Математические методы верификации схем и программ

Лекторы:

Захаров Владимир Анатольевич

Подымов Владислав Васильевич

e-mail рассказчика:

valdus@yandex.ru

Осень 2017

Семинар 4

NuSMV

(обзор средства)

Задача model checking

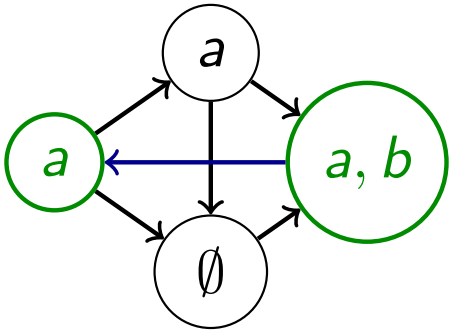
Широкая формальная постановка

Даны

- модель Крипке $M = (S, S_0, R, L)$
- темпоральная формула φ

Требуется вычислить множество состояний

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



Задача model checking

Узкая формальная постановка

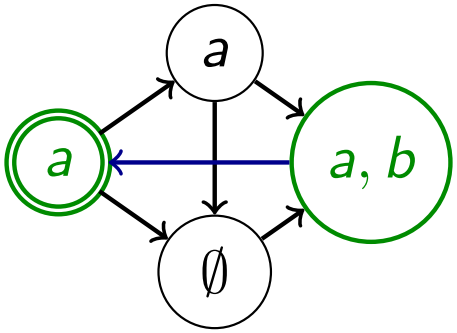
Даны

- модель Крипке $M = (S, S_0, R, L)$
- CTL-формула φ

Требуется проверить выполнимость формулы φ в модели M:

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

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



Задача model checking

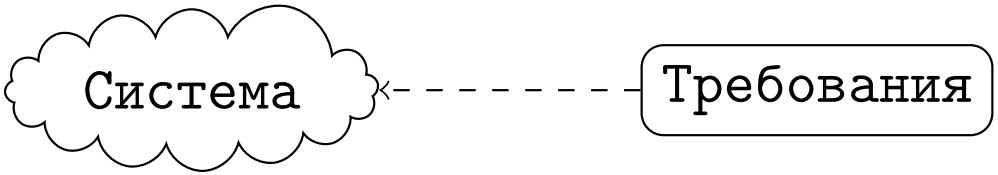
Содержательная постановка

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

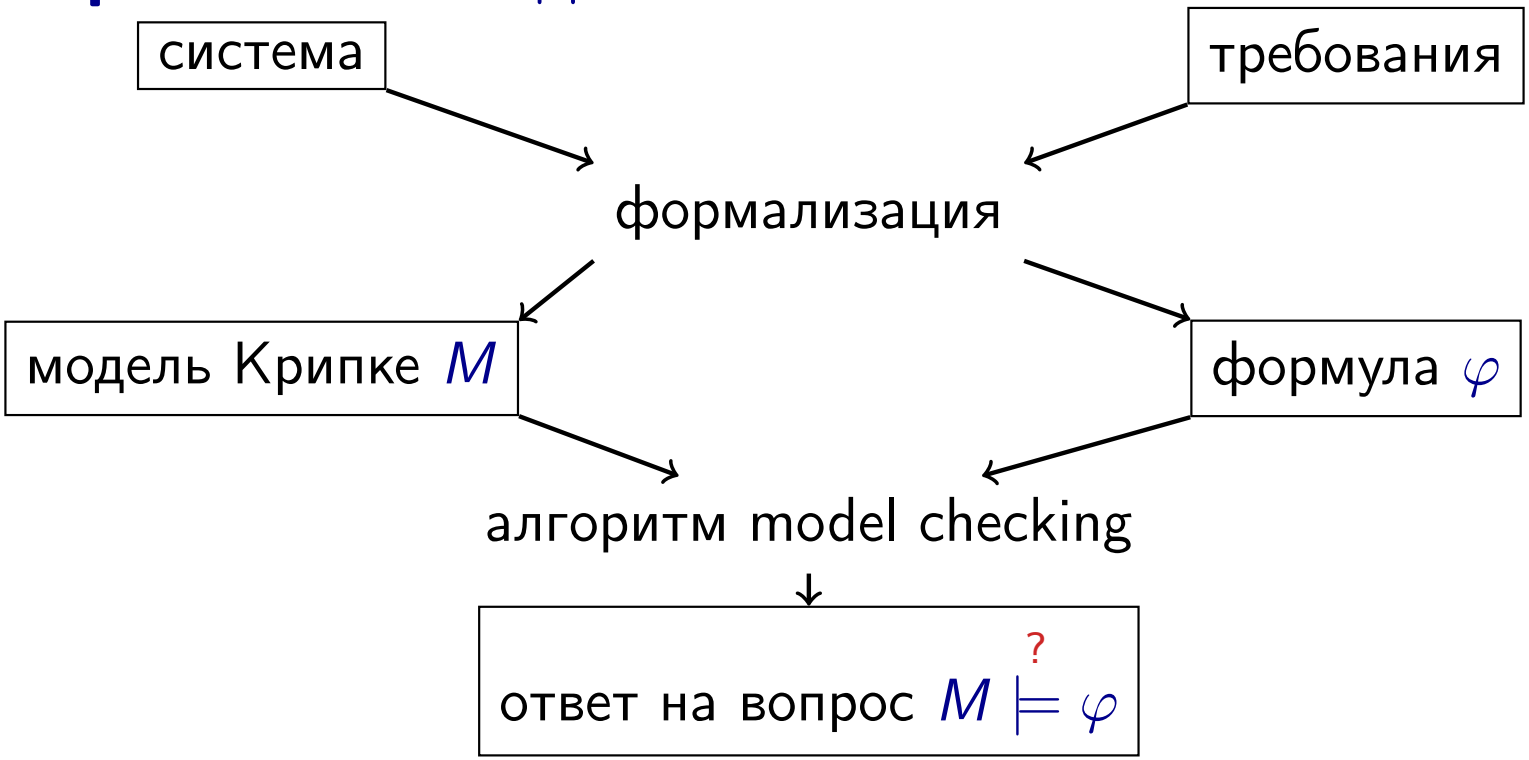
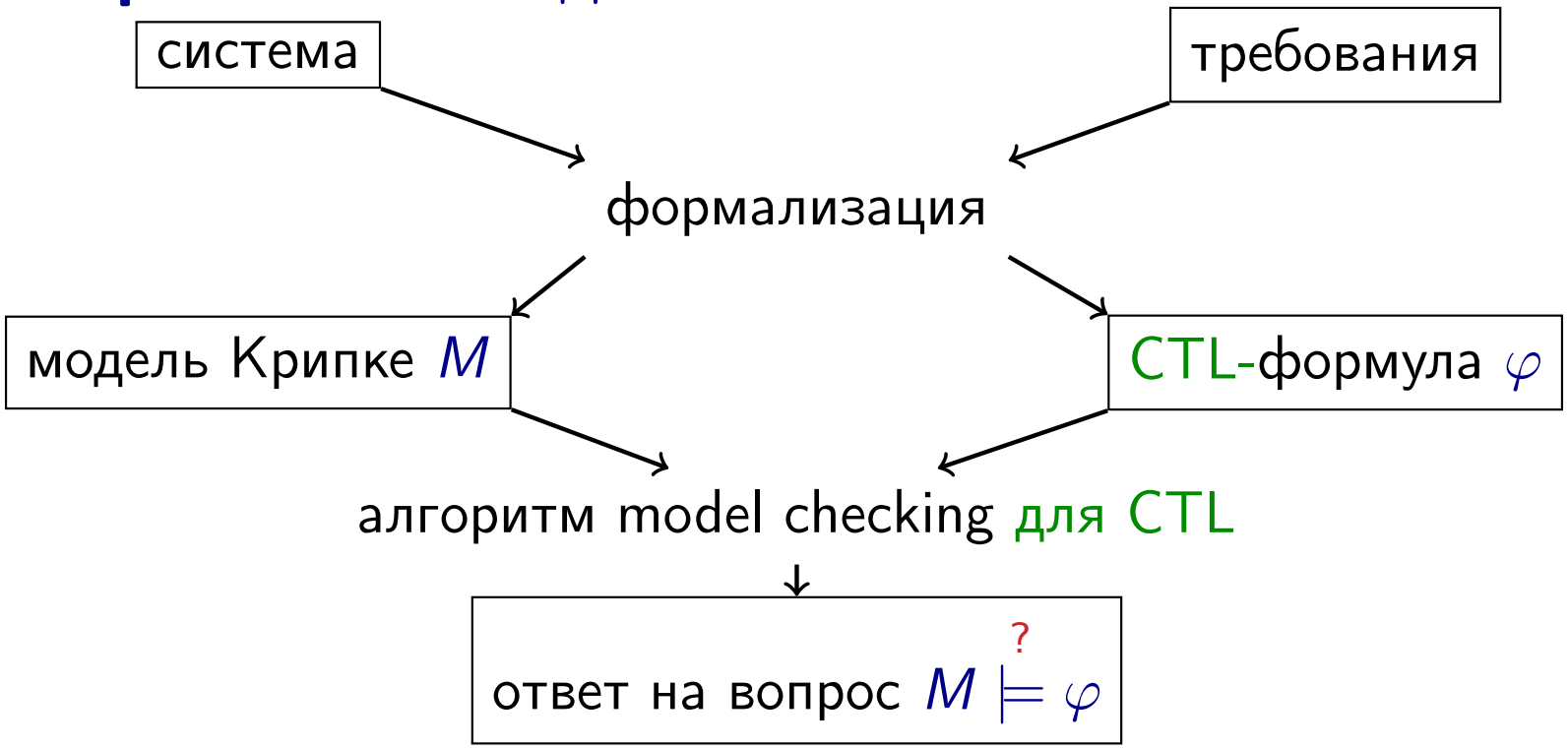


