Язык Promela инструмента SPIN

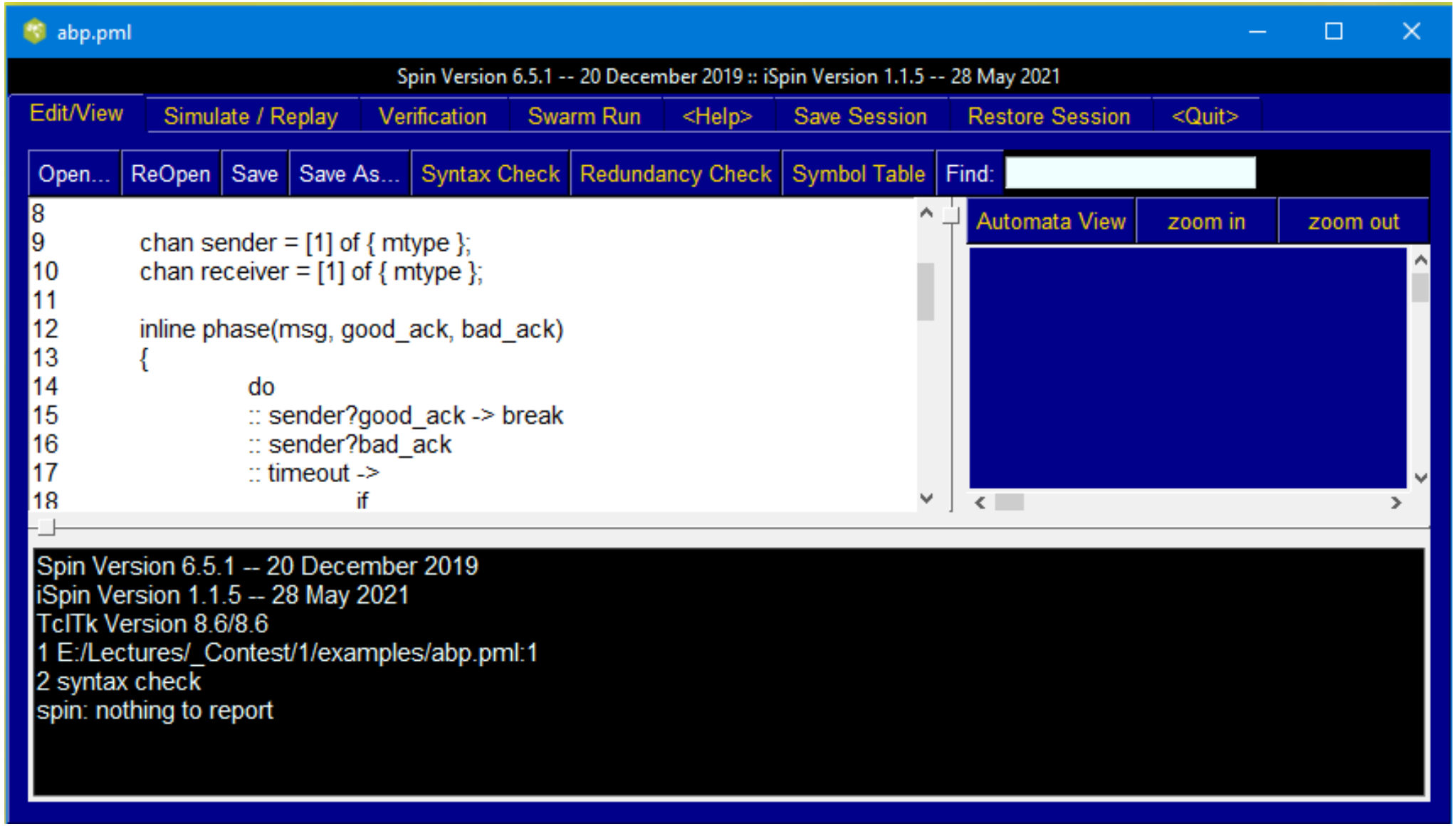
https://spinroot.com/

MODEL CHECKING

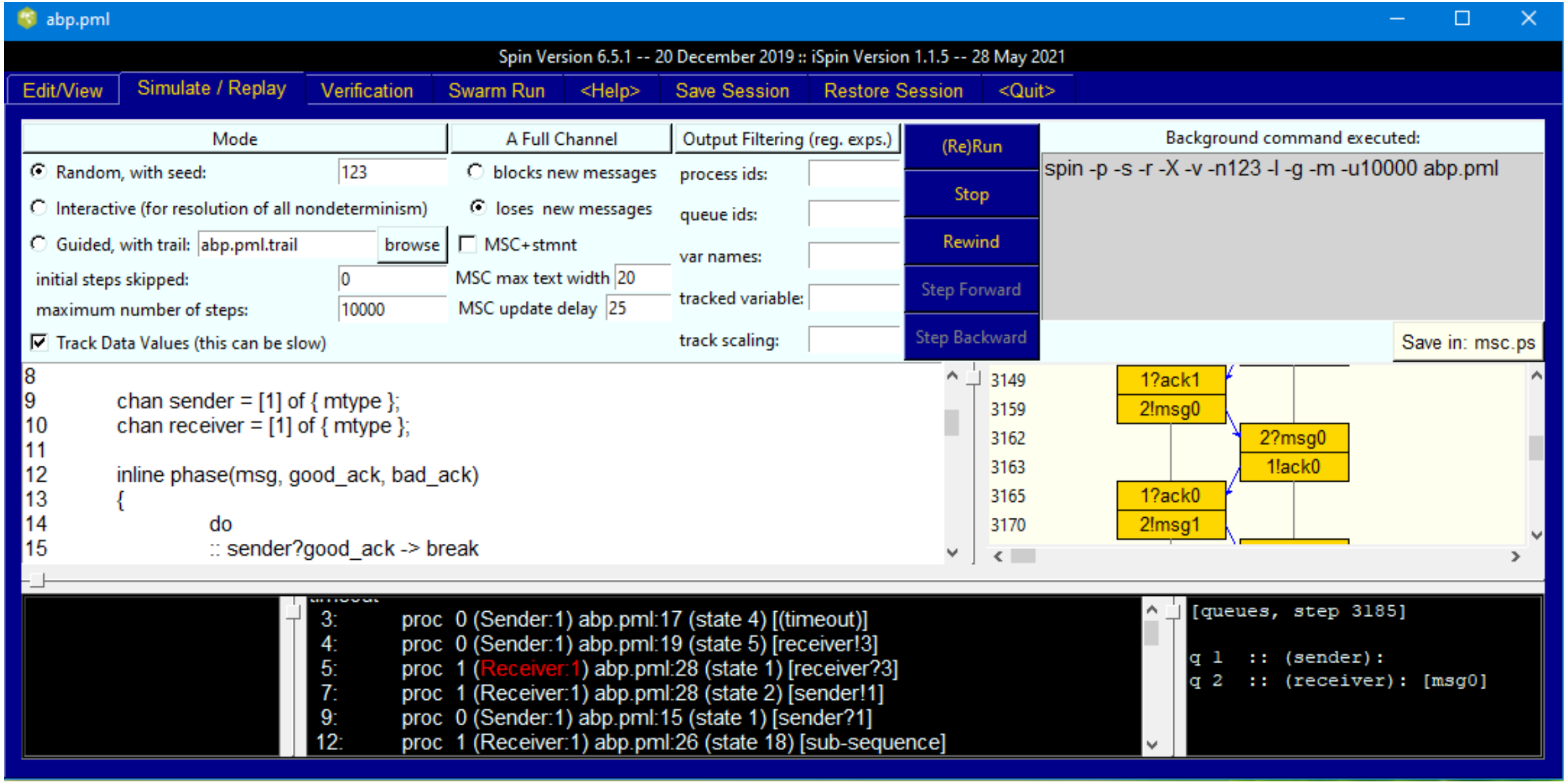
iSPIN

GUI

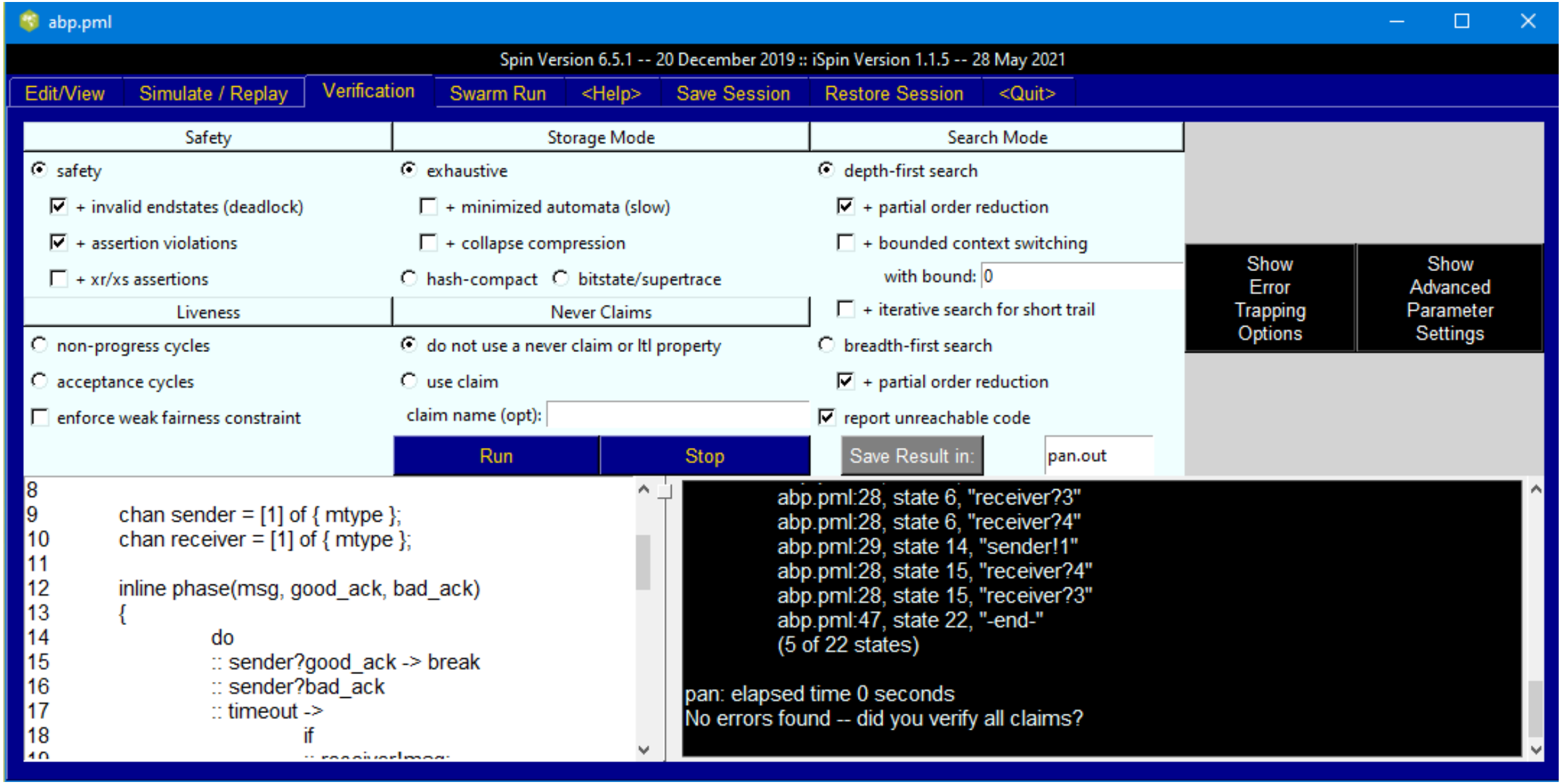
iSPIN


abp.pml
Spin Version 6.5.1 -- 20 December 2019 :: iSpin Version 1.1.5 -- 28 May 2021
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Open...
ReOpen
Save
Save As...
Syntax Check
Redundancy Check
Symbol Table
Find:
Automata View
zoom in
zoom out
8
9 chan sender = [1] of { mtype };
10 chan receiver = [1] of { mtype };
11
12 inline phase(msg, good_ack, bad_ack)
13 {
14 do
15 :: sender?good_ack -> break
16 :: sender?bad_ack
17 :: timeout ->
18 if
Spin Version 6.5.1 -- 20 December 2019
iSpin Version 1.1.5 -- 28 May 2021
TclTk Version 8.6/8.6
1 E:/Lectures/_Contest/1/examples/abp.pml:1
2 syntax check
spin: nothing to report

iSPIN


abp.pml
Spin Version 6.5.1 -- 20 December 2019 :: iSpin Version 1.1.5 -- 28 May 2021
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Mode
Random, with seed: 123
Interactive (for resolution of all nondeterminism)
Guided, with trail: abp.pml.trail
browse
initial steps skipped: 0
maximum number of steps: 10000
Track Data Values (this can be slow)
A Full Channel
blocks new messages
loses new messages
MSC+stmnt
MSC max text width 20
MSC update delay 25
Output Filtering (reg. exps.)
process ids:
queue ids:
var names:
tracked variable:
track scaling:
(Re)Run
Stop
Rewind
Step Forward
Step Backward
Background command executed:
spin -p -s -r -X -v -n123 -l -g -m -u10000 abp.pml
Save in: msc.ps
8
9 chan sender = [1] of { mtype };
10 chan receiver = [1] of { mtype };
11
12 inline phase(msg, good_ack, bad_ack)
13 {
14 do
15 :: sender?good_ack -> break
3149
3159
3162
3163
3165
3170
1?ack1
2!msg0
2?msg0
1!ack0
1?ack0
2!msg1
3: proc 0 (Sender:1) abp.pml:17 (state 4) [(timeout)]
4: proc 0 (Sender:1) abp.pml:19 (state 5) [receiver!3]
5: proc 1 (Receiver:1) abp.pml:28 (state 1) [receiver?3]
7: proc 1 (Receiver:1) abp.pml:28 (state 2) [sender!1]
9: proc 0 (Sender:1) abp.pml:15 (state 1) [sender?1]
12: proc 1 (Receiver:1) abp.pml:26 (state 18) [sub-sequence]
[queues, step 3185]
q 1 :: (sender):
q 2 :: (receiver): [msg0]

iSPIN


abp.pml
Spin Version 6.5.1 -- 20 December 2019 :: iSpin Version 1.1.5 -- 28 May 2021
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Safety
safety
+ invalid endstates (deadlock)
+ assertion violations
+ xr/xs assertions
Liveness
non-progress cycles
acceptance cycles
enforce weak fairness constraint
Storage Mode
exhaustive
+ minimized automata (slow)
+ collapse compression
hash-compact
bitstate/supertrace
Never Claims
do not use a never claim or ltl property
use claim
claim name (opt):
Search Mode
