Математические методы верификации схем и программ

Лекторы:

Захаров Владимир Анатольевич

Подымов Владислав Васильевич

e-mail рассказчика:

valdus@yandex.ru

Осень 2016

Напоминание

Какую задачу мы решали в последних лекциях?

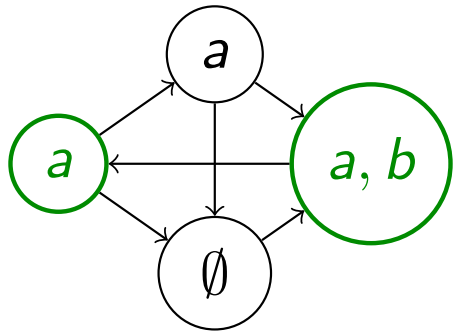
Широкая формальная постановка

Даны

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

Требуется вычислить множество состояний

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



Напоминание

Какую задачу мы решали в последних лекциях?

Узкая формальная постановка

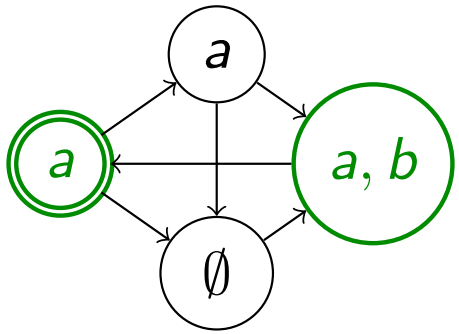
Даны

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

Требуется проверить выполнимость формулы φ в модели M:

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

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



Напоминание

Какую задачу мы решали в последних лекциях?

