Математические методы верификации схем и программ

mk.cs.msu.ru → Лекционные курсы
→ Математические методы верификации схем и программ

Блок O2

Обзор средства NuSMV

Лектор:
Подымов Владислав Васильевич
E-mail:
valdus@yandex.ru

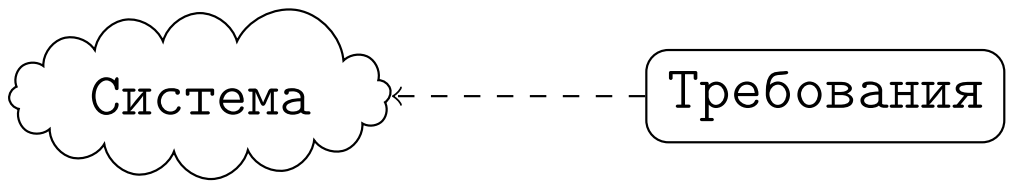
ВМК МГУ, 2022/2023, осенний семестр

Рассматриваемая ЗАДАЧА (напоминание)

Дано: неформальное описание

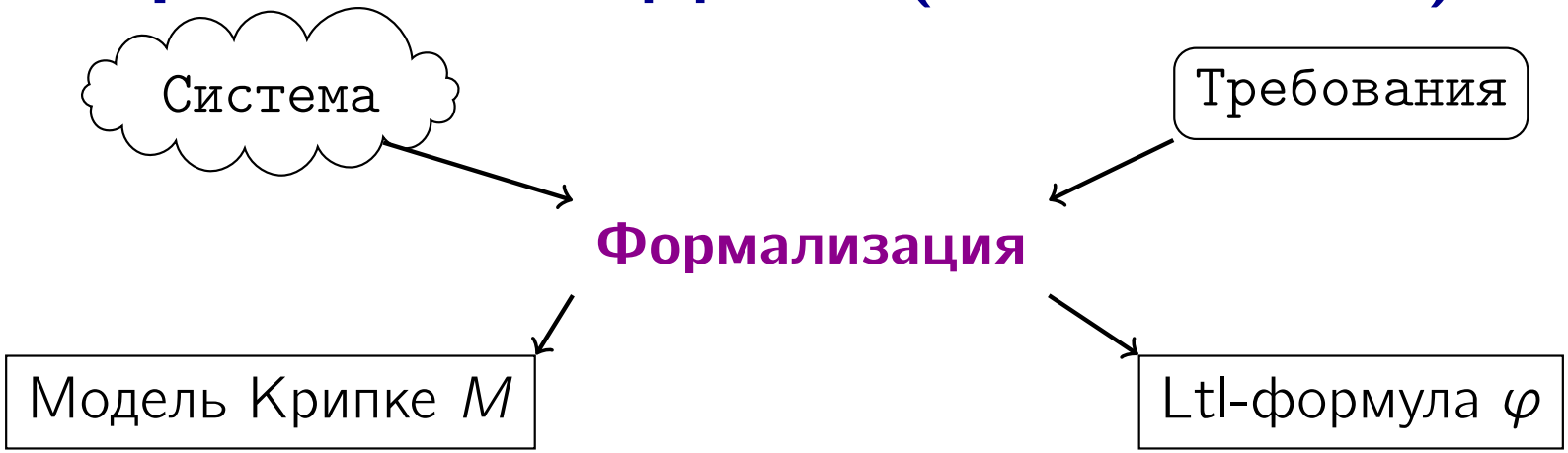
- системы и
- требований к ней

Требуется проверить, удовлетворяет ли система требованиям



Второе домашнее задание, как и первое, посвящено решению этой ЗАДАЧИ, но при помощи другого программного средства, использующего другой способ описания моделей и другой язык спецификаций

