Математические методы верификации схем и программ

Лекторы:

Захаров Владимир Анатольевич

Подымов Владислав Васильевич

e-mail рассказчика:

valdus@yandex.ru

Осень 2016

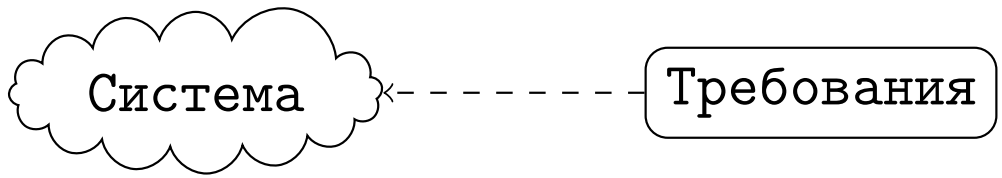
Напоминание

Какая глобальная цель перед нами стоит?

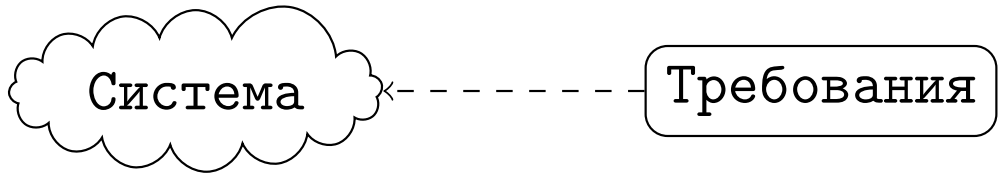
Даны

- неформальное описание системы
- содержательное описание требований к системе

Требуется проверить,
удовлетворяет ли система набору требований



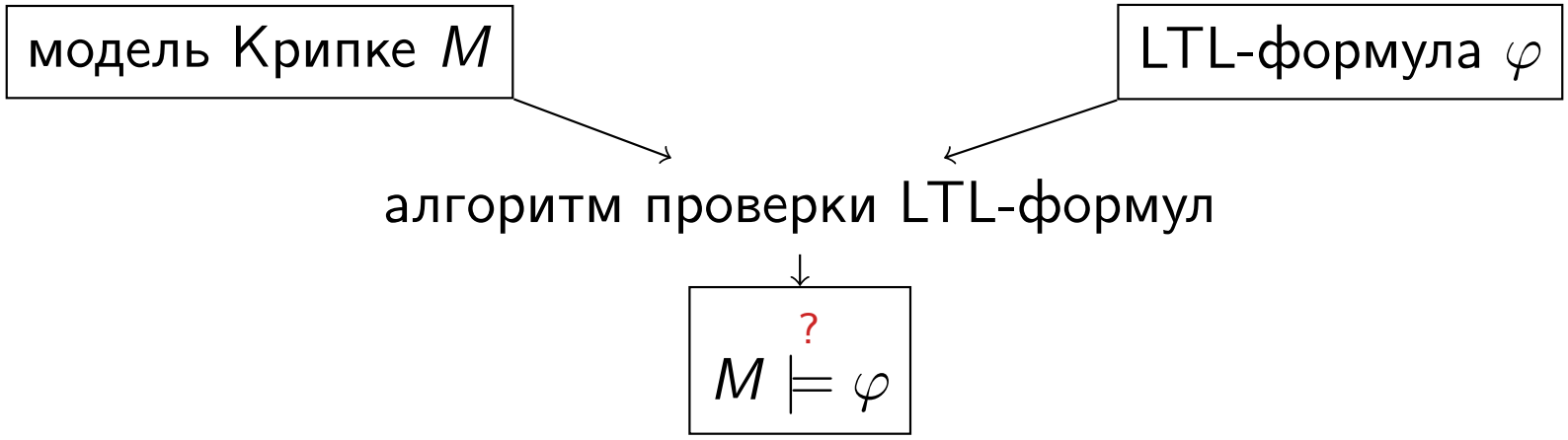
Задача model checking для LTL



Система — это модель Крипке M

Требование — это LTL-формула φ

Проверка требования — это проверка соотношения $M \models \varphi$



Средства верификации

Вот список программных средств, способных проверять выполнимость LTL-формул в *каких-то* моделях:

(*на случай если захотите их использовать*)

BANDERA	CADENCE SMV	LTSA	LTSmin
NuSMV	PAT	ProB	SAL
SATMC	SPIN	Spot	...

Disclaimer: список скорее всего неполный, и я не знаю большинства этих средств; информация взята из соответствующей страницы в википедии

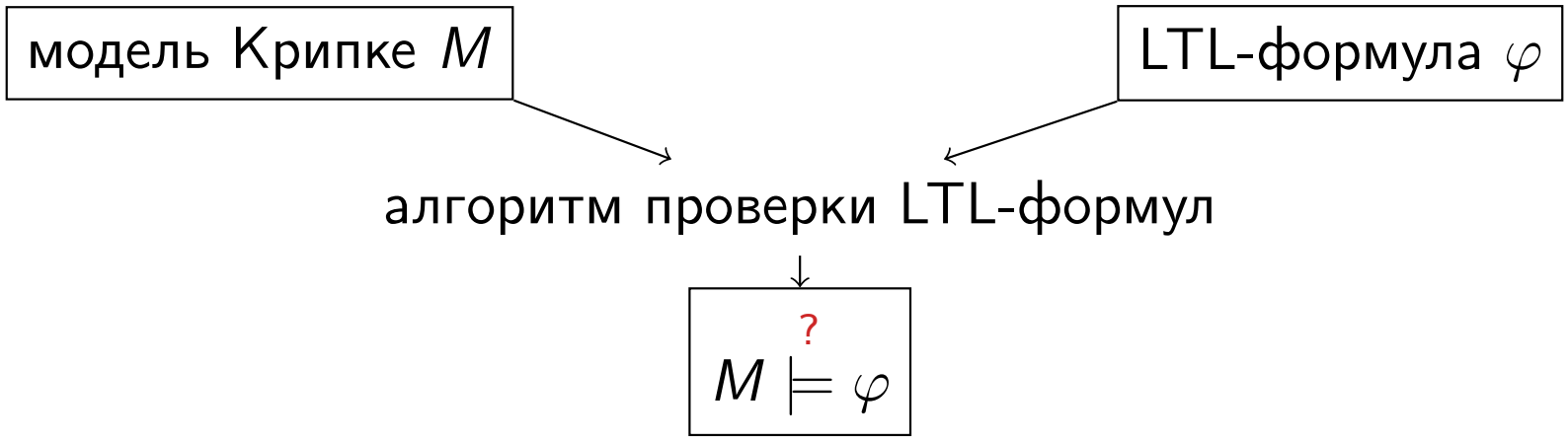
Некоторые из этих средств работают и с CTL