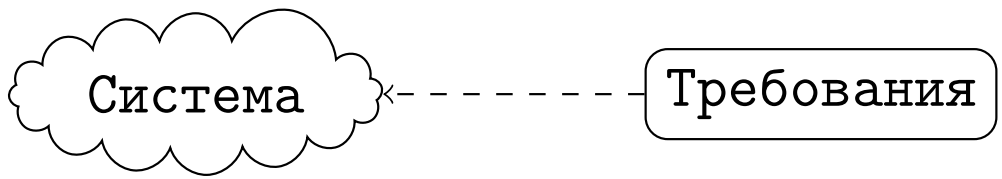
Содержательная постановка

Даны

- неформальное описание системы
- содержательное описание требований к системе

Требуется проверить,

удовлетворяет ли система набору требований



Программа-максимум — научить вас решать такую содержательно поставленную ЗАДАЧУ

Схема решения ЗАДАЧИ

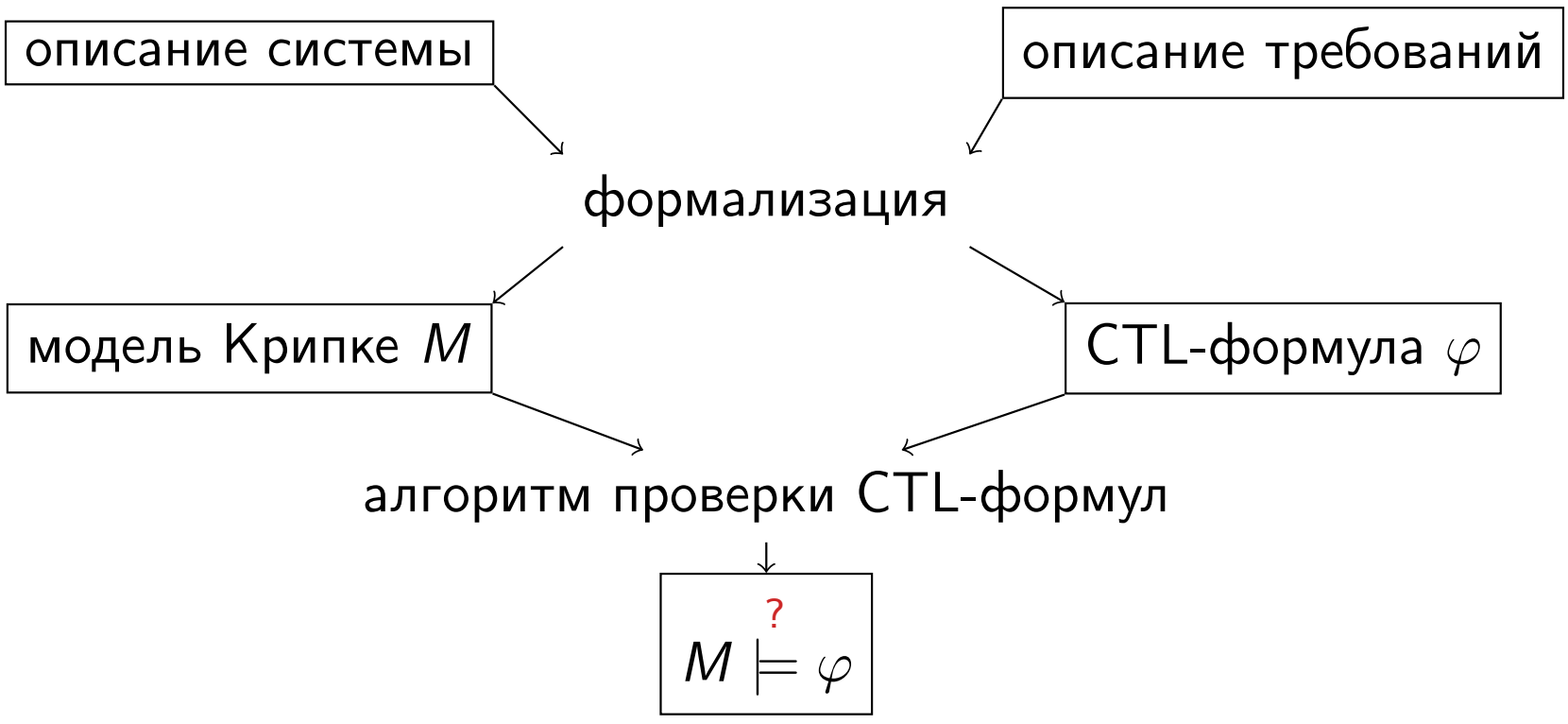
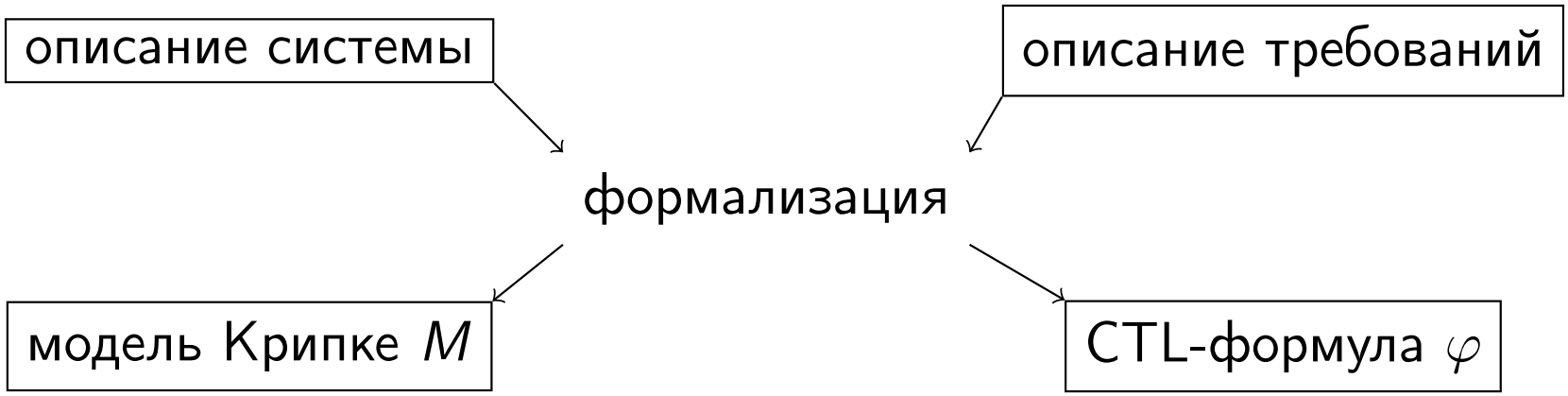
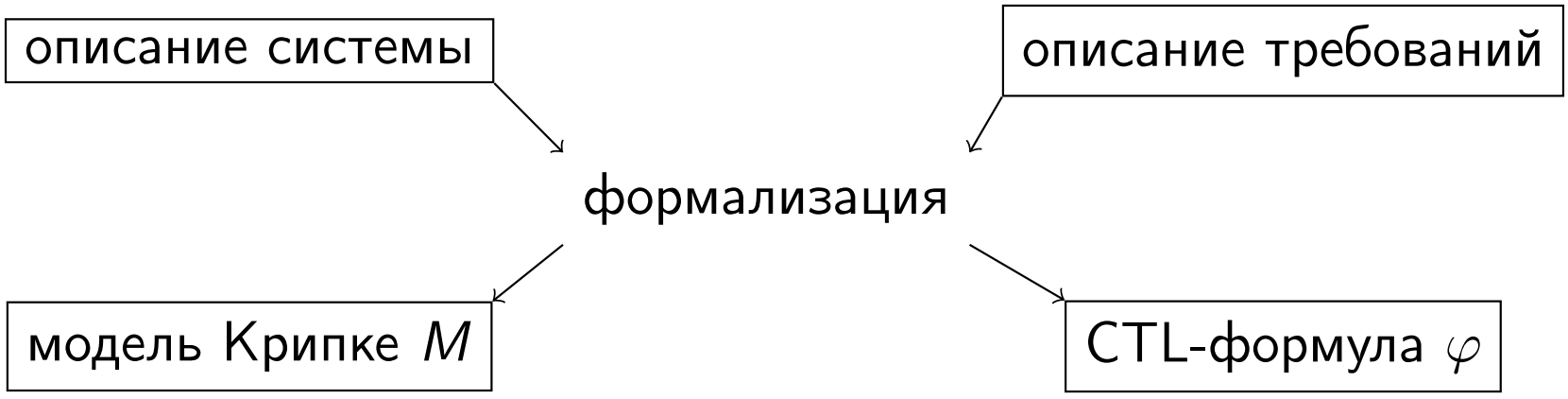