Схема решения ЗАДАЧИ



На ближайших семинарах будет рассматриваться логика ветвящегося времени

Схема решения ЗАДАЧИ

модель Крипке M → алгоритм model checking для CTL ← CTL-формула φ

↓

ответ на вопрос $M \overset{?}{\models} \varphi$

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

модель Крипке M → CTL-формула φ

↓

символьный алгоритм model checking для CTL

↓

ответ на вопрос $M \overset{?}{\models} \varphi$

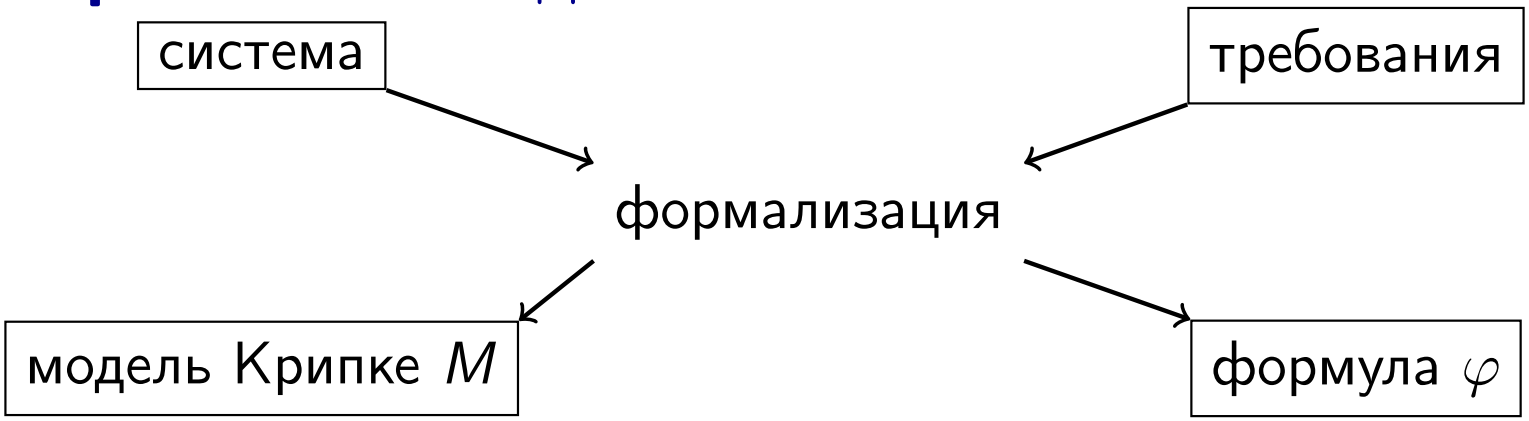
Эффективность символьного алгоритма зависит от эффективности средств работы с булевыми функциями