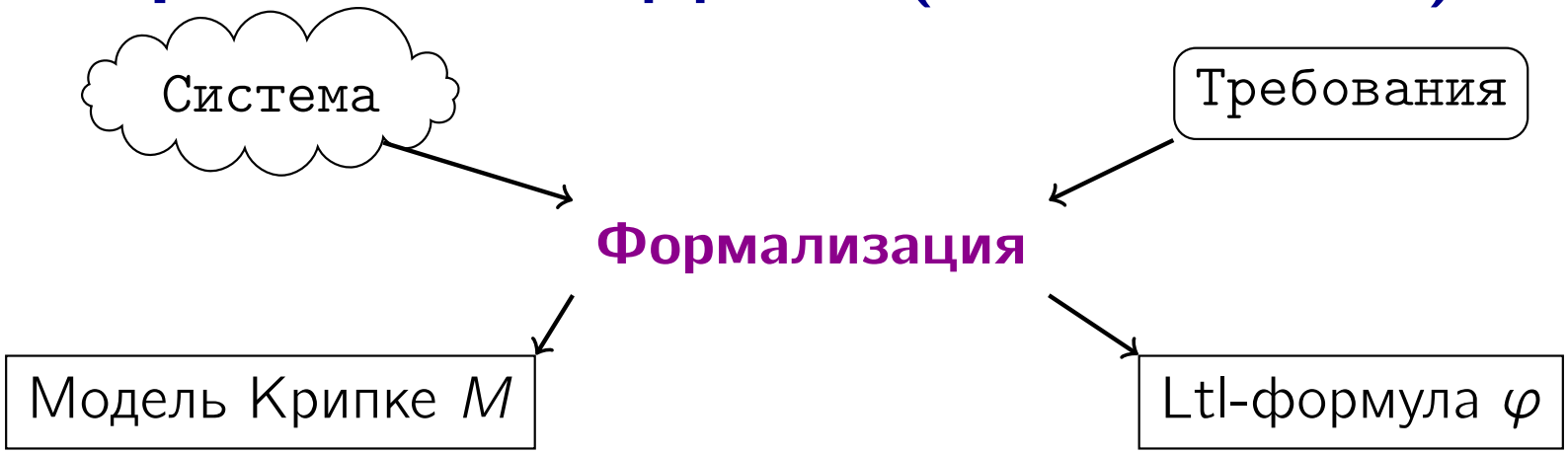
Рассматриваемая ЗАДАЧА (напоминание)



Средство NuSMV (для краткости — ν) будем обсуждать относительно

- моделей Крипке в символьном представлении как моделей рассматриваемых систем и
- CTL как языка формальных спецификаций

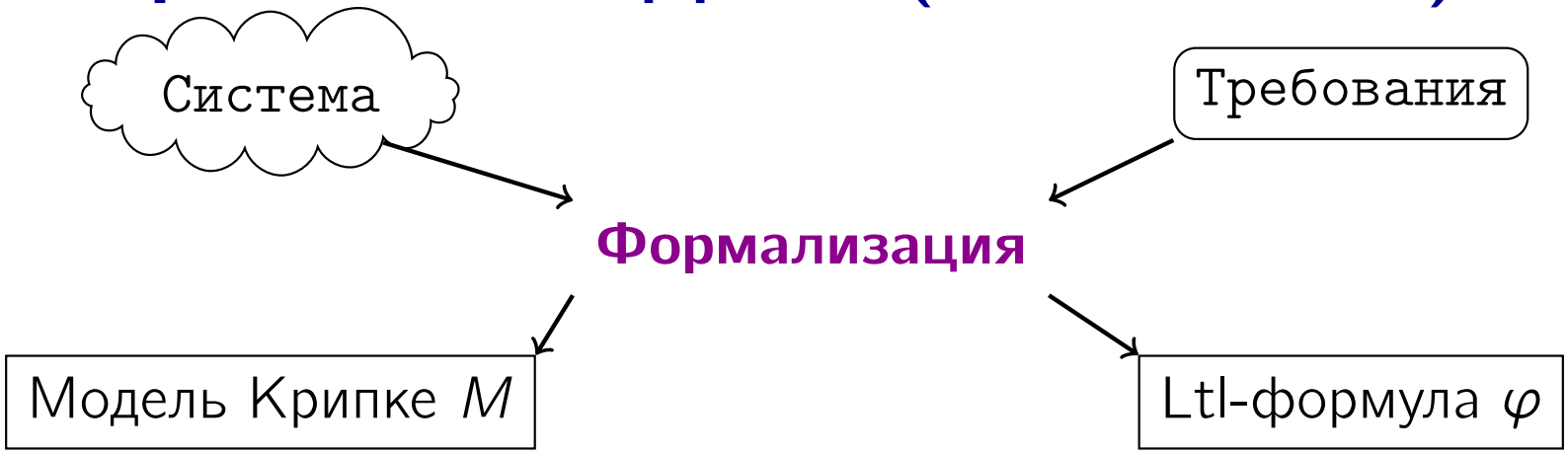
Рассматриваемая ЗАДАЧА (напоминание)



Основной трудностью по-прежнему будет этап **формализации**:

- Даются неформальные описания системы и требований
- Требуется придумать и реализовать модель и формальную спецификацию и убедить в их правильности сначала себя, и затем «заказчика» (*то есть меня*)