depth-first search
+ partial order reduction
+ bounded context switching
with bound: 0
+ iterative search for short trail
breadth-first search
+ partial order reduction
report unreachable code
Show Error Trapping Options
Show Advanced Parameter Settings
Run
Stop
Save Result in:
pan.out
8
9 chan sender = [1] of { mtype };
10 chan receiver = [1] of { mtype };
11
12 inline phase(msg, good_ack, bad_ack)
13 {
14 do
15 :: sender?good_ack -> break
16 :: sender?bad_ack
17 :: timeout ->
18 if
abp.pml:28, state 6, "receiver?3"
abp.pml:28, state 6, "receiver?4"
abp.pml:29, state 14, "sender!1"
abp.pml:28, state 15, "receiver?4"
abp.pml:28, state 15, "receiver?3"
abp.pml:47, state 22, "-end-"
(5 of 22 states)
pan: elapsed time 0 seconds
No errors found -- did you verify all claims?

Promela

Process Meta Language

Объекты Promela: процессы

Объекты: *процессы*, *переменные* и *каналы сообщений*

- *Процессы*
  - объявление **proctype**
    - объявляются глобально
    - задаёт поведение, но не запускает процесс
    - должен быть объявлен хотя бы один процесс

- Запуск процессов
  - префикс **active**:

```
active [2] proctype foo(){ printf("MSC: my pid is: %d\n", _pid)}
```

  - оператор **run**:
    - `active proctype bar() { run foo() }`

- **_pid** – зарезервированная переменная хранит неотрицательное значение уникального идентификатора