Схема решения ЗАДАЧИ



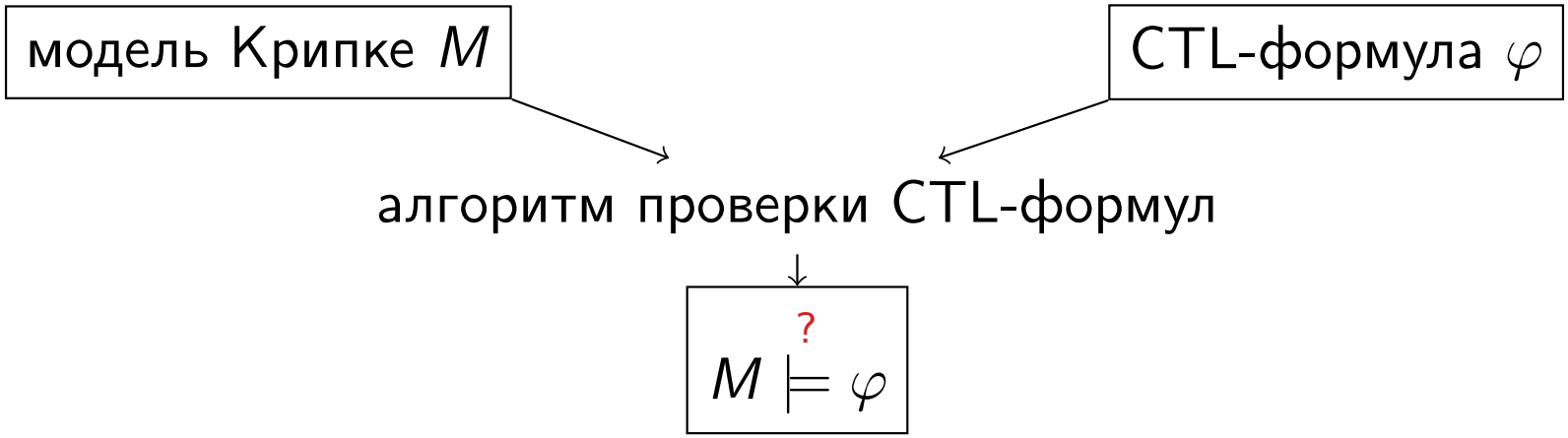
- для некоторых классов систем можно придумать алгоритм трансляции в модели Крипке
  - программы со строгой семантикой
  - комбинационные и последовательные схемы (*схемы из функциональных элементов с задержкой*)
  - ...
- в общем случае нет никаких ограничений на способы описания системы, а значит, нет и единообразных способов формализации системы