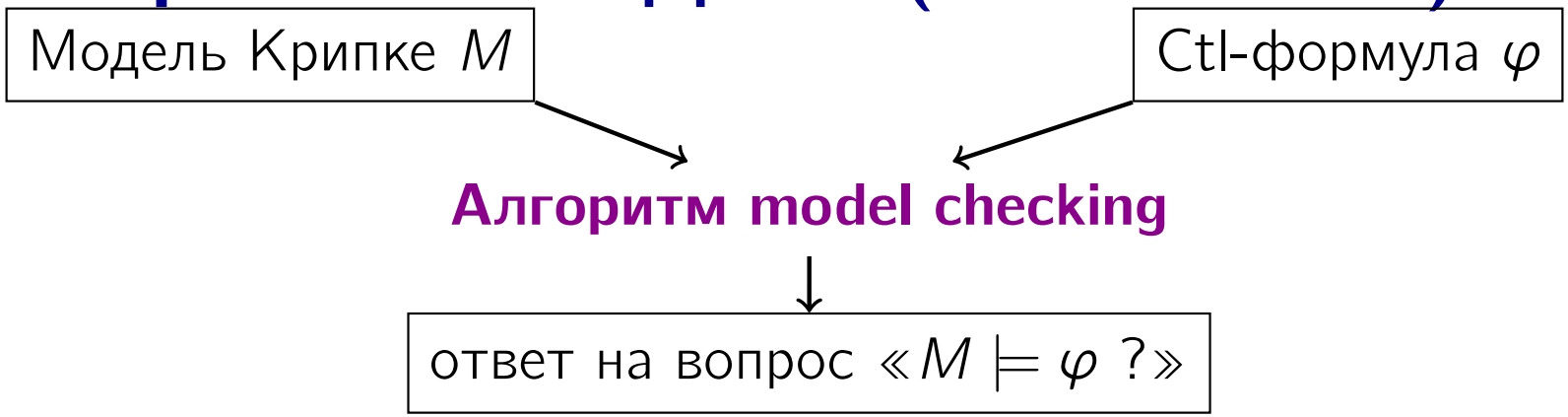
Рассматриваемая ЗАДАЧА (напоминание)



Дополнительная трудность по сравнению с первым обязательным заданием состоит в том, что

- язык `Promela` средства `Spin` достаточно близок к «привычным» императивным языкам и потому достаточно понятен даже тем, кто раньше не сталкивался с языками описания моделей, а
- язык ν основан на символьном представлении моделей, и придётся потрудиться в том числе и над тем, чтобы понять основные принципы разработки таких представлений

Рассматриваемая ЗАДАЧА (напоминание)



Алгоритмы верификации, используемые на практике в соответствующих программных средствах, как правило, так или иначе основаны на символьном алгоритме

В ν используется

- символьный алгоритм,
- основанный на BDD и на других символьных представлениях
- и снабжённый эвристиками и возможностью настройки деталей представлений для повышения эффективности

Рассматриваемая ЗАДАЧА (напоминание)

Сушествует немало программных средств для проверки выполнимости ctl-формул на моделях Крипке или родственных видах моделей:

ARC	BANDERA	CADENCE SMV	CWB-NC
Expander2	GEAR	LTS-min	MCMAS
NuSMV	ProB	TAPAs	...

Дисклеймер: *это просто несколько средств, выбранных как целенаправленно, так и наугад с соответствующей страницы в Википедии несколько лет назад*

Некоторые из этих средств (в том числе ν) способны проверять и выполнимость ltl-формул

Поробно остановимся на ν:

- У этого средства открытый исходный код, и его можно свободно использовать для академических целей
- Это средство (*новее и приятнее по сравнению со* `Spin` и) достаточно популярно
- Его язык хотя и труднее для понимания по сравнению с `Promela` (*тем, кто не очень хорошо знаком с символьными представлениями*), но всё же в целом достаточно прост

ν: Hello, World!

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

```
$ ls
helloworld.smv
$ NuSMV ./helloworld.smv
```

. . .

```
-- specification AG b  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    b = TRUE
  -> State: 1.2 <-
    b = FALSE
-- specification AG (!b -> AX b)  is true
$
```

ν: Модули

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Модуль — это описание модели Крипке и требований к ней

Описание модуля (*без параметров*) начинается со строки́

`MODULE` *<имя_модуля>*

Главный модуль — это модуль с именем main без параметров, и верификация производится для этого модуля

Идентификаторы, в том числе и для имён модулей, состоят из символов «A-Za-z0-9_$#-» и начинаются с «A-Za-z_»

$M[m]$ — Там будем обозначать модель Крипке, описанную в модуле m

ν: Переменные, состояния

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Состояние модели $M[m]$ *в простых случаях* — это оценка переменных, объявленных в m при помощи ключевого слова `VAR`:

`VAR` *<объявление>*`;` *<объявление>*`;` `...` *<объявление>*`;`

<объявление> `::=` *<идентификатор>* `:` *<тип>*

`boolean` — это тип с доменом $\{TRUE, FALSE\}$

Пример: в модели $M[main]$ для модуля выше содержится два состояния:

b/*FALSE*

b/*TRUE*

ν: Начальные состояния

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Начальные состояния модели $M[m]$ — это все состояния, для которых истинна **каждая** формула, записанная в m после ключевого слова `INIT`

Формула (или, по-другому, простое выражение) — это *булево выражение* над переменными модуля

Пример

Если из модуля выше удалить строку 3, то в $M[main]$ все состояния будут начальными:

Если не удалять эту строку, то в $M[main]$ будет ровно одно начальное состояние:

ν: Переходы

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Для описания переходов $M[m]$ используется два комплекта переменных:

1. переменные модуля m
   - эти переменные отвечают значениям в **текущем** состоянии, в начале перехода
2. переменные вида `next(x)`,
   где `x` — переменная модуля m
   - эти переменные отвечают значениям в **следующем** состоянии, в конце перехода
   - будем их называть next-переменными

ν: Переходы

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Next-выражение — это выражение над переменными и next-переменными

Next-формула — это булево next-выражение

Переход содержится в $M[m]$ $\Leftrightarrow$
каждая next-формула, записанная после ключевого слова `TRANS`, истинна на значениях переменных в начале дуги (первый комплект, без `next`) и в конце дуги (второй комплект, с `next`)

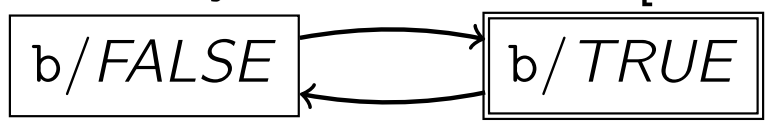
ν: Переходы

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Пример

Если из модуля выше удалить строку 4, то $M[main]$ будет устроена так:

Если оставить строку 4 в модуле выше, то $M[main]$ устроена так:



ν: Требования (CTL)

```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Спецификация модели $M[m]$ *обычно* записывается непосредственно в модуле m

Подробно рассмотрим только ctl-спецификации

Они записываются после ключевого слова `CTLSPEC`

БНФ для ctl-формул языка ν (Φ):

Φ ::= *формула* | (Φ) | !Φ | Φ&Φ | Φ|Φ |
Φ xor Φ | Φ xnor Φ | Φ -> Φ | Φ <-> Φ |
AXΦ | **EX**Φ | **AG**Φ | **EG**Φ | **AF**Φ | **EF**Φ |
A[Φ**U**Φ] | **E**[Φ**U**Φ]

(Полагаю, что синтаксис ясен и не требует дополнительных пояснений)

ν: Типы данных

- Булев тип `boolean`: домен $\{TRUE, FALSE\}$
- Перечисление, оно же множество $\{val_1, \ldots, val_k\}$, где val_i — либо целое число, либо идентификатор: домен — это собственно это множество
- Интервал `i..j`, где `i` и `j` — константные выражения: домен — множество целых чисел от `i` до `j`, включая границы
- Целые числа с двоичной записью заданной ширины `i`:
  - беззнаковые: `unsigned word [i]`
  - знаковые: `signed word [i]`
- Массив `array i..j of T` — это набор переменных типа `T`, проиндексированных числами из `i..j`
  - Можно объявлять и вложенные массивы — например, `array 0..2 of array 3..7 of boolean`

ν: Константы

- Булевы константы: `TRUE`, `FALSE`
- Целочисленные константы: 0, 1, 2, . . .
  (*такие константы можно использовать не везде, читайте документацию*)
- Символьные константы: идентификаторы в перечислениях
- Слова́, то есть целочисленные константы заданной ширины в заданной системе счисления, двоичной (b), восьмеричной (o), десятичной (d), или шестнадцатеричной (h) — например:
  - `0ub5_10011` и `0b_10011` — это число 19 в 5 битах
  - `0so_77` — это число -1 в 6 битах

ν: Выражения

Выражения строятся обычным образом над

- константами, объявленными переменными, скобками
- next-переменными (для next-выражений)
- булевыми операциями: `!`, `&`, `|`, `xor`, `xnor`, `->`, `<->`
- арифметическими операциями: `+`, `-`, `*`, `/`, `mod`, `abs()`, `max(,)`, `min(,)`
- арифметическими отношениями: `<`, `<=`, `>`, `>=`, `=`, `!=`
- побитовыми операциями: `<<`, `>>`, `::` (*concatenation*)
- операциями индексирования: `[i]` (*индексирование массивов*), `[i:j]` (*индексирование слов*)

ν: Выражения

Выражения строятся обычным образом над

- теоретико-множественными операциями: $\{e_1, \ldots, e_k\}$, `union`, `in`, `e1..e2`
- тернарным оператором: `?:`
- оператором выбора:

  `case` *<альтернатива>*`;` `...` *<альтернатива>* `esac`

  - *<альтернатива>* `::=` *<формула>* `:` *<выражение>*
  - выбирается первая по списку альтернатива, *<формула>* которой имеет значение `TRUE`
  - результат выражения case — это результат *<выражения>* выбранной альтернативы
- ...

В языке используется статическая типизация с некоторым спектром неявных преобразований (*см. документацию*)

ν: Композиция модулей