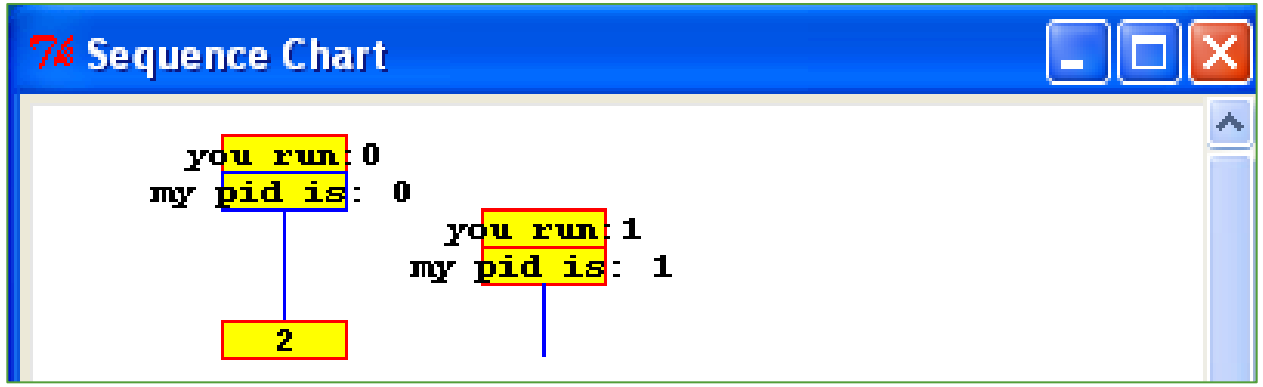
Объекты Promela: процессы

- Тело процесса:
  - объявления данных и операторов (может быть пустым)
  - разделители операторов:
    - точка с запятой “`;`”
      - не указатель конца оператора
      - допускается пустой оператор `; ; ; ;`
    - стрелка “`->`”
      - способ указания на причинно-следственное отношение между двумя операторами

- Состояние переменной или канала сообщений
  - изменяется или проверяется только в процессах.

Объекты Promela: процессы

```
active [2] proctype you_run(){
    printf("MSC: my pid is: %d\n", _pid)
}
```



- Окно симуляции: создано два процесса типа **you_run**.
- Диаграмма взаимодействий (Sequence Chart)
  - каждый столбец отображает один запущенный процесс.

Объекты Promela: процессы

- **init** – базовый процесс в Promela
  - всегда активируется в начальном состоянии модели.
  - ему нельзя передать параметры или создать его копию.
  - его идентификатор всегда равен 0.
  - лишние процессы увеличивают размер модели
- Выполняющийся процесс *завершается*, когда достигает конца своего тела, но не позже процесса, который его запустил.
- Количество процессов в Spin не более 256.

```
proctype you_run(byte x) {
    printf("MSC: x is %d\n", x);
    printf("MSC: my pid is = %d\n", _pid)
}
init {
    run you_run(0);
    run you_run(1)
}
```

Объекты Promela: переменные

- *Переменные*
  - глобальные – объявлены вне описания процесса
  - локальные – объявлены в описании процесса
    - нельзя ограничить доступ к локальной переменной для части процесса
      - нет аналога блока или области видимости
  - инициализируются нулями (**`false`**).
- Одномерные массивы
  - **`byte state[N]`**
    - **`state[0] = state[3] + 5 * state[3*2/n]`**
      - **`n`** – константа или переменная.
  - нумерация с 0
  - индекс массива – любое выражение с натуральным значением
    - вне диапазона 0.. N-1 результат не определен
- Многомерные массивы могут быть заданы неявно с помощью конструкции **`typedef`**.

Тип данных	Диапазон
`bit`	0,1
`bool`	false, true
`byte`	0..255
`chan`	1..255
`mtype`	1..255
`pid`	0..255
`short`	-2^{15} .. $2^{15}-1$
`int`	-2^{31} .. $2^{31}-1$
`unsigned`	0 .. $2^{32}-1$

Объекты Promela: переменные

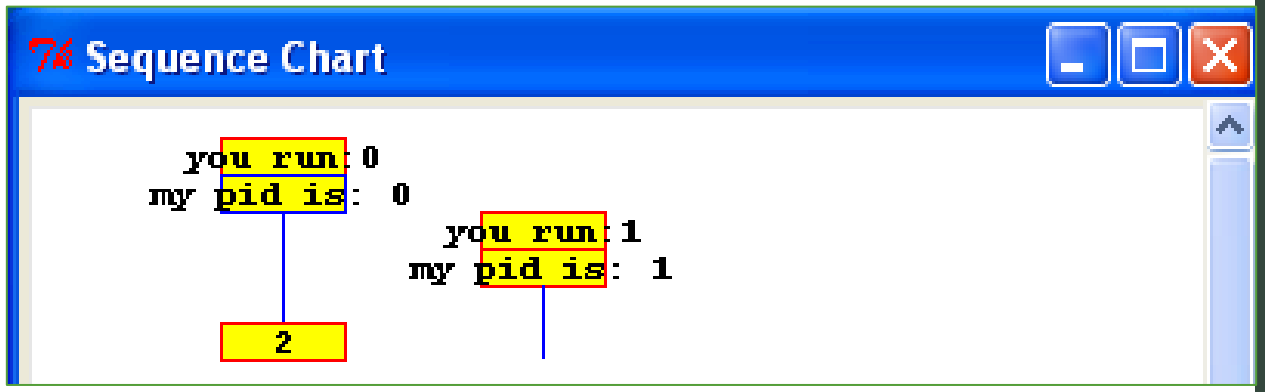
- Перечислимый тип
  - **mtype**
  - символьные значения переменных
  - одно или несколько объявлений
    - все переменные, объявленные как **mtype**, могут принимать все объявленные значения
  - В **mtype** можно задать не более 255 значений.

```
mtype = { grandad, grandma, grandaughter, dog, cat, mice, turnip };
mtype = { mammal, vegetable };
init {
    mtype n = grandad; /* инициализация n значением grandad */
    printf("MSC: %e ", n);
    n = vegetable; /* присвоение n значения vegetable */
    printf("MSC: is not %e\n ", n)
}
```

Объекты Promela: переменные

- Оператор **printf**
  - два аргумента: строка и список аргументов
  - форматы выводимых переменных
    - **d** – целое в десятичном формате,
    - **i** – беззнаковое целое значение,
    - **c** – один символ,
    - **e** – константа типа **mtype**
  - для вывода сообщения на диаграмме взаимодействия iSpin строка начинается с символов “MSC: ”
    - удобно для отладки

```
active [2] proctype you_run(){
    printf("MSC: my pid is: %d\n", _pid)
}
```



