

# Математические методы верификации схем и программ

Лекторы:

Захаров Владимир Анатольевич

Подымов Владислав Васильевич

е-mail рассказчика:

**valdus@yandex.ru**

Осень 2017

# Семинар 4

NuSMV  
(обзор средства)

# Задача model checking

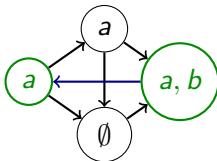
## Широкая формальная постановка

Даны

- ▶ модель Крипке  $M = (S, S_0, R, L)$
- ▶ темпоральная формула  $\varphi$

Требуется вычислить множество состояний

$$S_{\varphi, M} = \{s \mid s \in S, M, s \models \varphi\}$$



# Задача model checking

## Узкая формальная постановка

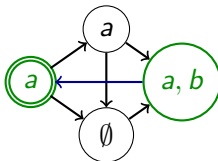
Даны

- ▶ модель Крипке  $M = (S, S_0, R, L)$
- ▶ CTL-формула  $\varphi$

Требуется проверить выполнимость формулы  $\varphi$  в модели  $M$ :

$$S_0 \stackrel{?}{\subseteq} S_{\varphi, M}, \text{ или}$$

$$M \stackrel{?}{\models} \varphi$$



# Задача model checking

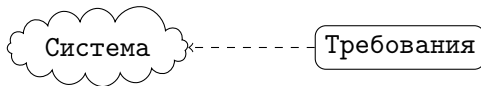
## Содержательная постановка

*Даны*

- ▶ неформальное описание системы
- ▶ содержательное описание требований к системе

*Требуется проверить,*

удовлетворяет ли система набору требований



Программа-максимум — научиться решать такую содержательно поставленную **ЗАДАЧУ**

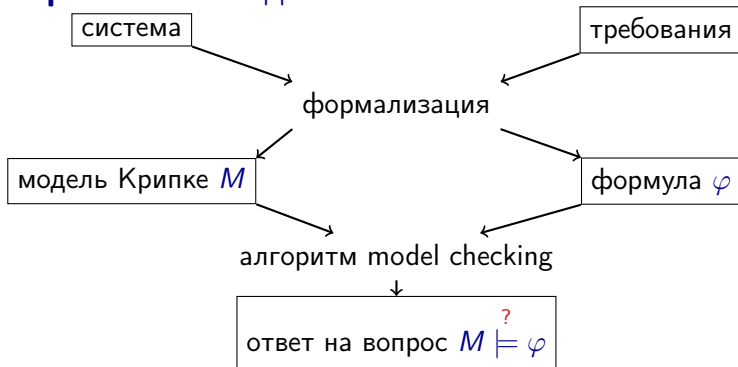
# Задача model checking

На всех оставшихся семинарах **ЗАДАЧА** будет решаться при помощи программных средств верификации

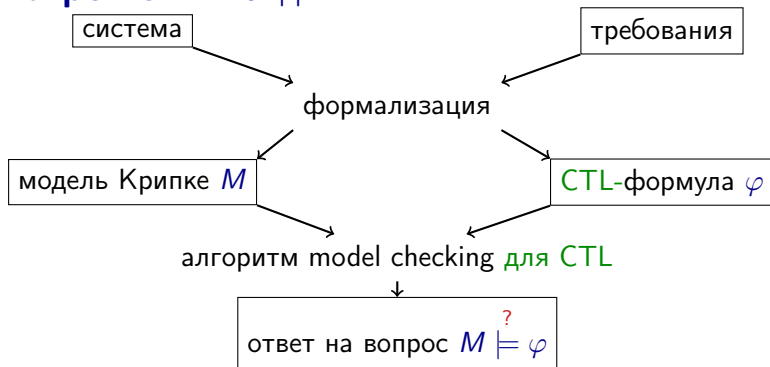
Каждое из этих средств принимает на вход систему и требования, описанные на специальном языке,

- ▶ достаточно понятном, чтобы можно было легко описывать большие системы и сложные требования к ним
- ▶ достаточно строгом, чтобы можно было легко переформулировать это описание как задачу model checking в узкой формальной постановке (проверить, выполняется ли формула на модели Крипке)