В курсе сосредоточим внимание на средстве SPIN:

- оно открытое и бесплатное
- оно довольно популярно
- его язык достаточно прост для понимания

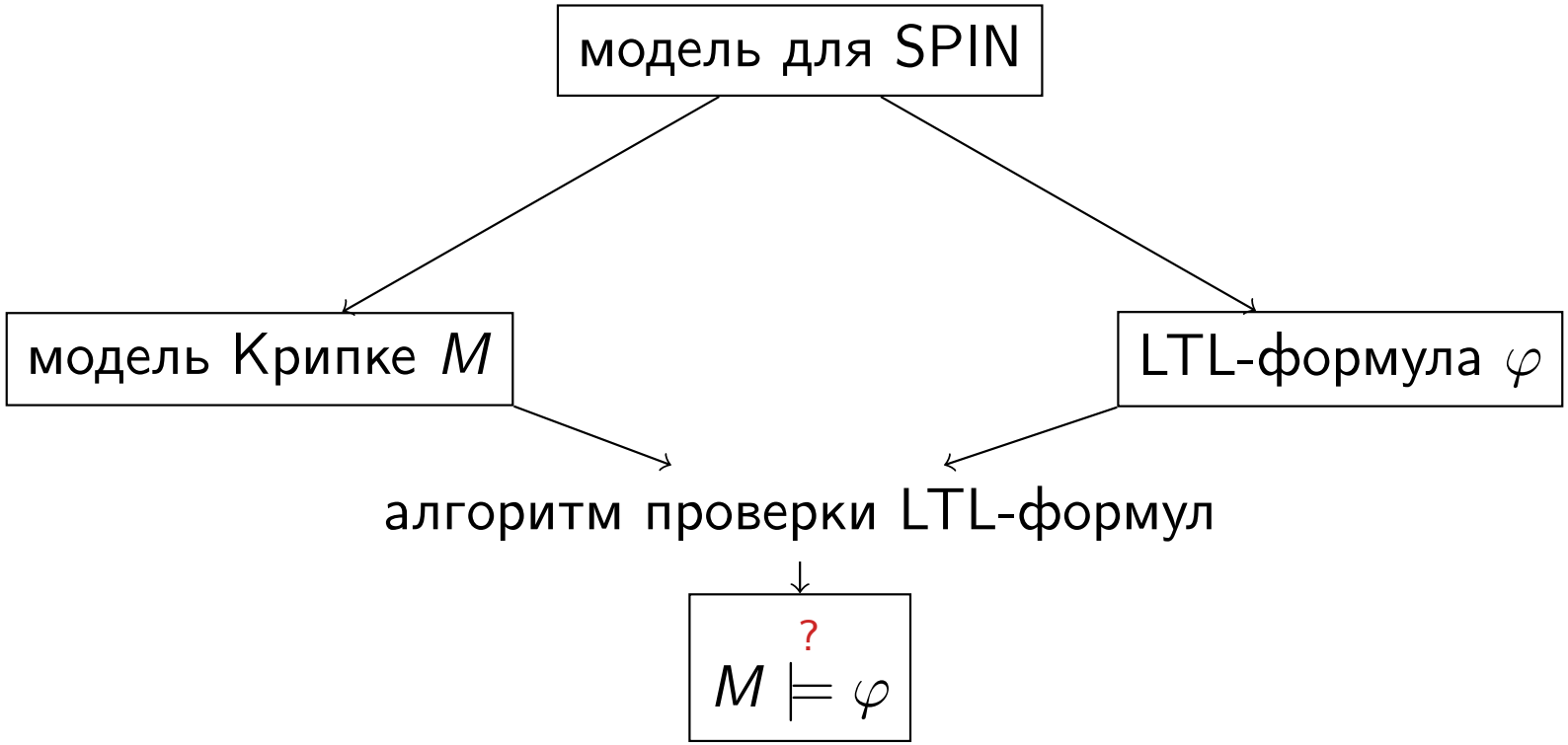
(*и более "последователен", чем язык NuSMV*)

SPIN: вступление



Всё ли так просто с этой схемой?

SPIN: вступление



Всё ли так просто с этой схемой?

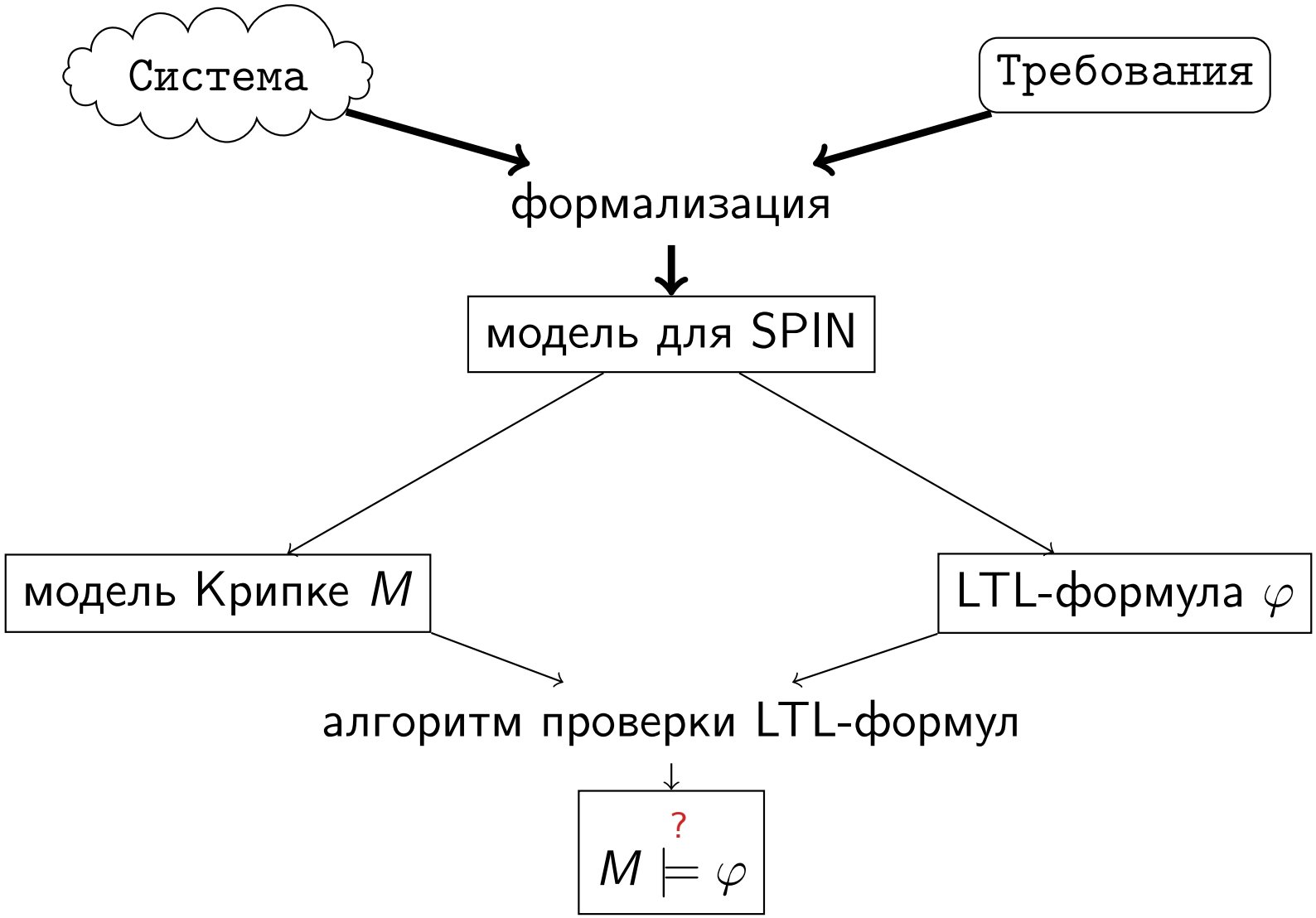
Каждое средство верификации имеет свой входной язык