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 полуформальном 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

(ν) Hello, World!

```
MODULE main
VAR b : boolean;
INIT b;
TRANS next(b) = !b;
CTLSPEC AG b;
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 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

(ν) 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:

$\boxed{\mathtt{b}/FALSE}$ $\boxed{\mathtt{b}/TRUE}$

(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 $M[md]$ — 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:

$$\boxed{\mathtt{b}/FALSE} \qquad \boxed{\mathtt{b}/TRUE}$$

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

$$\boxed{\mathtt{b}/FALSE} \qquad \boxed{\mathtt{b}/TRUE}$$

(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 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`

(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);
```

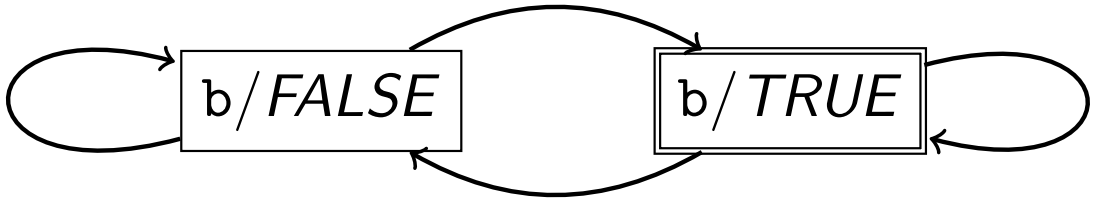
Next-формула — это *next-выражение* (*next-expression*) типа boolean, то есть выражение над переменными модуля и их next-аналогами

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

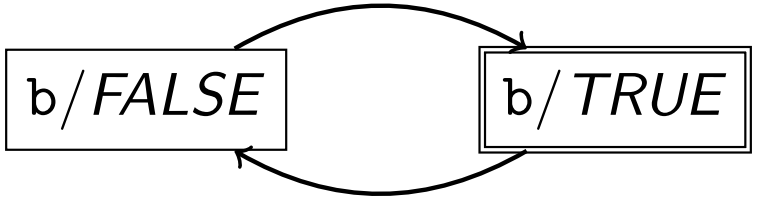
(ν) Переходы

```
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);
```

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



С учётом строки 4 модель $M[main]$ выглядит так:



(ν) Спецификации

```
MODULE main
VAR b : boolean;
INIT b;
TRANS next(b) = !b;
CTLSPEC AG b;
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 (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]

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

(ν) Типы данных

- булев тип: `boolean`, значения — $\{TRUE, FALSE\}$
- перечисление, или множество: $\{val_1, \ldots, val_k\}$, 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`

(ν) Константы

- константы типа `boolean`: `TRUE`, `FALSE`
- целочисленные константы: $0, 1, 2, \ldots$
  (*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

(ν) 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*)

(ν) Выражения

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

- операциями для множеств: $\{e_1, \ldots, e_k\}$, `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
- ...

(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;
```