Объекты Promela: каналы

- *Каналы* моделируют передачу данных от одного процесса к другому
- Объявление локально или глобально служебным словом **chan**:
  - **chan qname = [16] of { short }**
    - канал **qname** с буфером 16 для сообщений типа **short**
- Передают сообщения в порядке FIFO: первым вошел – первым вышел
- Оператор отправки сообщений "**!**":
  - **qname ! expr**
  - добавляет это значение к концу очереди в канале
  - *выполняется*, если канал назначения не переполнен, иначе *блокируется*
- Оператор приема сообщения "**?**":
  - **qname ? msg / qname ?? const**
  - сохраняет значение из начала/произвольного места очереди канала в переменную **msg**, удаляя его из очереди
  - *выполняется*, если канал назначения не пуст, иначе *блокируется*

Объекты Promela: каналы

- Составные сообщения
  - содержат конечное число полей
  - `chan pname = [16] of { byte, int, chan }`
    - одно восьмибитное значение (типа `byte`)
    - одно 32-битное значение (типа `int`)
    - имя канала.
- Передача идентификатора канала от одного процесса другому
  - в сообщении
  - как параметр экземпляра процесса
- Нельзя использовать массивы как поля сообщения
- Отправка нескольких значений в одном сообщении
  - `pname ! expr1, expr2, expr3`
- Приём такого сообщения
  - `pname ? var1, var2, var3`

Объекты Promela: каналы

- Использование первого поля сообщения для задания *типа сообщения*
  - В каналах данные **mtype** всегда интерпретируются символически, а не численно

```
/* объявляется тип сообщения */
mtype = { ack, nak, err, next, accept }
/* объявляются переменные этого типа */
mtype msgtype1, msgtype2;
```

- `chan tname = [4] of { mtype, int, bit };`
- `tname ! msgtype (data, b)`
  - `tname ! msgtype, data, b`

Объекты Promela: каналы

- Можно посылать или принимать константы
  - средство синхронизации

```
tname ! ack, var, 0
tname ? ack (data, 1)
```

- Оператор получения сообщения при наличии констант выполним,
  - только если сообщение в начале буфера канала в соответствующих полях имеет значения заданных констант.
    - В противном случае он будет блокирован.
      - Оператор получения сообщения невыполним,
        - если в начале буфера находится сообщение **< ack, 15, 0 >**.
- Невыполнимые операторы приостанавливают процесс, пока не станут выполнимыми.
  - В случае операций над каналами это позволяет моделировать коммуникацию «точка-точка» нескольких процессов, связанных одним каналом.

Объекты Promela: каналы

- Функции для каналов
  - **len**, **empty**, **nempty**, **full**, **nfull**.
- **len(qname)**
  - количество сообщений в канале **qname**
  - невыполним, если используется как оператор справа от присваивания и канал пуст, т.к.
    - возвращается нулевой результат
- Отправка сообщений **msgtype**, если канал **qname** не переполнен:
  - **(len(qname) < MAX) -> qname ! msgtype**
  - Если доступ к каналу **qname** разделяется несколькими процессами, то выполнение второго оператора необязательно будет происходить сразу после выполнения первого оператора проверки.

Объекты Promela: каналы

Взаимодействие рандеву

- `chan port = [0] of { byte }`
- *Рандеву-канал*
  - буфер рандеву-канала равен нулю
    - может передавать, но не может хранить сообщения.
  - Взаимодействия процессов по рандеву-каналам по определению синхронны

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A() {
    name ! msgtype(4);
    name ! msgtype(1) }
proctype B() {
    byte state;
    name ? msgtype(state) }
init {
    atomic { run A(); run B() }
}
```

Объекты Promela: каналы

- Канал **name** объявлен здесь как глобальный рандеву-канал.
- Два процесса синхронно выполнят свои первые операторы:
  - подтверждение связи (хэндшейк) по сообщению **msgtype**
  - передачу значения 4 в локальную переменную **state**.
- Второй оператор отправки в процессе **A** невыполним
  - нет соответствующей операции приема сообщения в процессе **B**.

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A() {
    name ! msgtype(4);
    name ! msgtype(1)
}
proctype B() {
    byte state;
    name ? msgtype(state)
}
init {
    atomic { run A(); run B() }
}
```

Объекты Promela: каналы

```
#define msgtype 1
chan name = [0] of { byte, byte };
proctype A() {
    name ! msgtype(4);
    name ! msgtype(1)
}
proctype B() {
    byte state;
    name ? msgtype(state)
}
init { atomic { run A(); run B()}}
```

- Размер буфера **name** равен 2
  - A может закончить выполнение
    до того, как B начнет работу.
- Размер буфера **name** равен 1
  - Процесс A может закончить свою первую отправку
    - блокируется на втором действии, т.к. теперь канал полон.
  - Процесс B читает первое сообщение и завершается.
  - В этой точке A становится опять выполнимым и завершается, оставляя свое последнее сообщение в канале.
- Взаимодействия рандеву бинарные:
  - только два процесса, отправитель и получатель, могут быть синхронизированы.

Операторы Promela: выполнимоть

Правило выполнимости

- Выполнимость обеспечивает базовые средства для моделирования синхронизации процессов.
- Любой оператор или выполним, или блокирован.
- Основные типы операторов:
  - оператор вывода переменных **`printf`** (всегда выполним),
  - оператор присваивания (всегда выполним),
  - операторы ввода/вывода (при передаче данных по каналам),
  - операторы-выражения.
- Если процесс достиг точки кода, где содержится невыполнимый оператор, то процесс блокируется.
  - Оператор может стать выполнимым, если другой активный процесс выполнит действия, позволяющие оператору, блокированному в данном процессе, выполняться далее.