```
1  MODULE main
2  VAR
3    b : boolean;
4    m : aux;
5  INIT b;
6  TRANS next(b) = m.b;
7  CTLSPEC AG (b != m.b); -- неправда
8  CTLSPEC AX AG (b != m.b); -- правда
9
10 MODULE aux
11 VAR b : boolean;
12 TRANS next(b) = !b;
```

В ν можно описывать системы, состоящие из нескольких модулей

Имя модуля может использоваться в качестве типа переменной

Переменная такого типа — это экземпляр модуля, отвечающий копии соответствующей модели, выполняющейся согласно синхронной композиции: выполнение перехода в композиции — это одновременное выполнение переходов всех участников композиции

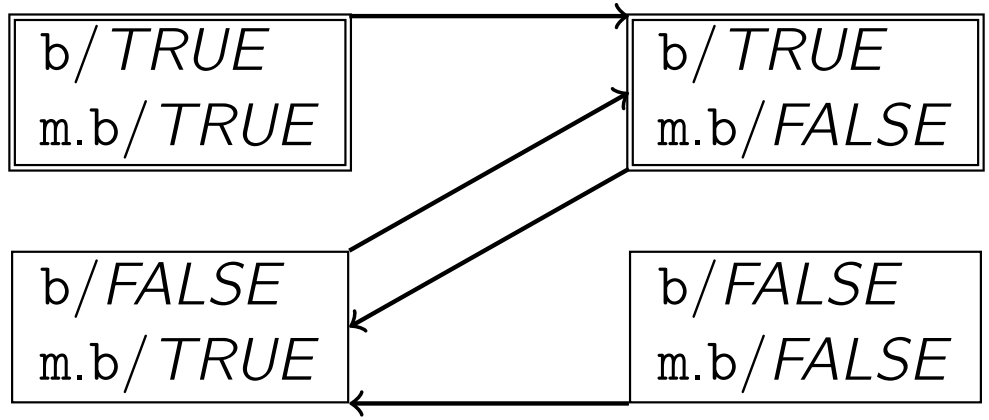
Доступ к локальным переменным экземпляра устроен так же, как доступ к структурам полей в C/C++

ν: Композиция модулей

```
1  MODULE main
2  VAR
3    b : boolean;
4    m : aux;
5  INIT b;
6  TRANS next(b) = m.b;
7  CTLSPEC AG (b != m.b); -- неправда
8  CTLSPEC AX AG (b != m.b); -- правда
9
10 MODULE aux
11 VAR b : boolean;
12 TRANS next(b) = !b;
```

Пример

Модель $M[main]$ устроена так:



ν: Макроопределения

```
1 MODULE main
2 VAR
3   b : boolean;
4   c : boolean;
5 DEFINE x := b xor c;
6 INIT x;
7 TRANS x -> next(b);
8 TRANS next(c) = x;
```

В ν можно задавать макроопределения:

DEFINE *<идентификатор>* := *<выражение>*;

Объявленный так *<идентификатор>* можно использовать так же, как и переменные, и считается, что в каждом месте, где записан *<идентификатор>*, вместо него подставлено *<выражение>* (обёрнутое в скобки)

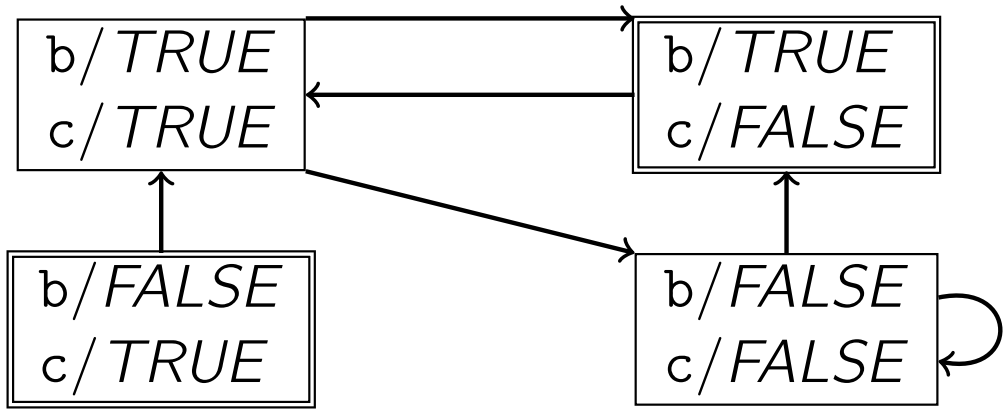
<Идентификаторы> макроопределений не учитываются в пространстве состояний модели и используются только как сокращения для *<выражений>*

ν: Макроопределения