Этот язык может быть довольно близок к входным данным “чистой” задачи model checking для LTL

(*как было в NuSMV для CTL*)

Язык, обрабатываемый средством SPIN, далёк от такого “чистого” описания

SPIN: вступление



Входной язык средства SPIN — PROMELA

(PROcess MEta LAnguage)

“SPIN” = Simple Promela INterpreter

SPIN: вступление

Чтобы язык PROMELA понимался проще, будем проводить аналогии между его конструкциями и конструкциями C++

При этом надо иметь в виду, что если программа на языке C++ работает **последовательно**, то модель, описанная на языке PROMELA, работает **параллельно-последовательно**

(*более точно: это параллельная композиция последовательных процессов*)

Таким цветом будут выделяться названия аналогичных конструкций в C++

PROMELA: типы переменных

- `bit` или `bool` (*bool*)
  - положительное значение: `1`, `true`
  - отрицательное значение: `0`, `false`
- `byte` (*uchar*)
  - значения: целые числа от 0 до 255
- `short` (*short*)
  - минимальный диапазон значений: $[-2^{15}, 2^{15})$
- `int` (*int*)
  - минимальный диапазон значений: $[-2^{31}, 2^{31})$
- беззнаковые числа заданной ширины
  - синтаксис: `unsigned ... : <`*число бит*`>;`

PROMELA: типы переменных

- структуры `typedef` (*struct*)
  - синтаксис:

```
typedef <имя типа> {
  <тип> <имя поля>;
  ...
  <тип> <имя поля>;
}
```

- одномерные *массивы* любого типа, кроме массивов
  - в том числе можно объявить массив структур, содержащих поля-массивы
  - синтаксис — как в C++: после имени переменной пишется [<*константа*>]
  - индексация элементов массива производится с ноля

PROMELA: типы переменных

- перечисление `mtype` (*enum*)
  - `mtype` — это **один** тип
  - синтаксис объявления этого типа:

    `mtype = {`id_1`, `id_2`, ..., `id_k`};`
  - объявление типа `mtype` можно делать несколько раз: все идентификаторы будут внесены в этот тип
  - с каждым идентификатором в `mtype` можно работать как с целочисленной константой

Трансляция значений `mtype` в целочисленные константы на примере:

```
mtype = {a, b, c, d};
mtype = {e, f, g};
mtype = {h, i, j};
```

Отождествление идентификаторов с константами происходит так:

```
a = 4   b = 3   c = 2   d = 1
e = 7   f = 6   g = 5
h = 10  i = 9   j = 8
```

PROMELA: объявления переменных

Объявления переменных и полей выглядят так же, как и в C++, единственное отличие: объявление типа должно быть отделено от объявления переменной

Пример

```
bool a;
unsigned b : 3, c : 4;
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

PROMELA: объявления переменных

Объявления переменных и полей выглядят так же, как и в C++, единственное отличие: объявление типа должно быть отделено от объявления переменной

Пример

```
bool a;
unsigned b : 3, c : 4;
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

Попробуем это прочитать:

Тип `mtype` содержит значения `A`, `B`, `C`, `D`, `E`, `F`

PROMELA: объявления переменных

Объявления переменных и полей выглядят так же, как и в C++, единственное отличие: объявление типа должно быть отделено от объявления переменной

Пример

```
bool a;
unsigned b : 3, c : 4;
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

Попробуем это прочитать:

Имеется тип `T` с единственным полем — одномерным булевым массивом `a` из пяти элементов

PROMELA: объявления переменных

Объявления переменных и полей выглядят так же, как и в C++, единственное отличие: объявление типа должно быть отделено от объявления переменной

Пример

```
bool a;
unsigned b : 3, c : 4;
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

Попробуем это прочитать:

Имеется переменная a булевого типа

PROMELA: объявления переменных

Объявления переменных и полей выглядят так же, как и в C++, единственное отличие: объявление типа должно быть отделено от объявления переменной

Пример

```
bool a;
unsigned b : 3, c : 4;
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

Попробуем это прочитать:

Имеются переменные b и c: беззнаковое целые числа ширины 3 и 4 соответственно

PROMELA: объявления переменных

Объявления переменных и полей выглядят так же, как и в C++, единственное отличие: объявление типа должно быть отделено от объявления переменной

Пример

```
bool a;
unsigned b : 3, c : 4;
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

Попробуем это прочитать:

Имеется массив `twodim` из трёх элементов типа `T`

Так можно объявлять многомерные массивы

PROMELA: объявления переменных

Объявления переменных и полей выглядят так же, как и в C++, единственное отличие: объявление типа должно быть отделено от объявления переменной

Пример

```
bool a;
unsigned b : 3, c : 4;
mtype = {A, B, C};
typedef T {
  bool a[5];
};
mtype = {D, E, F};
T twodim[3];
mtype symvar;
```

Попробуем это прочитать:

Имеется переменная `symvar` типа `mtype`

PROMELA: инициализация переменных

Каждая переменная имеет однозначно определённое начальное значение:

- все переменные и поля типов `bit`, `bool`, `short`, `int`, `unsigned` инициализируются значением `0`
- переменные типа `mtype` также инициализируются значением `0`, не совпадающим ни с одним из значений типа `mtype`

Так же, как и для переменных в C++ (*как минимум стандарта 11*), можно явно задавать начальные значения переменных и полей:

```
unsigned a : 3 = 4;
bool b[3] = {true, false};
typedef T {
  int a = 6;
};
```

PROMELA: типы процессов

Тип процесса — это нечто среднее между *описанием функции* и *описанием класса*:

```
proctype <имя типа процесса>(<аргументы>) {
  <тело процесса>
}
```

Более точно, это описание устройства **последовательно** работающей сущности, преобразующей в числе прочего *глобальные переменные*

Во время выполнения системы могут **параллельно** запускаться процессы — экземпляры типа процесса (*объекты класса*)

Самый простой способ запуска процесса выглядит так:

```
active proctype P() {
  ...
}
```

В этом случае в начале работы системы запускается один процесс типа `P`

PROMELA: тело процесса: присваивание

<тело процесса> ::= <последовательность>

<последовательность> ::=
    <инструкция> [; <инструкция>]* [;]

В теле процесса можно писать много разных инструкций

Будем их описывать постепенно, от простых к сложным

Присваивания похожи по написанию и значению на то, что пишется в C++:

<имя переменной> = <выражение>

<имя переменной>++

<имя переменной>--

В выражениях могут использоваться многие операции, имеющие те же значения и приоритеты, что и в C++, например: +, -, *, /, %, <<, >>, <, <=, >, >=, ==, !=, !, ~, &, ^, |, &&, ||, ->: (*это аналог ?:*), индексирование ([]), обращение к полям структур (.), константы

Арифметические операции работают **с переполнением**

PROMELA: пример

Теперь можно привести какой-нибудь простой полноценно работающий пример, чтобы хоть что-нибудь стало понятно:

```
bool a;
active proctype P() {
  a = true;
}
```

Как работает эта система:

- в начале работы переменная `a` имеет значение `false`, запущен один процесс типа `P`, и процесс готовится выполнить единственную инструкцию своего тела
- на следующем шаге работы переменная `a` имеет значение `true`, и запущенный процесс завершил своё выполнение

А что произойдёт с системой дальше?

Этот вопрос разъяснится позже

PROMELA: метки состояний

Как и в C++, в PROMELA можно помечать состояния управления:

<имя метки>: *<инструкция>*

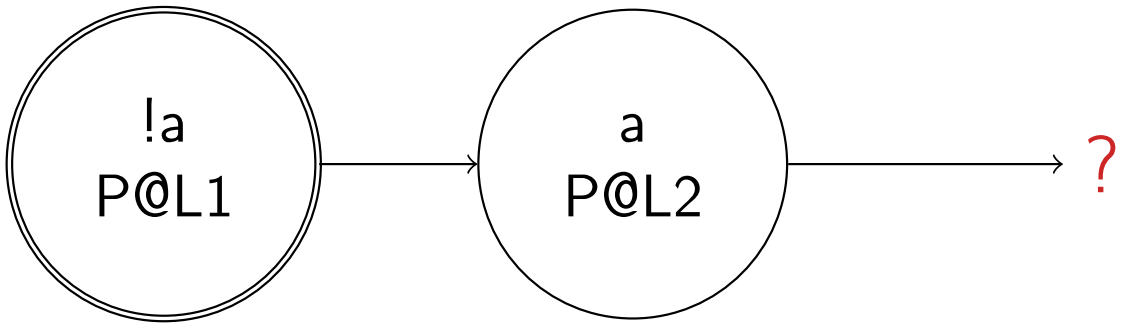
Будем использовать запись `P@L` для обозначения факта "процесс типа `P` готовится выполнить инструкцию с меткой `L`"

Забегая немного вперёд: запись `P@L` можно будет писать в спецификациях

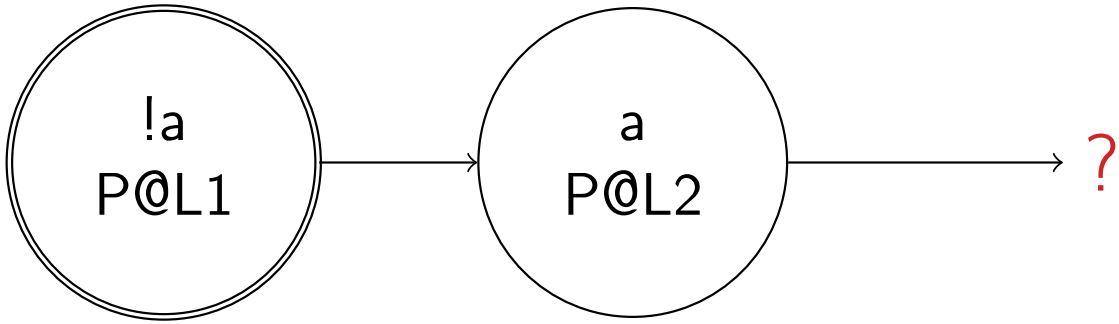
PROMELA: пример

```
bool a;
active proctype P() {
  L1: a = true;
  L2:
}
```

Теперь можно строго описать семантику написанной модели: состояния и переходы модели Крипке (*или нечто похожее на неё*)



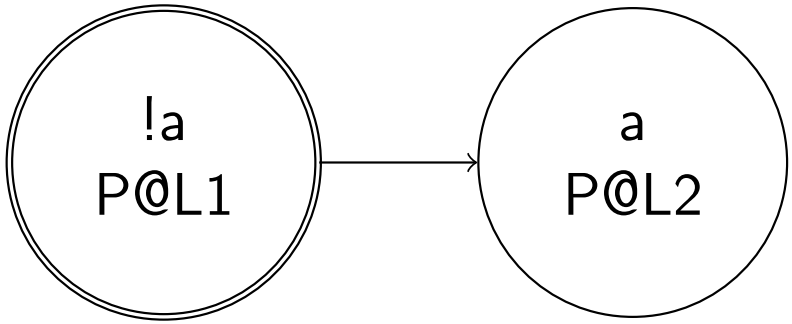
SPIN: безопасность и живость



В SPIN имеется (*как минимум*) два режима верификации:

- верификация свойств безопасности
- верификация свойств живости

SPIN: безопасность и живость



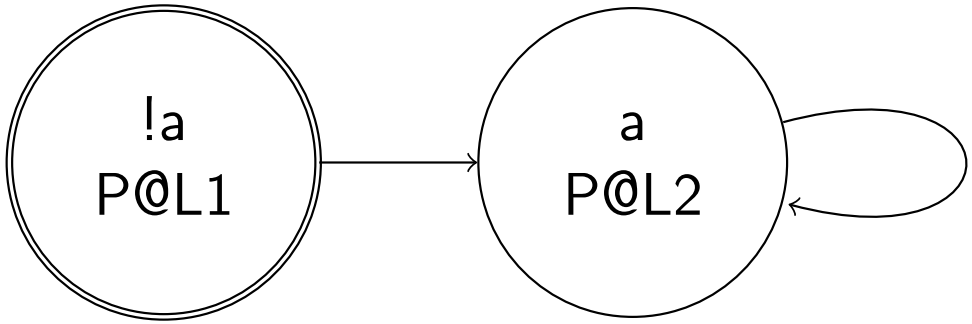
В SPIN имеется (*как минимум*) два режима верификации:

- верификация свойств безопасности
- верификация свойств живости

Свойство безопасности: *достижимо ли "плохое" состояние?*

В этом режиме в модели могут быть состояния блокировки, из которых не исходит ни одной дуги, и проверяется достижимость плохого состояния в имеющейся системе — *не совсем* модели Крипке

SPIN: безопасность и живость



В SPIN имеется (*как минимум*) два режима верификации:

- верификация свойств безопасности
- верификация свойств живости

Свойство живости: *верно ли, что как бы ни работала система, всегда можно получить "хорошую" бесконечную трассу?*

В этом режиме в каждое состояние блокировки добавляется петля

PROMELA: бесконечный цикл

Процесс, работающий бесконечно долго, как правило имеет в теле **бесконечный цикл**:

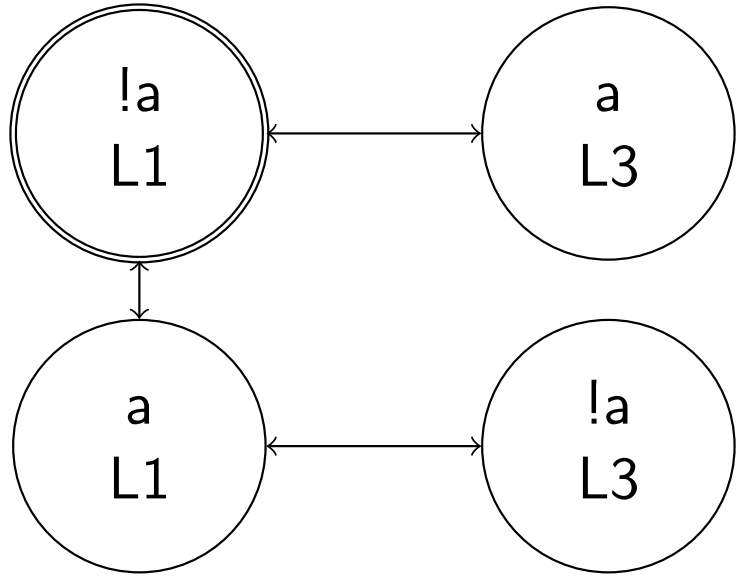
```
do
:: <последовательность>
...
:: <последовательность>
od
```

Шаг работы цикла выглядит так:

1. если цикл собирается выполниться, то недетерминированно выбирается одна из последовательностей, *в которой может быть выполнена первая инструкция*, и эта инструкция выполняется
2. если на предыдущем шаге выполнилась непоследняя инструкция последовательности, то выполняется следующая по порядку инструкция
3. после выполнения последней инструкции происходит переход к пункту 1

PROMELA: пример

```
bool a;
active proctype P() {
  L1: do
  :: L2: a = !a; L3: a = !a;
  :: L4: a = !a;
  od
  L5:
}
```



PROMELA: бесконечный цикл

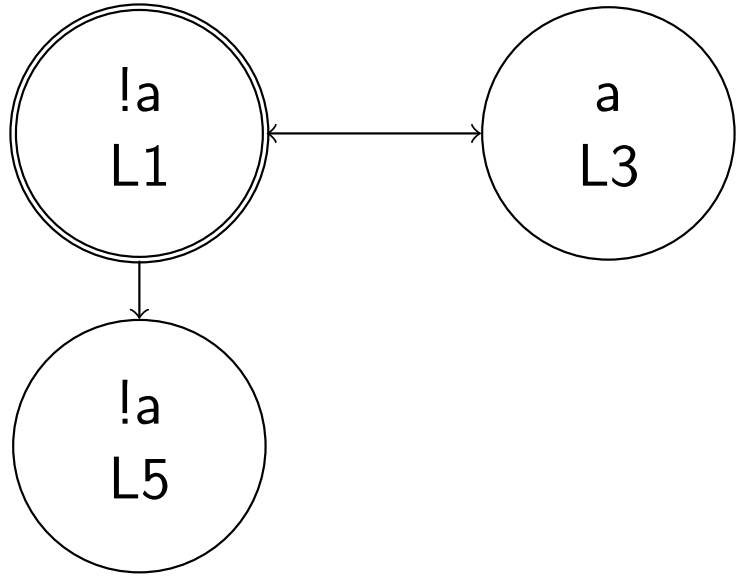
```
do
:: <последовательность>
...
:: <последовательность>
od
```

В выполняемой последовательности может встретиться инструкция `break`

Выполнение этой инструкции приводит к выходу из цикла и переходу к следующей после цикла инструкции

PROMELA: пример

```
bool a;
active proctype P() {
  L1: do
  :: L2: a = !a; L3: a = !a;
  :: L4: break;
  od
  L5:
}
```