Цикл ожидания: `while (a != b) -> skip`
Promela: `(a == b)`

Операторы Promela

- Два процесса разделяют доступ к глобальной переменной **state**.
- Процессы ожидают выполнения условия (**state == 1**).
- Если завершается, то **state** может иметь значение: 0, 1 или 2.
- Если один из процессов успеет изменить значение **state** до проверки условия другим процессом, то другой процесс *заблокируется*.

```
byte state = 1;
active proctype A(){
    byte tmp;
    (state==1) -> tmp = state;
    tmp = tmp+1;
    state = tmp
}
active proctype B(){
    byte tmp;
    (state==1) -> tmp = state;
    tmp = tmp-1;
    state = tmp
}
```

Операторы Promela: выражения

- Выражения в Promela рассматриваются как утверждения, т.е. проверяются на истинность/ложность в любом контексте.
- Выражение выполнимо тогда и только тогда, когда
  - его булево значение истинно (*true*)
    - любому ненулевое значение

Операторы	
`() []`	Скобки, скобки массивов
`! ++ --`	отрицание, плюс 1, минус 1
`* / %`	умножение, деление, остаток
`+ -`	сложение, умножение
`<< >>`	сдвиг влево, сдвиг вправо
`< <= > >=`	сравнение
`== !=`	равенство, не равенство
`&`	побитовое И
`^`	побитовое исключающее ИЛИ
`|`	побитовое ИЛИ
`&&`	логическое И
`||`	логическое ИЛИ
`-> :`	оператор условий
`=`	присваивание

Операторы Promela: присваивание и печать

- **`variable = expression`**
  - не рассматривается как выражение, так же, как и оператор печати
  - сначала оценивается выражение, стоящее справа
  - результат выражения приводится к типу **`variable`**
  - **`variable`** получает это значение.
- Инкремент и декремент
  - только постфиксные (**`a++`**, не **`++a`**)
  - только в выражении, но не в операторе присваивания
- Оператор **`printf`**
  - вывод значений переменных или текста
  - не меняет состояние системы **`printf`**, как и **`skip`**
    - используются в моделях, когда нужен всегда выполнимый шаг
  - вывод текста производится только в режиме симуляции
    - является побочным действием оператора

Операторы Promela: входные/выходные операторы

- Входные/выходные операторы выполняются
  - если возможна отправка/получение сообщения
  - иначе процесс блокируется
- Тестирование возможности отправки/получения без выполнения.
  - взять аргументы оператора в квадратные скобки
    - оператор не выполняется, но вычисляется его значение как выражения
  - **`qname ? [ack, var, 0] / qname ?? [ack, var, 0]`**
    - проверка, что очередным/каким-либо сообщением в канале **`qname`** является структура, состоящая из
      - мнемонического значения **`ack`**, некоторого значения переменной **`var`** и значения 0.
  - Если утверждение выполнимо, то возвращается 1, иначе возвращается 0.
- Недопустимы выражения вида
  - **`(qname ? var == 0)`** или **`(a > b && qname ! 123)`**
  - не могут быть вычислены без побочных эффектов
    - попытки выполнить операции ввода/вывода
- Сами операторы отправки и получения данных не являются выражениями

Операторы Promela: составные операторы

Блок **atomic**

- **atomic {op1 op2 ... opn}**
  - блок **op1, op2 ... , opn** выполняется как неделимый модуль, не чередующийся с другими процессами.
  - другие процессы “видят” разделяемые глобальные переменные и каналы, используемые в блоке **atomic**, либо до, либо после выполнения всей последовательности операторов.
  - если оператор внутри **atomic** невыполним, то весь блок невыполним и другой процесс может начать действовать.
  - инструмент понижения сложности моделей верификации –
    - уменьшение числа глобальных состояний
      - блоки **atomic** ограничивают количество чередований
    - Обеспечить корректную реализацию блоков **atomic**

- **atomic** предотвращает доступ конкурирующего процесса к глобальной переменной

```
byte state = 1;
active proctype A(){
    atomic {
    (state==1) -> state = state+1 } }
active proctype B(){
    atomic {
    (state==1) -> state = state-1 } }
```

Операторы Promela: составные операторы

Блок детерминированных шагов **`d_step`**

- **`d_step {op1 op2 ... opn}`**
  - блок **`op1, op2 ... , opn`** выполняется как неделимый модуль, не чередующийся с другими процессами.
  - другие процессы “видят” разделяемые глобальные переменные и каналы, используемые в блоке **`d_step`**, либо до, либо после выполнения всей последовательности операторов.
  - если оператор внутри **`d_step`** невыполним, то весь блок невыполним, и это *ошибка моделирования*.
  - более мощное понижение сложности верификации –
    - для последовательности **`atomic`** Spin генерирует переходы для других процессов, в отличии от **`d_step`**.