# Схема решения ЗАДАЧИ



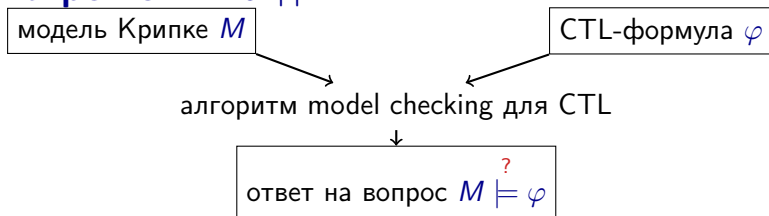
# Схема решения ЗАДАЧИ



На ближайших семинарах будет рассматриваться логика ветвящегося времени



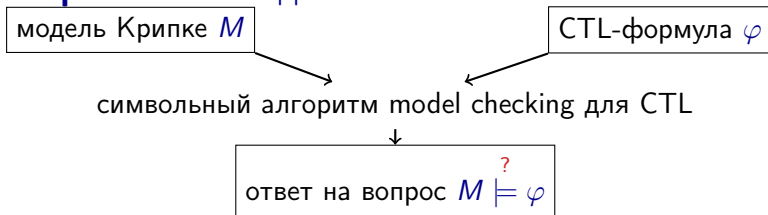
# Схема решения ЗАДАЧИ



Вам известно (как минимум) два алгоритма проверки CTL-формул:

- ▶ **табличный алгоритм**
  - ▶ наглядный и понятный
  - ▶ лежит в основе всех других алгоритмов
  - ▶ крайне неэффективный
- ▶ **символьный алгоритм**
  - ▶ ненаглядный и не очень понятный
  - ▶ намного эффективнее табличного (?)

# Схема решения ЗАДАЧИ



Эффективность символьного алгоритма зависит от эффективности средств работы с булевыми функциями. Таких средств в программистском арсенале огромное множество, и обычно такое средство — это одно из двух:

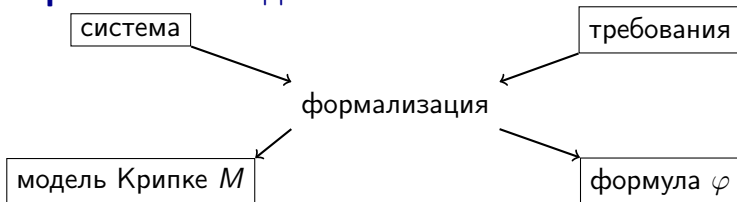
1. библиотека для работы с BDD
2. средство проверки выполнимости булевых и иных формул: SAT/SMT-решатель

Средство верификации, как правило, содержит полную реализацию алгоритма, так что

“применение алгоритма верификации” =

“применение средства верификации согласно инструкции”

# Схема решения ЗАДАЧИ



Как правило, система и требования формулируются на естественном или полужформальном языке

Формализация системы и требований как модели Крипке и формулы темпоральной логики (*или как описание на входном формальном языке средства верификации*) — это нетривиальный процесс, требующий немалой квалификации

Именно этому процессу будут посвящены все оставшиеся семинары

А может, никто так не делает,  
и вообще верификация CTL никому не нужна?

# Схема решения ЗАДАЧИ

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

*(на случай если захотите их использовать)*

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

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

В курсе сосредоточим внимание на средстве NuSMV:

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

# ( $\nu$ ) 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
$
```

## ( $\nu$ ) 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 предъявляемых к ней требований

Первая строка модуля с именем **name** (*без параметров*) выглядит так:

**MODULE** name

Допустимые **имена** в NuSMV состоят из символов “A-Za-z0-9\_ \$#-” и начинаются с “A-Za-z\_”

Для ясности записью **M[m]** будем обозначать модель Крипке, описываемую модулем **m**

**Главный модуль** называется **main** и не содержит параметров, и именно для него проверяются требования утилитой NuSMV

## ( $\nu$ ) Переменные

```
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  $M[md]$  (в простом случае), — это совокупность значений всех **переменных**, объявленных в  $md$  при помощи ключевого слова VAR:

*VAR объявление; объявление; ... объявление;*  
*объявление ::= имя : тип*

**boolean** — это тип с множеством значений  $\{TRUE, FALSE\}$

Модель  $M[main]$  имеет два состояния:

b/*FALSE*

b/*TRUE*

## ( $\nu$ ) 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  $M[md]$  — это все состояния, удовлетворяющие каждой формуле в описании  $md$ , записанной под ключевым словом **INIT**

**Формула** — это любое выражение (*simple expression*) типа **boolean** над переменными, объявленными в  $md$

Если бы в модуле  $main$  не было строки 3, то каждое состояние модели  $M[main]$  было бы начальным:

b/FALSE

b/TRUE

С учётом строки 3 модель  $M[main]$  содержит ровно одно начальное состояние:

b/FALSE

b/TRUE



## ( $\nu$ ) 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 множества переходов модели  $M[md]$  используется два комплекта переменных:

1. переменные, объявленные в  $md$ 
  - ▶ ими обозначаются значения переменных в **текущем** состоянии модели (**начальные значения**)
2. переменные вида  $next(x)$ ,  
где  $x$  — переменная, объявленная в  $md$ 
  - ▶ ими обозначаются значения переменных в **следующем** состоянии модели (**конечные значения**)
  - ▶ переменную  $next(x)$  будем называть **next-аналогом** переменной  $x$

## ( $\nu$ ) 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);
```

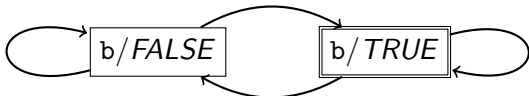
**Next-формула** — это *next-выражение* (*next-expression*) типа `boolean`, то есть выражение над переменными модуля и их `next`-аналогами

Переход содержится в модели  $M[md]$   $\Leftrightarrow$   
начальные и конечные значения перехода удовлетворяют  
каждой `next`-формуле, записанной под ключевым словом  
**TRANS**

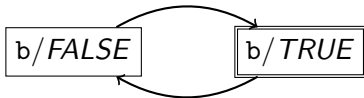
## ( $\nu$ ) Переходы

```
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);
```

Если бы в модуле `main` не было строки 4, то модель  $M[\text{main}]$  выглядела бы так:



С учётом строки 4 модель  $M[\text{main}]$  выглядит так:



## ( $\nu$ ) Спецификации