```
1 MODULE main
2 VAR
3   b : boolean;
4   c : boolean;
5 DEFINE x := b xor c;
6 INIT x;
7 TRANS x -> next(b);
8 TRANS next(c) = x;
```

Пример

Модель $M[main]$ устроена так:



ν: Параметры модулей

```
1  MODULE main
2  VAR n : neg(n.b);
3      s : sum(n.b, s.b);
4  CTLSPEC AG (n.b -> AX !n.b); -- правда
5  CTLSPEC AG (!n.b -> AX n.b); -- правда
6  CTLSPEC AG (n.b xor s.b -> AX s.b); -- правда
7  CTLSPEC AG (n.b xnor s.b -> AX !s.b); -- правда
8
9  MODULE neg(x)
10 VAR b : boolean;
11 TRANS next(b) = !x;
12
13 MODULE sum(x,y)
14 VAR b : boolean;
15 TRANS next(b) = x xor y;
```

Параметры модуля — это идентификаторы, перечисляющиеся в скобках через запятую после имени модуля

Параметр расценивается *практически* как макроопределение:

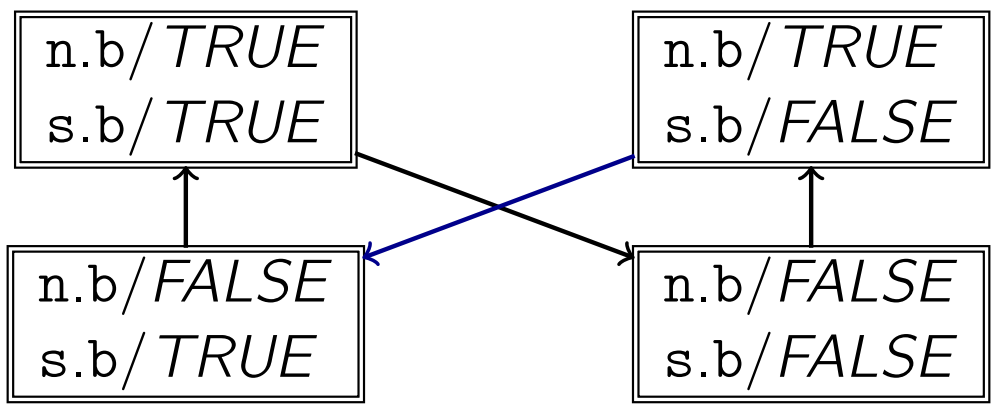
- Имя параметра возле имени модуля — это имя макроопределения
- Next-выражение в соответствующем месте в объявлении экземпляра модуля — это выражение макроопределения

ν: Параметры модулей

```
1  MODULE main
2  VAR n : neg(n.b);
3      s : sum(n.b, s.b);
4  CTLSPEC AG (n.b -> AX !n.b); -- правда
5  CTLSPEC AG (!n.b -> AX n.b); -- правда
6  CTLSPEC AG (n.b xor s.b -> AX s.b); -- правда
7  CTLSPEC AG (n.b xnor s.b -> AX !s.b); -- правда
8
9  MODULE neg(x)
10 VAR b : boolean;
11 TRANS next(b) = !x;
12
13 MODULE sum(x,y)
14 VAR b : boolean;
15 TRANS next(b) = x xor y;
```

Пример

Модель $M[main]$ устроена так:



ν: Инварианты состояний