PROMELA: оператор ветвления

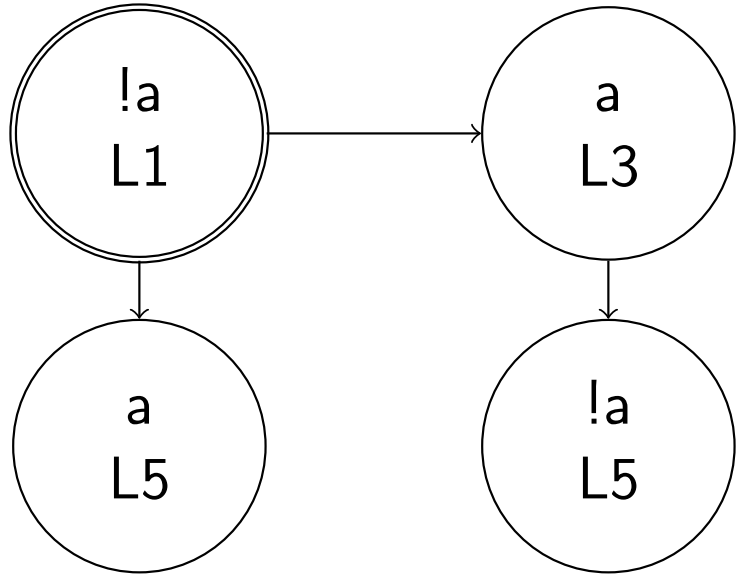
```
if
:: <последовательность>
...
:: <последовательность>
fi
```

Шаг работы оператора ветвления выглядит так:

1. если оператор собирается выполниться, то недетерминированно выбирается одна из последовательностей, *в которой может быть выполнена первая инструкция*, и эта инструкция выполняется
2. если на предыдущем шаге выполнилась непоследняя инструкция последовательности, то выполняется следующая по порядку инструкция
3. после выполнения последней инструкции происходит переход к следующей после условного оператора инструкции

PROMELA: пример

```
bool a;
active proctype P() {
  L1: if
  :: L2: a = !a; L3: a = !a;
  :: L4: a = !a;
  fi
  L5:
}
```



PROMELA: условная инструкция

Булево выражение может выступать в роли (*условной*) инструкции

Условная инструкция работает так:

- если значение инструкции — `true`, то происходит переход к следующей инструкции
- если значение инструкции — `false`, то эта инструкция **не может быть выполнена**

Язык PROMELA содержит особую условную инструкцию `else`:

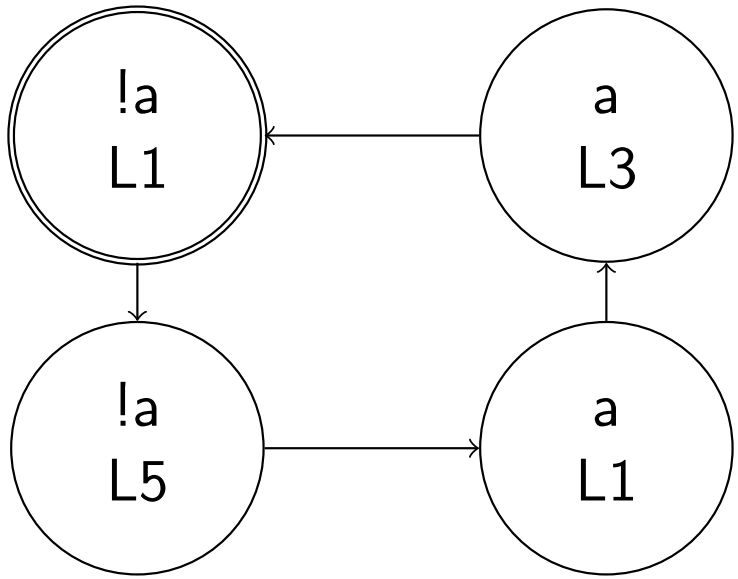
- она может быть только первой инструкцией последовательности в условном операторе и операторе цикла
- `else = true` тогда и только тогда, когда никакая другая инструкция не может быть выполнена

Язык PROMELA разрешает писать “`->`” вместо “`;`”

Правило хорошего тона — ставить `->` вместо `;` после условных инструкций

PROMELA: пример

```
bool a;
active proctype P() {
  L1: do
  :: L2: a -> L3: a = !a;
  :: L4: else -> L5: a = !a;
  od
}
```



PROMELA: композиция процессов

В общем случае в модели PROMELA в каждый момент времени может быть запущено более одного процесса

Запущенные процессы работают **асинхронно**, и шаг работы системы выглядит следующим образом:

- недетерминированно выбирается процесс, в котором может выполниться какая-либо инструкция
- этот процесс выполняет одну инструкцию

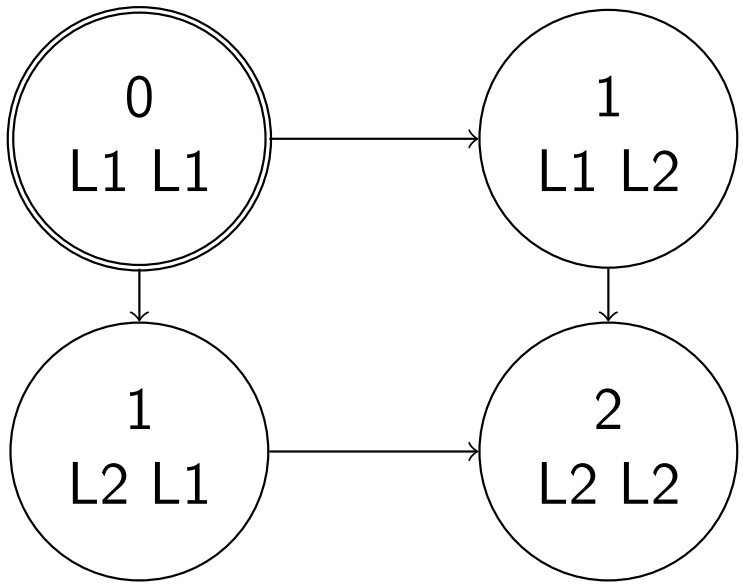
PROMELA: запуск процесса

Процесс может быть запущен двумя способами:

- `active [N] proctype ...`
  - `N` процессов запускаются в начале работы системы (часть `[N]` может быть опущена, и тогда запускается один процесс)
- `run` *<имя процесса>*`()`
  - это **инструкция**, по выполнении которой в систему добавляется один новый процесс заданного типа, находящийся в начальном состоянии управления