```
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);
```

Требования, предъявляемые к модели  $M[md]$ , записываются непосредственно в модуле  $md$

Нас будут интересовать требования, записанные в виде CTL-формул

Такие требования записываются под ключевым словом CTLSPEC

БНФ, определяющая синтаксис CTL-формул ( $\Phi$ ):

$$\begin{aligned} \Phi ::= & \text{формула} \mid (\Phi) \mid !\Phi \mid \Phi \ \& \ \Phi \mid \text{“}\Phi \mid \Phi\text{”} \mid \\ & \Phi \ \text{xor} \ \Phi \mid \Phi \ \text{xnor} \ \Phi \mid \Phi \rightarrow \Phi \mid \Phi \leftrightarrow \Phi \mid \\ & \mathbf{AX}\Phi \mid \mathbf{EX}\Phi \mid \mathbf{AG}\Phi \mid \mathbf{EG}\Phi \mid \mathbf{AF}\Phi \mid \mathbf{EF}\Phi \mid \\ & \mathbf{A}[\Phi \mathbf{U} \Phi] \mid \mathbf{E}[\Phi \mathbf{U} \Phi] \end{aligned}$$

(вроде бы синтаксис довольно естественный  
и пояснений не требует?)

## ( $\nu$ ) Типы данных

- ▶ **булев тип**: `boolean`, значения —  $\{TRUE, FALSE\}$
- ▶ **перечисление**, или **множество**:  $\{val_1, \dots, val_k\}$ , где  $val_i$  — число или имя
- ▶ **интервал**: `i..j` — это множество целых чисел от `i` до `j` включительно, где `i` и `j` — константные выражения
- ▶ **целые числа** с двоичной записью ширины `i`:
  - ▶ **беззнаковые**: `unsigned word [i]`
  - ▶ **знаковые**: `signed word [i]`
- ▶ **массивы**: `array i..j of T`, это набор переменных типа `T`, индексируемых от `i` до `j` включительно
  - ▶ в том числе вложенные массивы, например,  
`array 0..2 of array 3..7 of boolean`

## ( $\nu$ ) Константы

- ▶ константы типа `boolean`: `TRUE`, `FALSE`
- ▶ целочисленные константы: `0`, `1`, `2`, ...  
(их можно использовать не везде)
- ▶ символьные константы: имена, встречающиеся в перечислениях
- ▶ word-константы в одной из обычных систем счисления: двоичной (`b`), восьмеричной (`o`), десятичной (`d`), шестнадцатеричной (`h`) — записываются в особом формате:
  - ▶ `0ub5_10011` и `0b_10011` — число `19` в `5`-ти битах
  - ▶ `0so_77` — число `-1` в `6`-ти битах

## ( $\nu$ ) Выражения

Выражения строятся в условиях статической типизации с небольшими возможностями приведения типов (*о которых можно почитать в документации*) над

- ▶ константами, объявленными переменными, скобками
- ▶ *next*-аналогами объявленных переменных (*для next-выражений*)
- ▶ булевыми операциями: `!`, `&`, `|`, `xor`, `xnor`, `->`, `<->`
- ▶ арифметическими операциями: `+`, `-`, `*`, `/`, `mod`, `abs()`, `max(,)`, `min(,)`
- ▶ арифметическими отношениями: `<`, `<=`, `>`, `>=`, `=`, `!=`
- ▶ побитовыми операциями: `<<`, `>>`, `::` (*конкатенация*)
- ▶ операциями индексирования: `[i]` (*элемент массива*), `[i:j]` (*подслово слова*)

## ( $\nu$ ) Выражения

Выражения строятся в условиях статической типизации с небольшими возможностями приведения типов (о которых можно почитать в документации) над

- ▶ операциями для множеств:  $\{e_1, \dots, e_k\}$ , `union`, `in`, `e1..e2`
- ▶ тернарным оператором: `?:`
- ▶ оператором выбора:

`case альтернатива; ... альтернатива esac`

- ▶ `альтернатива ::= формула : выражение`
- ▶ выбирается **первая по прочтению** альтернатива, значение формулы которой — `TRUE`
- ▶ результат — значение выражения выбранной альтернативы
- ▶ ...



## ( $\nu$ ) 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;
```

Можно описывать системы, состоящие из многих модулей

Имя модуля можно использовать в качестве типа переменной

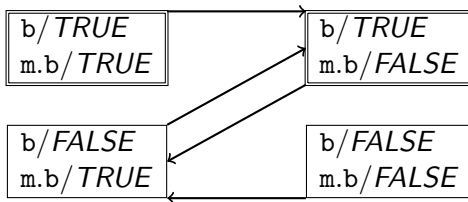
Объявленная так “переменная” — это **экземпляр** модуля, участвующий в **синхронной композиции**: при совершении перехода главный модуль и **все** объявленные экземпляры одновременно совершают переход

Доступ к локальным переменным модуля выглядит так же, как доступ к полям структуры в C/C++

## ( $\nu$ ) 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;
```

Модель  $M[\text{main}]$  выглядит так:



## ( $\nu$ ) 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;
```

Объявление макроопределения выглядит так:

`DEFINE имя := выражение ;`

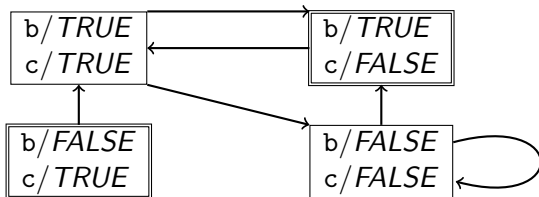
Вместо имени макроопределения во всех местах модуля подставляется соответствующее выражение

Значение макроопределения не входит в состояние модели

## ( $\nu$ ) 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;
```

Модель  $M[\text{main}]$  выглядит так:



## ( $\nu$ ) Модули с параметрами