```
1 MODULE main
2 VAR b : boolean;
3 INVAR b;
4 CTLSPEC EF !b; -- неправда
```

Иногда бывает удобно описывать модель Крипке как подграф заданного графа, порождённый некоторым множеством вершин

Это множество вершин записывается:

INVAR *<выражение>*;

Более точно, каждое такое *<выражение>* добавляется

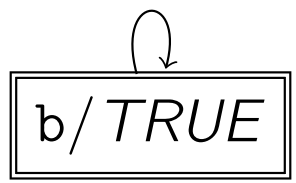
- ко всем другим выражениям системы, ограничивающим множество состояний, и
- для обоих комплектов переменных в выражениях, ограничивающих множество переходов

ν: Инварианты состояний

```
1 MODULE main
2 VAR b : boolean;
3 INVAR b;
4 CTLSPEC EF !b; -- неправда
```

Пример

Модель $M[main]$ устроена так:



ν: Специальные переменные

Застывшая переменная объявляется так же, как и «обычная», но с ключевым словом `FROZENVAR` вместо `VAR`

Значение застывшей переменной задаётся для начального состояния (т.е. под ключевым словом `INIT`) и не изменяется при выполнении переходов

Подробнее:

- Застывшая переменная учитывается в множестве состояний
- Использование next-переменной, отвечающее застывшей переменной, запрещено
- В модель неявно включается ограничение `TRANS next(x) = x;` для каждой застывшей переменной `x`

ν: Специальные переменные

Входная переменная объявляется так же, как и «обычная», но с ключевым словом `IVAR` вместо `VAR`

Значениями входной переменной размечаются не состояния модели, а её переходы

Подробнее:

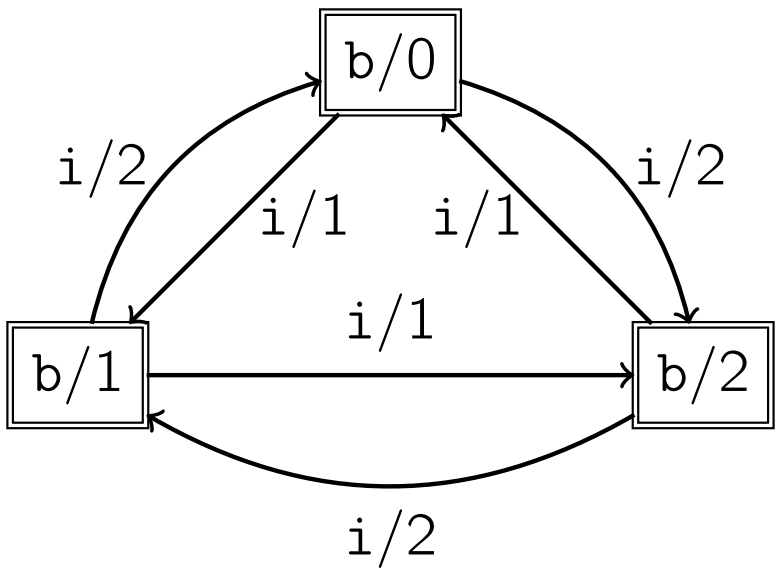
- Входные переменные не учитываются в множестве состояний
- Запрещено использовать next-переменные, отвечающие входным переменным, и использовать входные переменные вне выражений, задающих переходы системы
- Входные переменные предназначены для использования в выражениях под словом `TRANS` наряду с остальными переменными

ν: Специальные переменные

```
1 MODULE main
2 VAR b : {0,1,2};
3 IVAR i : {1,2};
4 TRANS next(b) = (b + i) mod 3;
5 CTLSPEC AG (b = 0 -> EX b = 1); -- правда
6 CTLSPEC AG (b = 0 -> EX b = 2); -- правда
7 CTLSPEC AG (b = 0 -> AX b != 0); -- правда
```

Пример

Модель $M[main]$ устроена так:



ν: Асинхронная композиция

В `NuSMV 2.6.0` поддерживаются, хотя и считаются устаревшими, средства асинхронной композиции модулей

Здесь эти средства не обсуждаются, и если хотите их использовать, то **внимательно** прочитайте документацию

Асинхронная композиция в ν может быть реализована как частный случай синхронной:

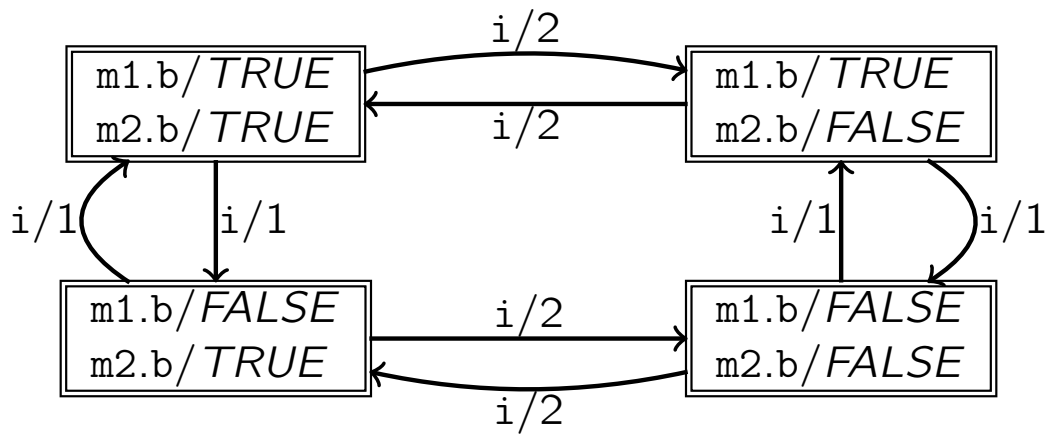
- В экземпляр модуля добавляется булев параметр «сейчас мой ход»
- Если «сейчас мой ход» имеет значение `TRUE`, то переменные модуля изменяются обычным образом, а иначе **явно** прописано, что они остаются неизменными
- Модуль, совмещающий экземпляры, может использоваться в качестве арбитра, распределяющего ходы

ν: Асинхронная композиция