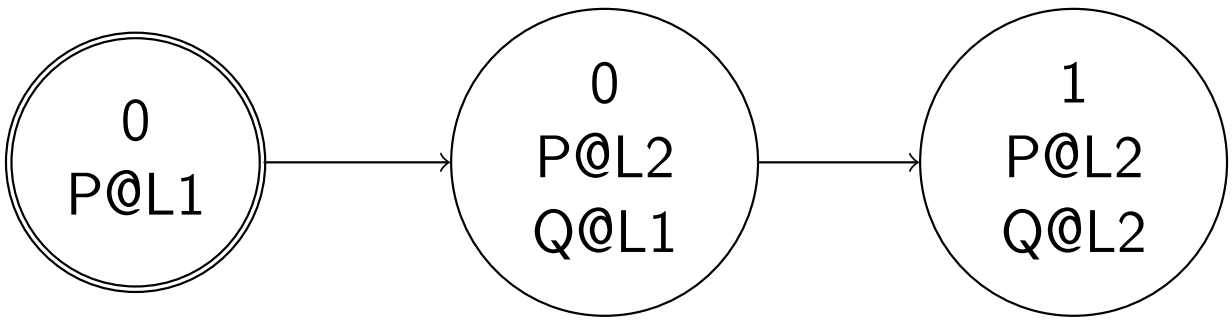
PROMELA: пример

```
int a;
active [2] proctype P() {
  L1: a++;
  L2:
}
```



PROMELA: пример

```
bool a;
active proctype P() {L1: run Q(); L2:}
proctype Q() {L1: a = !a; L2:}
```



PROMELA: локальные переменные

В теле процесса можно объявлять локальные переменные так же, как вне тела всех процессов объявляются **глобальные**

Локальная переменная может быть объявлена (*и инициализирована*) в любом месте внешней последовательности тела процесса

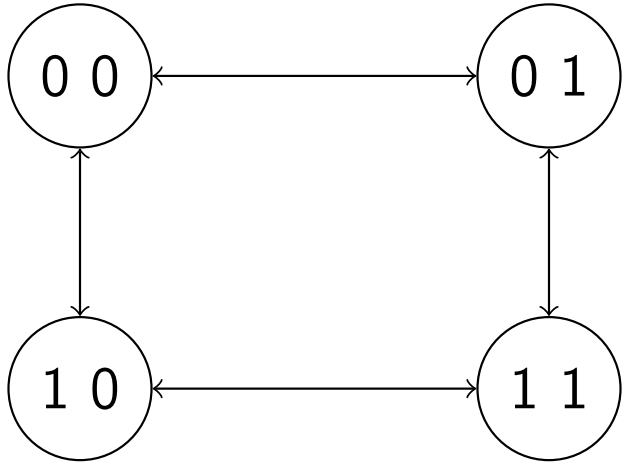
Объявление локальной переменной **не считается инструкцией**

Доступ к локальной переменной может быть получен из любой инструкции, принадлежащей идущим далее по тексту элементам последовательности

Инициализация локальной переменной происходит **при запуске процесса**, поэтому рекомендуется объявлять все локальные переменные **в начале** описания тела процесса

PROMELA: пример

```
active [2] proctype mosquito() {
  bool buzzing;
  do
  :: buzzing = !buzzing;
  od
}
```



PROMELA: каналы связи

В языке PROMELA есть ещё один тип переменных, не упомянутый ранее и никак не соотносящийся с типами в C++: каналы связи `chan`

Канал, объявленный среди глобальных переменных, как правило инициализируется:

`chan <`*имя*`> = [N] of {<`*тип*`>}`

Так заводится канал ёмкости `N`, по которому пересылаются сообщения заданного типа

Ёмкость канала — это максимальное количество сообщений, которые может хранить канал

PROMELA: посылка и приём сообщений

Инструкция посылки сообщения в канал имеет вид

<имя канала>!<выражение>

Инструкция приёма сообщения из канала имеет вид

<имя канала>?<имя переменной>

или

<имя канала>?<выражение>

В зависимости от ёмкости каналы разбиваются на два класса

Если ёмкость положительна, то канал асинхронный:

- инструкция посылки сообщения может быть выполнена тогда и только тогда, когда канал хранит сообщений меньше, чем его ёмкость
- инструкция приёма сообщения может быть выполнена тогда и только тогда, когда канал хранит хотя бы одно сообщение и (*в случае инструкции второго вида*) значение сообщения совпадает со значением выражения

PROMELA: посылка и приём сообщений

Инструкция посылки сообщения в канал имеет вид

<имя канала>!<выражение>

Инструкция приёма сообщения из канала имеет вид

<имя канала>?<имя переменной>

или

<имя канала>?<выражение>

В зависимости от ёмкости каналы разбиваются на два класса

Если ёмкость положительна, то канал асинхронный:

- по выполнении инструкции посылки сообщения в канал добавляется сообщение с тем же значением, что и выражение в инструкции
- по выполнении инструкции приёма первого вида в переменную записывается значение сообщения, посланного раньше остальных
- по выполнении инструкции приёма обоих видов прочитанное сообщение удаляется из канала

PROMELA: посылка и приём сообщений

Инструкция посылки сообщения в канал имеет вид

<имя канала>!<выражение>

Инструкция приёма сообщения из канала имеет вид

<имя канала>?<имя переменной>

или

<имя канала>?<выражение>

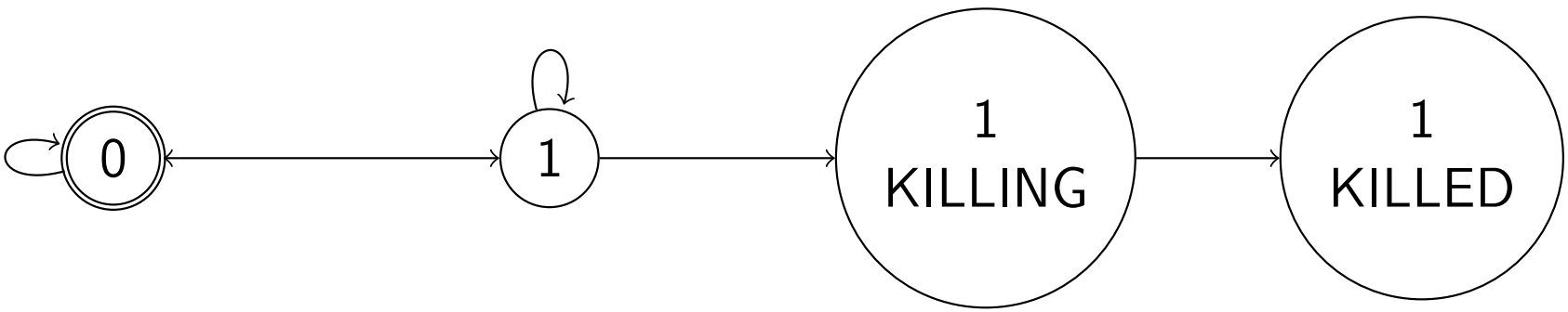
В зависимости от ёмкости каналы разбиваются на два класса