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

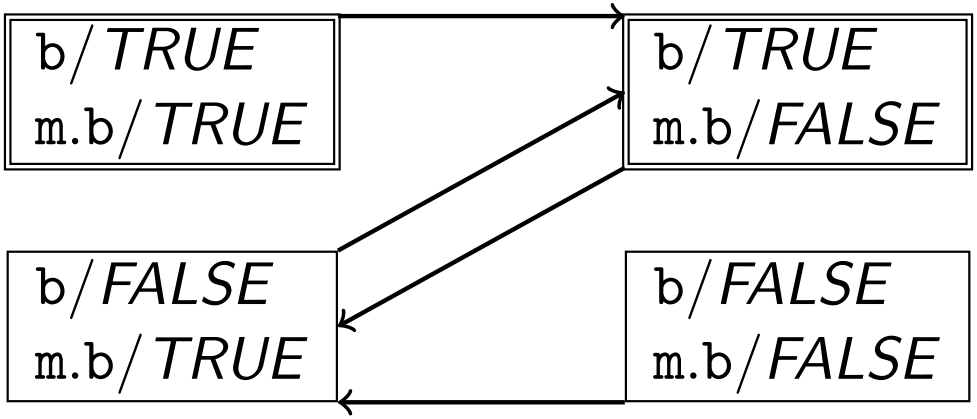
Доступ к локальным переменным модуля выглядит так же, как доступ к полям структуры в C/C++

(ν) Композиция модулей

```
MODULE main
VAR
  b : boolean;
  m : aux;
INIT b;
TRANS next(b) = m.b;
CTLSPEC AG (b != m.b); -- неправда
CTLSPEC AX AG (b != m.b); -- правда

MODULE aux
VAR b : boolean;
TRANS next(b) = !b;
```

Модель $M[main]$ выглядит 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;
```

Объявление макроопределения выглядит так:

DEFINE *имя* := *формула* ;

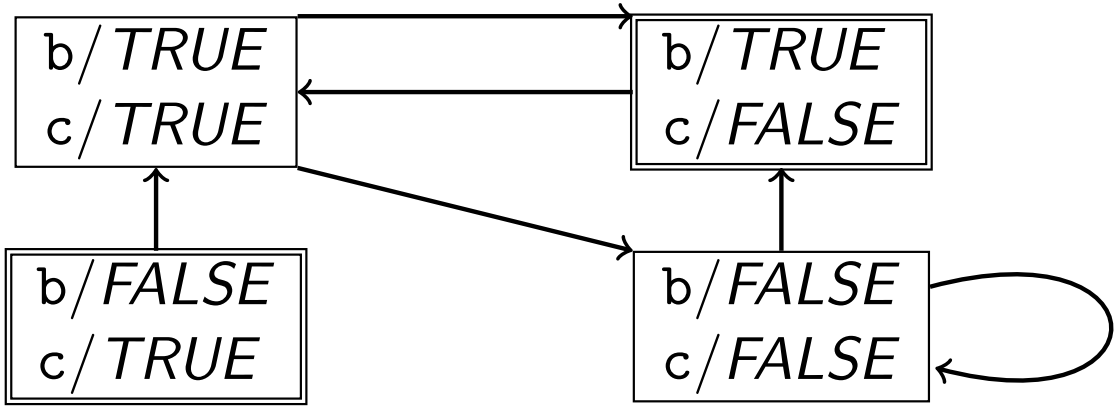
Вместо имени макроопределения во всех местах модуля подставляется соответствующая формула

Значение макроопределения не входит в состояние модели

(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;
```

Модель $M[main]$ выглядит так:



(ν) 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

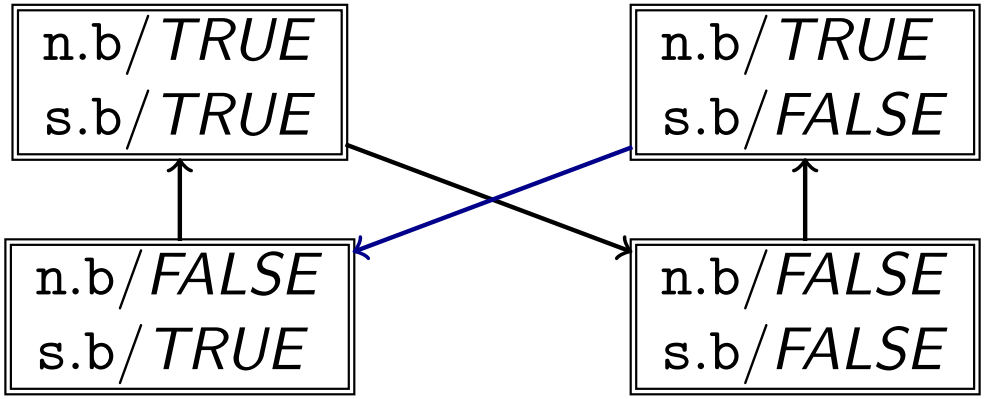
(ν) Модули с параметрами

```
MODULE main
VAR n : neg(n.b);
    s : sum(n.b, s.b);
CTLSPEC AG (n.b -> AX !n.b); -- правда
CTLSPEC AG (!n.b -> AX n.b); -- правда
CTLSPEC AG (n.b xor s.b -> AX s.b); -- правда
CTLSPEC AG (n.b xnor s.b -> AX !s.b); -- правда

MODULE neg(x)
VAR b : boolean;
TRANS next(b) = !x;

MODULE sum(x,y)
VAR b : boolean;
TRANS next(b) = x xor y;
```

Модель $M[main]$ выглядит так:



(ν) 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 φ, и переходы между этими состояниями"

Такое ограничение множества состояний записывается так:

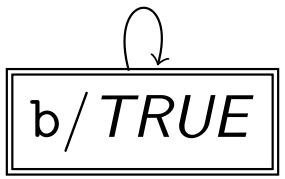
`INVAR` φ`;`

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

(ν) Инвариант состояний

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

Модель $M[main]$ содержит ровно одно состояние:



(ν) 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`

(ν) Специальные переменные

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

Входная переменная — это переменная, допустимые значения которой определены для каждого конкретного состояния и используются для более удобного описания соотношения начальных и конечных значений состояний перехода

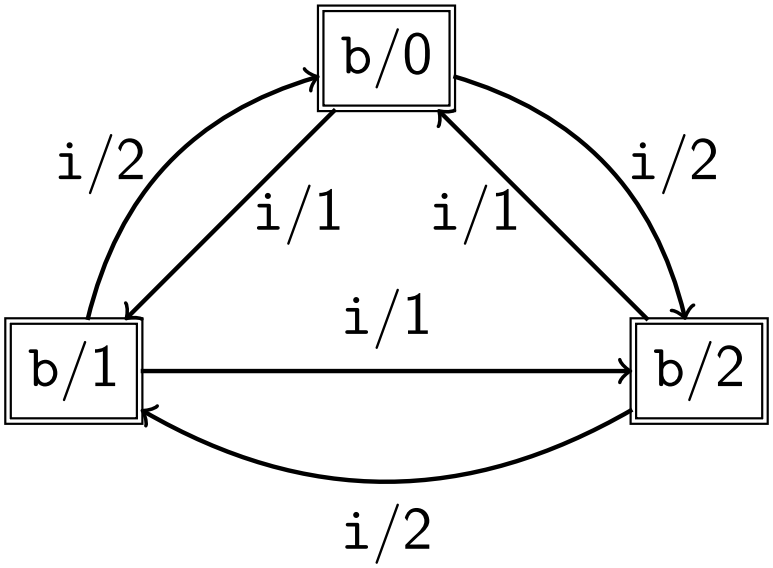
Более точно:

- значение входной переменной **не является** составной частью состояния модели
- входные переменные **не могут** встречаться в виде `next`-аналогов, в `INIT` и в описании требований
- входные переменные могут встречаться в `INVAR` и `TRANS`

(ν) Специальные переменные

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

Модель $M[main]$ выглядит так:



(ν) 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

(ν) 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:

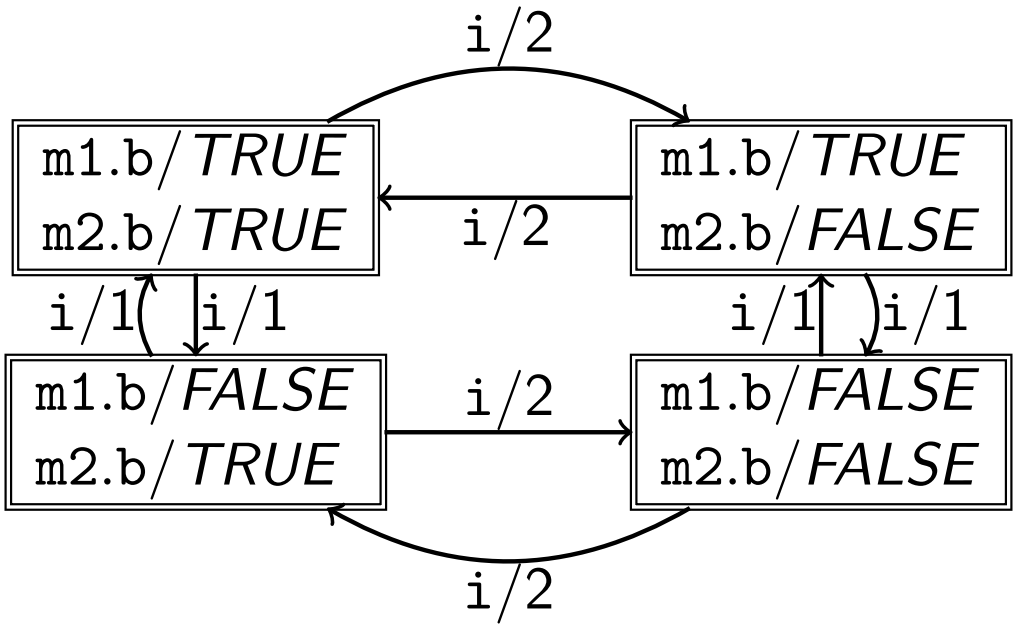
- в каждом экземпляре есть параметр "твой ход"
- если "твой ход" = `TRUE`, то экземпляр изменяет своё состояние, иначе сохраняет текущие значения
- внешний модуль (или специальный экземпляр) выступает в роли арбитра: определяет очерёдность ходов

(ν) Асинхронная композиция

```
MODULE main
IVAR turn : {1,2};
VAR m1 : aux(turn = 1);
    m2 : aux(turn = 2);
CTLSPEC AG AF (!m1.b | !m2.b); -- правда
CTLSPEC AG AF !m1.b; -- неправда

MODULE aux(active)
VAR b : boolean;
ASSIGN next(b) := case
  active : !b;
  TRUE : b;
esac;
```

Модель $M[main]$ выглядит так:



(ν) 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 φ"
  - $E\varphi$ = "существует **справедливый** путь, для которого верно φ"

(ν) 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

(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 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` $\varphi(x_1, \ldots, x_n)$, описанным или порождённым в модуле, экземпляр `e` которого объявлен в модуле `m`, порождается ограничение `JUSTICE` $\varphi\{\mathtt{x}_1/e.x_1, \ldots \mathtt{x}_\mathtt{n}/e.x_n\}$ в модуле `m`

(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;
```

В данном примере модель $M[main]$ 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

(ν) 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

(ν) Нетотальные модели Крипке

```
MODULE main
VAR b : boolean;
INVAR b;
TRANS next(b) = !b;
CTLSPEC AG b;
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