Операторы Promela: составные операторы

Условный оператор

- **if**
  - содержит минимум две последовательности операторов.
  - выполняется только одна последовательность из списка выполнимых.
    - последовательность выполнима, если ее первый оператор выполним.
    - первый оператор называется *защитой* (*guard)* или *условием*.
  - если выполнимы несколько операторов, недетерминировано выбирается какой-либо один
    - порядок перечисления альтернатив выбора несущественен
  - если все условия невыполнимы, то процесс заблокируется до выполнения одного из условий.
  - нет ограничений на типы выражений для условий.

```
if
  :: (a != b) -> option1
  :: (a == b) -> option2
fi
```

Операторы Promela: составные операторы

- **option1** выполнима, если канал содержит сообщение-константу **a**.
- **option2** выполнима, если канал содержит сообщение-константу **b**.
- Какой из операторов будет выполнен, зависит от относительных скоростей процессов.

```
#define a 1
#define b 2
chan ch = [1] of { byte };
proctype A(){ ch ! a }
proctype B(){ ch ! b }
proctype C(){
    if
        :: ch ? a -> option1
        :: ch ? b -> option2
    fi
}
init{
    atomic {
        run A(); run B(); run C()
    }
}
```

Операторы Promela: составные операторы

- Процесс для изменения значения переменной **count**
- Оба выражения в примере всегда выполнимы
  - выбор между ними полностью недетерминирован

```
byte count;
proctype counter(){
    if
        :: count = count + 1
        :: count = count - 1
    fi
}
```

Операторы Promela: составные операторы

- Для выполнения цикла может быть выбрана только одна опция
- После завершения выполнения выбранной опции управление передается на начало оператора цикла
- Выход из цикла – с помощью оператора **`break`**
- В примере
  - цикл будет прерван, когда **`count == 0`**.
  - два других оператора всегда выполнимы
    - выход недетерминирован

```
byte count;
proctype counter(){
    do
        :: count = count + 1
        :: count = count - 1
        :: (count == 0) -> break
    od
}
```

Операторы Promela: составные операторы

- Гарантия завершения цикла при нужном условии

```
byte count;
proctype counter(){
    do
        :: (count != 0) ->
            if
            :: count = count + 1
            :: count = count - 1
            fi
        :: (count == 0) -> break
    od
}
```

Операторы Promela: составные операторы

- Условие **`else`**.
  - выполнимо в операторах выбора или цикла, только если ни одно другое условие не выполнимо

```
byte count;
proctype counter(){
    do
        :: (count != 0) ->
            if
            :: count = count + 1
            :: count = count - 1
            fi
        :: else -> break
    od
}
```

  - условие **`else`** выполнимо, когда
    - `!(count != 0)` ≅ `(count == 0)`.

Операторы Promela: составные операторы

- Оператор *безусловного перехода* **goto**
  - всегда выполним, если существует метка, на которую выполняется переход
- Алгоритм Эвклида нахождения НОД:

```
proctype Euclid (int x, y){
    do
        :: (x > y) -> x = x - y
        :: (x < y) -> y = y - x
        :: (x == y) -> goto done
    od;
    done: skip
}
```

- *Метка* может быть поставлена только перед оператором
  - пустой оператор **skip**
    - оператор-заполнитель всегда выполним, но не производит никакого эффекта.

Примеры Promela программ: фильтр

- Фильтрующий и объединяющий процессы выполняются бесконечно

```
#define N 128
#define size 16
chan ch = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };
proctype split(){
    short data;
    do :: ch ? data ->
          if :: (data >= N) -> large ! data
             :: (data < N)  -> small ! data
          fi
    od
}
proctype merge(){
    short data;
    do :: if :: large ? data
             :: small ? data
          fi;
          ch ! data
    od
}
init{   ch ! 345; ch ! 13; ch ! 6777; ch!32; ch ! 0;
    run split(); run merge() }
```

Примеры Promela программ: факториал

- Рекурсивные процессы
  - Возвращаемое значение передается обратно в вызывающий процесс через глобальную переменную или сообщение

```
proctype fact( int n; chan p) {
    chan child = [1] of { int };
    int result;
    if
      :: (n <= 1) -> p ! 1
      :: (n >= 2) -> run fact(n-1, child);
                     child ? result;
                     p ! n*result
    fi
}
init{
    chan child = [1] of { int };
    int result;
    run fact(7, child);
    child ? result;
    printf("MSC: result: %d\n", result)
}
```

Примеры Promela программ: система управления

```
proctype Environment() {
do
::  turn ? Environment_n;
    atomic {
     if
     :: action = true;
     :: action = false;
     fi;
     if
     :: opened -> time++;
     :: else -> time = 0;
     fi;
    }
    turn ! Sensor_n;
od; }
proctype Sensor() {
do
:: turn ? Sensor_n;
   atomic {
    if
    :: !opened && action -> valid = true;
    :: else -> valid = false;
    fi;
   }
   turn ! Controller_n;
od; }
```

```
proctype Controller() {
do
:: turn ? Controller_n;
   atomic {
    if
    :: valid ->
         opened = true;
         valid = false;
    :: else -> skip;
    fi;
    if
    :: (!valid && opened) || time >= 15) ->
         opened = false;
         valid = false;
         time = 0;
    :: else -> skip;
    fi;
   }
   turn ! Environment_n;
od;
}
init { run Environment(); run Sensor(); run Controller()}
ltl p0 {[](!timeout)}
ltl p1 {[]( time < 16 )}
```

Свойства систем в Promela и SPIN

Оператор **`assert`**

- задаёт локальные инварианты
  - свойства, которые должны выполняться в определенных точках программы
- **`assert`** (**`expr`**)
- **`expr == true`**
  - оператор **`assert`** не производит никакого эффекта.
- **`expr == false`**
  - при симуляции SPIN выдаст сообщение «Error: assertion violated».
  - при верификации нарушение операторов **`assert`** проверяется на всех конечных вычислениях.
- В режиме симуляции могут быть проверены только свойства системы, описанные оператором **`assert`**
  - остальные только в режиме верификации.