```
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;
```

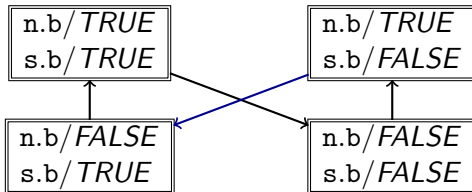
Описание модуля может содержать **параметры** — имена, перечисленные в скобках через запятую после имени модуля. Внутри модуля параметр работает *примерно* как макроопределение:

- ▶ имя макроопределения — это имя параметра;
- ▶ тело макроопределения в конкретном экземпляре — это *next*-выражение, записанное на соответствующем месте в объявлении экземпляра

## ( $\nu$ ) Модули с параметрами

```
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;
```

Модель  $M[main]$  выглядит так:



## ( $\nu$ ) Инвариант состояний

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

Иногда бывает удобно описать общее устройство модели и после этого сказать:

“из всего, что я описал, в модели остаются только состояния, удовлетворяющие формуле  $\varphi$ , и переходы между этими состояниями”

Такое ограничение множества состояний записывается так:

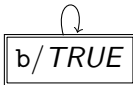
**INVAR**  $\varphi$ ;

Записанное так ограничение  $\varphi$  добавляется ко всем остальным ограничениям в описании модуля как для переменных, так и для их next-аналогов

## ( $\nu$ ) Инвариант состояний

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

Модель  $M[\textit{main}]$  содержит ровно одно состояние:





## ( $\nu$ ) Специальные переменные

Стабильная переменная определяется так же, как и обычная, но с ключевым словом `FROZENVAR` вместо `VAR`.

Стабильная переменная — это переменная, значение которой определяется в начальном состоянии системы и больше не изменяется

Более точно:

- ▶ значение стабильной переменной присутствует в состоянии модели
- ▶ запрещено писать выражения, в которых говорится, каким должно быть значение стабильной переменной после перехода
- ▶ неявно предполагается ограничение `TRANS next(x) = x;` для каждой стабильной переменной `x`

## ( $\nu$ ) Специальные переменные

**Входная переменная** определяется так же, как и обычная, но с ключевым словом **IVAR** вместо **VAR**

Входная переменная — это переменная, значениями которой размечаются **переходы** из каждого конкретного состояния

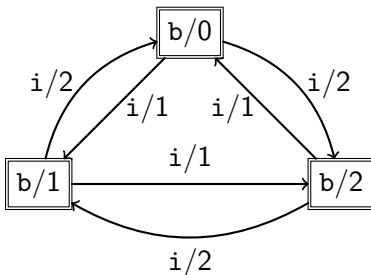
Более точно:

- ▶ значение входной переменной **не является** составной частью состояния модели
- ▶ входные переменные **не могут** встречаться в виде **next**-аналогов и в формулах, не относящихся к переходам
- ▶ входные переменные могут встречаться в **TRANS**, и в **TRANS** описываются ограничения не только на начальные и конечные переменные состояния, но и на значения входных переменных

## ( $\nu$ ) 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
```

Модель  $M[\text{main}]$  выглядит так:



## ( $\nu$ ) ASSIGN

Ключевое слово **ASSIGN** используется для единообразной записи распространённых видов ограничений:

- ▶ **ASSIGN** *переменная* := *выражение* равносильно
  - ▶ **INVAR** *переменная* in *выражение*, если значение выражения — множество
  - ▶ **INVAR** *переменная* = *выражение* в противном случае
- ▶ **ASSIGN** **init**(*переменная*) := *выражение* равносильно
  - ▶ **INIT** *переменная* in *выражение*, если значение выражения — множество
  - ▶ **INIT** *переменная* = *выражение* в противном случае
- ▶ **ASSIGN** **next**(*переменная*) := *next-выражение* равносильно
  - ▶ **TRANS** **next**(*переменная*) in *next-выражение*, если значение выражения — множество
  - ▶ **TRANS** **next**(*переменная*) = *next-выражение* в противном случае

