enter CS.\n", _pid);
   printf("Process #%d leave CS.\n", _pid);
   inCS[_pid] = false;
   critical--;
   release(_pid);
od;
}
```

Алгоритм Питерсона для n процессов (алгоритм фильтра):

Файл n-processes-peterson-filter.pml

```
#define PROC_NUM 4

//  ====  LTL  ====  //

int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };

//  ====  Algorithm  ====  //

int level[PROC_NUM];
int turn[PROC_NUM];

inline acquire(i)
{
    int l = 1;
    do
    :: (l >= PROC_NUM) -> break;
    :: else ->
       level[i] = l;
       turn[l] = i;

       do
       :: (turn[l] != i) -> break;
       :: else ->
          int k = 0;
          do
          :: (k >= PROC_NUM) -> goto NL;
          :: (k < PROC_NUM && k != i && level[k] >= l) -> break;
          :: else -> k++;
          od;
       od;

NL:  l++;
    od;
}

inline release(i)
{
    level[i] = 0;
}

//  ====  Processes  ====  //

proctype P(int i)
{
    do
    :: acquire(i);
       critical++;
       inCS[i] = true;
       printf("Process %d enter CS.\n", i);
       printf("Process %d leave CS.\n", i);
       inCS[i] = false;
```

```
       critical--;
       release(i);
    od;
}

init
{
    atomic
    {
        int i = 0;
        do
        :: (i >= PROC_NUM) -> break;
        :: else ->
           level[i] = 0;
           turn[i] = 0;
           inCS[i] = false;
           run P(i);
           i++;
        od;
    }
}
```

Алгоритм Дейкстры для n процессов:

Файл n-processes-deijkstra.pml

```
#define PROC_NUM 4

//  ====  LTL  ====  //

int critical = 0;
bool inCS[PROC_NUM];

#define MUTEX (critical <= 1)
#define NO_STARVATION(i) (inCS[i] == true)

ltl mutex { [] MUTEX };
ltl no_starvation { ([]<> NO_STARVATION(0)) && ([]<> NO_STARVATION(1)) && ([]<>
NO_STARVATION(2)) && ([]<> NO_STARVATION(3)) };

//  ====  Algorithm  ====  //

bool b[PROC_NUM];
bool c[PROC_NUM];
int k = 0;

inline acquire(i)
{
    b[i] = true;

L0: do
    :: (k == i) -> break;
    :: else ->
       c[i] = true;
       if
       :: (b[k]) -> k = i;
       :: else -> skip;
       fi;
    od;

    c[i] = false;
    int j = 0;
```

```
    do
    :: (j >= PROC_NUM) -> break;
    :: (j < PROC_NUM && i != j && !c[j]) -> goto L0;
    :: else -> j++;
    od;
}

inline release(i)
{
    c[i] = true;
    b[i] = true;
}

//  ====  Processes  ====  //

proctype P(int i)
{
do
:: acquire(i);
   critical++;
   inCS[i] = true;
   printf("Process #%d enter CS.\n", i);
   printf("Process #%d leave CS.\n", i);
   inCS[i] = false;
   critical--;
   release(i);
od;
}

init
{
    atomic
    {
        int i = 0;
        do
        :: (i >= PROC_NUM) -> break;
        :: else ->
           b[i] = true;
           c[i] = true;
           inCS[i] = false;
           run P(i);
           i++;
        od;
    }
}
```