Свойства систем в Promela и SPIN

- Формулы линейной темпоральной логики LTL
  - темпоральные операторы
    - ***F****p* – **`<>p`**
      - Свойство *p* будет выполняться в каком-то последующем состоянии пути (***F****uture*)
    - ***G****p* – **`[]p`**
      - Свойство *p* выполняется в каждом состоянии пути (***G****lobally*)
    - *p****U****q* – **`p U q`**
      - Свойство *p* верно до тех пор, пока не начнёт выполняться *q* (***U****ntil*)
  - атомарные высказывания – булевы выражения языка Promela
  - булевы операторы **`&&, ||, ->`** и **`!`**

  - **`ltl prp {[](p -> (p U q))}`**
    - Всегда, если *p* стало истинным, то когда-нибудь в будущем станет истинным *q*, а *p* будет оставаться истинным до тех пор, пока *q* не станет истинным

Типичные темпоральные формулы

- $\boldsymbol{G}\varphi$ (*инвариант*)
  - свойство φ будет истинным всегда
- $\boldsymbol{F}\varphi$ (*достижимость*)
  - свойство φ будет истинным всегда
- $\boldsymbol{GF}\varphi$ (*живость*)
  - свойство φ будет истинным бесконечно часто
- $\boldsymbol{FG}\varphi$ (*стабилизация*)
  - когда-нибудь свойство φ станет истинным и останется таким навсегда
- $\boldsymbol{G}\,(\varphi \rightarrow \boldsymbol{F}\psi)$ (*отклик, реакция*)
  - если получен запрос φ, то рано или поздно будет отклик ψ
  - если получен стимул φ, то рано или поздно будет реакция ψ
  - если случился триггер φ, то рано или поздно будет реакция ψ
- $\boldsymbol{G}\,(\varphi \rightarrow \varphi \boldsymbol{U} \psi)$ (*отклик, реакция*)
  - если возникла ситуация φ, то она продолжается, пока не произойдёт реакция ψ

Свойства систем в Promela и SPIN

Процесс **never**

- дает возможность задания глобальных инвариантов
  - **assert** не предназначен для проверки во всех состояниях системы.
- описание поведения, которое не должно произойти в системе.
- предназначен для слежения за поведением системы
  - не оказывает влияния на состояния
    - нельзя объявить переменные
    - нельзя изменить значение переменной
    - нельзя манипулировать каналами сообщений
- в модели может быть только один процесс **never**.
- учитывается только при верификации.
- позволяет проверить свойство системы
  - в начальном состоянии и
  - после каждого шага вычисления
    - после выполнения каждого оператора любого процесса

```
never { /* !G (p → (p U q)) */
S0:     do
        :: p && !q -> break
        :: true
        od;
accept: do
        :: p && !q
        :: !(p || q) -> break
        od
}
```

Свойства систем в Promela и SPIN

- Проверка выполнения условия **p** на каждом шаге системы:

```
never {
    do
      :: !p -> break
      :: else
    od
}
```

- Выполняется на каждом шаге системы.
- Если условие **p** ложно
  - процесс **never** прерывается, переходя в завершающее состояние.
    - Завершение **never** интерпретируется как ошибочное поведение анализируемой системы.
- Если **p** всегда истинно,
  - процесс **never** остается в цикле
    - ошибки в анализируемой системе нет.

Свойства систем в Promela и SPIN

- Проверка выполнения условия **p** на каждом шаге системы:
  - без **never**

```
active proctype monitor(){
    atomic { !p -> assert(false) }
}
```

- Процесс **monitor** может инициировать выполнение блока **atomic** в любой точке вычисления системы.
  - В любом достижимом состоянии системы, в котором инвариант **p** нарушается, **monitor** сообщает об ошибке с помощью оператора **assert**.

Свойства систем в Promela и SPIN

- Всегда, если p стало истинным, то когда-нибудь в будущем станет истинным q, а p будет оставаться истинным до тех пор, пока q не станет истинным.
  - LTL: $\boldsymbol{G}(p \rightarrow (p\ \boldsymbol{U}\ q))$
- При проверке модели нас не интересуют все те вычисления, в которых свойство удовлетворяется
  - для модели проверяется наличие в ней вычислений, на которых свойство нарушается.
    - $\neg\boldsymbol{G}(p \rightarrow (p\ \boldsymbol{U}\ q))$
    - p стало истинным, а q осталось ложным на всем вычислении, или p стало ложным до того как, q стало истинным.

Свойства систем в Promela и SPIN

- Нарушение свойства, где q остается ложным всегда
  - только на бесконечных вычислениях.
    - **assert** не подойдёт
      - проверяется только на конечных вычислениях
- В режиме верификации в начальном состоянии системы проверяется возможность выполнения первого оператора процесса **never**.
  - метка **S0**: цикл с недетерминированным выбором.

```
never { /* !G (p → (p U q)) */
S0:     do
        :: p && !q -> break
        :: true
        od;
accept: do
        :: p && !q
        :: !(p || q) -> break
        od
}
```

Свойства систем в Promela и SPIN

- Условие **`true`**
  - всегда выполнимо и не влияет на вычисления.
  - возвращает процесс **`never`** в его начальное состояние.
  - не позволяет заблокироваться
- Условие **`p && !q`**
  - поведение модели: стало истинно p, но q еще не истинно.
  - с этого состояния может начаться некорректная траектория выполнения анализируемой программы.

```
never { /* !G (p → (p U q)) */
S0:     do
        :: p && !q -> break
        :: true
        od;
accept: do
        :: p && !q
        :: !(p || q) -> break
        od
}
```

Свойства систем в Promela и SPIN

- Некорректная траектория
  - во всех последующих состояниях будет истинно p и ложно q
    - вечное пребывание в метке **accept**
  - если встретится состояние, в котором не будут истинны ни p, ни q
    - завершение процесса **never**

```
never { /* !G (p → (p U q)) */
S0:     do
        :: p && !q -> break
        :: true
        od;
accept: do
        :: p && !q
        :: !(p || q) -> break
        od
}
```

- p и q истинны
  - ни одно из условий выбора невыполнимо
    - процесс **never** заблокируется.
      - блокировка – желаемое поведение
        - не завершился и
        - не проходил бесконечный цикл с меткой **accept**