## ( $\nu$ ) Асинхронная композиция

В NuSMV версии 2.6.0 поддерживаются, но считаются **устаревшими** встроенные средства асинхронной композиции модулей (*согласно семантике чередующихся вычислений*)

При выполнении заданий разрешается использовать эти средства, однако для понимания того, как именно их следует применять, следует **внимательно** прочитать документацию

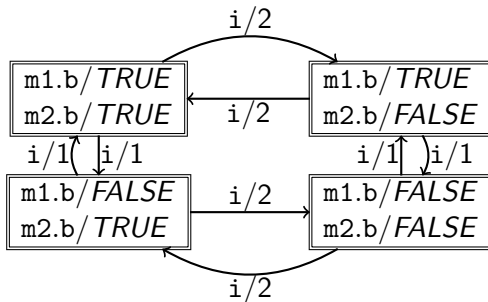
Асинхронная композиция модулей может быть организована при помощи синхронной:

- ▶ в каждом экземпляре есть параметр “твой ход”
- ▶ если “твой ход” = TRUE, то экземпляр изменяет своё состояние, иначе сохраняет текущие значения
- ▶ внешний модуль (или специальный экземпляр) выступает в роли арбитра: определяет очерёдность ходов

## ( $\nu$ ) 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;
```

Модель  $M[\text{main}]$  выглядит так:



# ( $\nu$ ) Справедливость

*Напоминание, лекция 4:* ограничения справедливости

- ▶ делят все пути модели Крипке на справедливые и несправедливые
- ▶ изменяют семантику кванторов пути:
  - ▶  $A\varphi$  = “для любого **справедливого** пути верно  $\varphi$ ”
  - ▶  $E\varphi$  = “существует **справедливый** путь, для которого верно  $\varphi$ ”

## ( $\nu$ ) Справедливость

*Напоминание, лекция 5:* классический способ задания ограничения справедливости для CTL выглядит так:

- ▶ элементарное ограничение — это множество состояний модели Крипке
- ▶ путь справедлив относительно элементарного ограничения  $\Leftrightarrow$  хотя бы одно из состояний ограничения встречается в нём бесконечно часто
- ▶ ограничения справедливости — это множество элементарных ограничений
- ▶ путь справедлив относительно ограничений справедливости  $\Leftrightarrow$  он справедлив относительно каждого элементарного ограничения



## ( $\nu$ ) 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 классическое элементарное ограничение справедливости для CTL:

- ▶ путь модели  $M[main]$  справедлив  $\Leftrightarrow$  формула каждого ограничения **JUSTICE**, записанного в *main* или порождаемого в *main* объявленными экземплярами, истинна в бесконечно многих состояниях этого пути
- ▶ каждым ограничением **JUSTICE**  $\varphi(x_1, \dots, x_n)$ , описанным или порождённым в модуле, экземпляр **e** которого объявлен в модуле **m**, порождается ограничение **JUSTICE**  $\varphi\{x_1/e.x_1, \dots, x_n/e.x_n\}$  в модуле **m**

## ( $\nu$ ) 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;
```

В данном примере модель  $M[\textit{main}]$  совпадает с моделью предыдущего примера, и в модуле *main* порождаются два ограничения справедливости:

- ▶ JUSTICE turn = 1;
- ▶ JUSTICE turn = 2;

Содержательная трактовка этих ограничений справедливости:  
несправедливый путь — это путь,  
в котором какой-либо из процессов в некоторый момент  
навсегда становится неактивным

## ( $\nu$ ) Нетотальные модели Крипке

Согласно определению в *лекции 3*, модель Крипке — это **тотальный** граф: из каждой вершины исходит хотя бы одна дуга

Во многих средствах верификации (в том числе *NuSMV*) можно описывать такие модели, граф которых **нетотален**

Такие модели (**нетотальные модели Крипке**) обычно считаются **некорректными**, за исключением случаев, явно описанных в документации

В частности, при “наивном” использовании NuSMV для проверки CTL-формул нетотальные модели Крипке **всегда** некорректны: результат верификации не отражает в полной мере устройство модели

Следить за тотальностью модели — это

**задача того, кто описывает модель**

## ( $\nu$ ) 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
```

Конец семинара 4