```
1  MODULE main
2  IVAR turn : {1,2};
3  VAR m1 : aux(turn = 1);
4      m2 : aux(turn = 2);
5  CTLSPEC AG AF (!m1.b | !m2.b); -- правда
6  CTLSPEC AG AF !m1.b; -- неправда
7
8  MODULE aux(active)
9  VAR b : boolean;
10 ASSIGN next(b) := case
11   active : !b;
12   TRUE : b;
13 esac;
```

Пример

Модель $M[main]$ устроена так:



ν: Справедливость

```
1  MODULE main
2  IVAR turn : {1,2};
3  VAR m1 : aux(turn = 1);
4      m2 : aux(turn = 2);
5  CTLSPEC AG AF !m1.b; -- правда
6
7  MODULE aux(active)
8  VAR b : boolean;
9  ASSIGN next(b) := case
10   active : !b;
11   TRUE : b;
12 esac;
13 JUSTICE active;
```

`JUSTICE <формула>;` — это ограничение безусловной справделивости, означающее, что справедливыми считаются только те пути в модели, в которых *формула* становится истинной бесконечно часто

Если в модели содержится несколько ограничений справедливости, то считается, что путь справедлив, если он удовлетворяет **каждому** ограничению

ν: Справедливость

```
1  MODULE main
2  IVAR turn : {1,2};
3  VAR m1 : aux(turn = 1);
4      m2 : aux(turn = 2);
5  CTLSPEC AG AF !m1.b; -- правда
6
7  MODULE aux(active)
8  VAR b : boolean;
9  ASSIGN next(b) := case
10   active : !b;
11   TRUE : b;
12 esac;
13 JUSTICE active;
```

Пример

Модель $M[main]$ устроена так же, как и в предыдущем примере

Справедливыми считаются те и только те пути, в которых

- `turn` бесконечно часто принимает значение 1 и
- `turn` бесконечно часто принимает значение 2

Иными словами, несправедливый путь — это такой, в котором один из экземпляров `m1`, `m2` с некоторого момента навсегда становится неактивным

ν: Нетотальные модели Крипке

Согласно лекциям, модель Крипке — это тотальный граф: из каждого состояния должен исходить хотя бы один переход

В некоторых средствах (в том числе ν) допускается записывать и Some verification tools (including NuSMV) allow descriptions of нетотальные модели

Обычно такие модели считаются **некорректными**, кроме тех случаев, когда в документации явно написано обратное

В частности, в ν нетотальные модели по умолчанию считаются некорректными, и те опции и алгоритмы, которые позволяют сделать иначе, здесь не обсуждаются и не рекомендуются к использованию

Следить за тотальностью модели — задача разработчика модели

ν: Нетотальные модели Крипке

```
1 MODULE main
2 VAR b : boolean;
3 INVAR b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC !AG b;
```

```
$ NuSMV nontotal.smv
```

. . .

```
********   WARNING   ********
Fair states set of the finite state machine is empty.
This might make results of model checking not trustable.
******** END WARNING ********
-- specification AG b  is true

********   WARNING   ********
Fair states set of the finite state machine is empty.
This might make results of model checking not trustable.
******** END WARNING ********
-- specification !(AG b)  is true
```

Модель $M[main]$ нетотальна, и обычно ν для такой модели явно предупреждает, что результатам верификации доверять нельзя («... model checking not trustable»)