Если канал имеет ёмкость 0, то он синхронный:

- инструкции посылки и приёма сообщения выполняются **одновременно**, каждая ровно в одном процессе
- посылаемое значение напрямую передаётся принимающему процессу
- в остальном передача сообщения происходит так же, как для асинхронных каналов

PROMELA: пример

```
chan buzz = [0] of {bool}
active proctype mosquito() {
  do ::buzz!false; ::buzz!true; od
}
active proctype me() {
  bool found;
  do ::buzz?found; ::found->KILLING:break; od
  KILLED:
}
```



PROMELA: LTL-требования

`ltl <`*имя требования*`> {<`*тело требования*`>}`

Тело требования — это LTL-формула в таком синтаксисе:

- `[]` пишется вместо **G**
- `<>` пишется вместо **F**
- **U** используется как обычно
- можно использовать булевы выражения
- в качестве булевой переменной можно использовать запись `P@L`: хотя бы один экземпляр процесса типа `P` находится в состоянии управления с меткой `L`
- настоятельно не рекомендуется использовать **X**

PROMELA: пример

```
bool near, dead, hunted;
mtype = {ping}; chan c = [0] of {mtype};
active proctype mosquito() {   do
  :: !near -> near = true; c!ping;
  :: near && !hunted && !dead -> near = false;
  od }
active proctype bird() {   do
  :: c?ping -> hunted = true;
  :: hunted && near -> dead = true; hunted = false;
  :: hunted && !near -> hunted = false;
  od }
ltl inevitable_death {<>(near && <> dead)}
ltl cannot_fly_away {[](near -> <> dead)}
ltl reliable_hunt {[](hunting -> <> dead)}
```

Как выглядит модель Крипке для этой системы?

Что означают требования?

PROMELA: пример

```
bool near, dead, hunted;
mtype = {ping}; chan c = [0] of {mtype};
active proctype mosquito() {   do
  :: !near -> near = true; c!ping;
  :: near && !hunted && !dead -> near = false;
  od }
active proctype bird() {   do
  :: c?ping -> hunted = true;
  :: hunted && near -> dead = true; hunted = false;
  :: hunted && !near -> hunted = false;
  od }
ltl inevitable_death {<>(near && <> dead)}
ltl cannot_fly_away {[](near -> <> dead)}
ltl reliable_hunt {[](hunting -> <> dead)}
```

Обязательно ли включать в SPIN режим верификации свойств живости, или достаточно режима для свойств безопасности?

Какие из требований выполнены для системы?

PROMELA: неделимые последовательности инструкций

А что делать, если хочется выполнить последовательность инструкций за один шаг выполнения системы?

Например, в *протоколах доступа к критической секции* часто бывает важным уметь неделимо проверять и изменять значение переменной (*семафора*)

Неделимость последовательности действий моделируется в языке PROMELA так:

`atomic{<`*последовательность*`>}`

Как это работает: (*в простом случае*)

1. если сейчас выполнилась инструкция из блока `atomic` и на следующем шаге есть возможность выполнить инструкцию из этого же блока, то эта инструкция обязательно выполняется

PROMELA: неделимые последовательности инструкций

А что делать, если хочется выполнить последовательность инструкций за один шаг выполнения системы?

Например, в *протоколах доступа к критической секции* часто бывает важным уметь неделимо проверять и изменять значение переменной (*семафора*)

Неделимость последовательности действий моделируется в языке PROMELA так:

`atomic{<`*последовательность*`>}`

Как это работает: (*в простом случае*)

2 если сейчас выполнилась инструкция из блока `atomic`, но ни одна инструкция из этого блока больше не может выполниться, то выполнение системы происходит обычным образом

PROMELA: неделимые последовательности инструкций

А что делать, если хочется выполнить последовательность инструкций за один шаг выполнения системы?

Например, в *протоколах доступа к критической секции* часто бывает важным уметь неделимо проверять и изменять значение переменной (*семафора*)

Неделимость последовательности действий моделируется в языке PROMELA так:

`atomic{<`*последовательность*`>}`

Как это работает: (*в простом случае*)

3. если при выполнении обычным образом *вдруг* выполнилась ещё одна инструкция из блока `atomic`, то дальнейшее выполнение опять происходит согласно пунктам 1, 2

PROMELA: пример

```
bool near, dead, hunted;
mtype = {ping}; chan c = [0] of {ping};
active proctype mosquito() {   do
  :: !near -> atomic{near = true; c!ping;}
  :: near && !hunted && !dead -> near = false;
  od }
active proctype bird() {   do
  :: atomic{c?ping -> hunted = true;}
  :: hunted && near -> dead = true; hunted = false;
  :: hunted && !near -> hunted = false;
  od }
ltl inevitable_death {<>(near && <> dead)}
ltl cannot_fly_away {[](near -> <> dead)}
ltl reliable_hunt {[](hunting -> <> dead)}
```

А теперь как работает система, и какие требования для неё выполнены?

PROMELA: аргументы процесса

При описании типа процесса можно задавать список аргументов:

`proctype P(<`*`аргументы`*`>) ...`

В списке аргументов перечисляются переменные с явным указанием их типов (как *аргументы функции*) **без инициализации**

Переменные одного типа перечисляются через запятую с одним указанием типа; аргументы разных типов разделены точкой с запятой (как в обычном объявлении переменных)

При вызове процесса явно указываются значения всех аргументов (как при *вызове функции*)

В теле процесса аргументы используются так же, как и локальные переменные (как при *передаче параметров по значению*)

Для синхронизации процессов через значения переменных **аргументы процессов не подходят**: текущее значение аргумента видно **только** внутри процесса

PROMELA: пример

```
int a;
active proctype P() {run Q(1); run Q(2);}
proctype Q(int x) {a = x;}
```