Схема решения ЗАДАЧИ



- требования обычно формулируются на естественном языке
- тот, кто предъявляет требования к системе, часто не может точно сказать, чего он хочет
- формализация требований — отдельный долгий и кропотливый процесс

Схема решения ЗАДАЧИ



Как только модель и формула получены, процесс проверки становится абсолютно бездумным

Вы уже знаете два основных алгоритма проверки соотношения $M \models \varphi$:

- табличный алгоритм
- символьный алгоритм

А какой алгоритм разумнее реализовывать в средстве верификации?

Средства верификации

Табличный алгоритм **нагляден** и лежит в основе всех других алгоритмов, но **неэффективен**: чем больше модель Крипке, тем медленнее она будет обрабатываться

Символьный алгоритм **менее нагляден**, но **намного более эффективен**:

- основа символьного алгоритма — преобразование и анализ булевых функций
- программистский арсенал содержит средства эффективной работы с *большими* булевыми функциями:
  - библиотеки для работы с ROBDD
  - SAT/SMT-решатели
  - ...

Как правило, в средствах верификации используется **символьный алгоритм**, насыщенный разнообразными оптимизациями и эвристиками

А насколько популярен model checking для CTL в среде программистов?

Средства верификации

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 скорее всего 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

Синтаксис NuSMV: модули

Система в NuSMV описывается как композиция модулей:

`MODULE <имя модуля>(<аргументы>) <тело модуля>`

Модуль — это описание
недетерминированного конечного автомата

Блок с аргументами модуля может быть опущен *вместе со скобками*

Вид и назначение аргументов будут описываться дальше, а сейчас полагаем, что их нет

Имя модуля, как и все имена, — это непустая строка из символов *A..Za..z*0..9_$#−, начинающаяся с одного из символов *A..Za..z*_

Все компоненты описания автомата, которые будут показаны дальше, — это элементы тела модуля

Синтаксис NuSMV: типы данных

Пространство состояний модуля образовано (*если коротко*) декартовым произведением областей переменных, используемых при описании модуля

Типы переменных:

- `boolean`: значения `TRUE` и `FALSE`
- `integer`: значения от `INT_MIN + 1` до `INT_MAX`; это те же машинно-зависимые константы, что и в C/C++
- $\{val_1, \ldots, val_k\}$: перечисление (*enumeration*); val_i — либо число, либо имя
  - в частности, `integer` — это перечисление особого вида

Синтаксис NuSMV: типы данных

Пространство состояний модуля образовано (*если коротко*) декартовым произведением областей переменных, используемых при описании модуля

Типы переменных:

- `unsigned word[i]`, где $i > 0$: беззнаковые числа с битовой записью ширины `i`
- `signed word[i]`, где $i > 0$: знаковые числа с битовой записью ширины `i`
- `array i..j of T`, где i, j — константы и T — тип: массив элементов типа `T` с индексацией от `i` до `j`
  - при определении массивов допускается вложенность, например, `array 0..2 of array 3..7 of boolean`

NuSMV умеет (*немного, но всё же*) **приводить типы** (*читайте про это в документации или постигайте практикой*)

Синтаксис NuSMV: пространство состояний

Пространство состояний модуля описывается так:

```
VAR
  <имя> : <тип>;
  <имя> : <тип>;
...
```

Пример:

```
MODULE bird
VAR
  satiety : {FED, HUNGRY, DEAD};
  flying : boolean;
```

Так описывается автомат с шестью состояниями:

(FED, FALSE)	(HUNGRY, FALSE)	(DEAD, FALSE)
(FED, TRUE)	(HUNGRY, TRUE)	(DEAD, TRUE)

Синтаксис NuSMV: начальные состояния

Множество начальных состояний модуля — это конъюнкция `INIT`-выражений такого вида:

`INIT` <*выражение*>;

Точку с запятой можно опускать (*здесь и в других аналогичных местах*)

<*выражение*> — это выражение типа boolean, построенное над переменными модуля

Следует запомнить, что **все выражения в NuSMV — это формулы**, даже если в них встречается равенство

Например, запись вида "`a = b`" часто используется при описании систем в NuSMV, но означает не "обычное" последовательное присваивание, а **формулу**, истинную тогда и только тогда, когда значения `a` и `b` совпадают

Синтаксис NuSMV: выражения

Что **точно** можно использовать при построении выражений:

- константы
  - булевого типа: `TRUE`, `FALSE`
  - целочисленного типа: `0, 1, -1, ...`
  - символьного типа: все имена, использовавшиеся в перечислениях
  - интервального типа: `i..j`, где i, j — целочисленные константы; значение — множество целых чисел от `i` до `j`
  - word-типа: целочисленное значение в двоичной (`b`), восьмеричной (`o`), десятичной (`d`) либо шестнадцатеричной (`h`) системе счисления в особом формате, например:
    - `0ub5_10011` или `0b_10011` — беззнаковая константа ширины 5, описывающая число 19
    - `0so_77` — знаковая константа ширины 6, описывающая число -1

Синтаксис NuSMV: выражения

Что **точно** можно использовать при построении выражений:

- имена переменных
- скобки, как обычно в формулах
- булевы и побитовые операции: `!` (отрицание), `&`, `|`, `xor`, `xnor`, `->`, `<->`
- арифметические (*и где возможно, также булевы*) операции и отношения: `+`, `-`, `*`, `/`, `mod`, `<`, `<=`, `>`, `>=`, `=`, `!=`, `abs(...)` (модуль), `max(..., ...)`, `min(..., ...)`
- операции над битовыми векторами: `»` (сдвинуть вправо), `«` (сдвинуть влево), `::` (конкатенация)
- операции индексирования: `...[e]` (*e*-й элемент массива, *e*-й бит слова), `...[e1:e2]` (подслово слова от бита *e*1 до бита *e*2

Синтаксис NuSMV: выражения

Что **точно** можно использовать при построении выражений:

- операции над множествами: `union`, `in`, `{e1, ..., ek}` (множество из k элементов), `e1..e2` (интервал от $e1$ до $e2$)
- условное выражение: `e ? e1 : e2`
- case-выражение: `case` <*выражение*> `:` <*выражение*>`;` <*выражение*> `:` <*выражение*>`; ... esac`
  - просматриваются пары <*выражение*> `:` <*выражение*> в порядке следования
  - выбирается **первая** пара, левое выражение которой истинно
  - результат case-выражения — правое выражение этой пары
- ...

Синтаксис NuSMV: начальные состояния

Пример

```
MODULE bird
VAR
  satiety : {FED, HUNGRY, DEAD};
  flying : boolean;
INIT ! (satiety = DEAD);
INIT flying = TRUE | flying = FALSE;
```

Так описывается множество из четырёх начальных состояний:

(FED, FALSE) (HUNGRY, FALSE)
(FED, TRUE) (HUNGRY, TRUE)

Вторая строка с INIT избыточна, она здесь только для того, чтобы показать, что разрешено писать несколько INIT-выражений

Синтаксис NuSMV: переходы автомата

Совокупность переходов автомата определяется **конъюнкцией** TRANS-выражений:

TRANS <*next-выражение*>;

Next-выражение отличается от обычного выражения тем, что в нём могут встречаться записи вида next(<*имя переменной*>)

<*имя переменной*> — это значение переменной **до** выполнения перехода

next(<*имя переменной*>) — это значение переменной **после** выполнения перехода

Подобные выражения уже встречались в лекциях раньше: *в лекции 3* рассказывалось, как описывать отношения переходов с помощью формул, используя два комплекта переменных ("штрихованный" и "нештрихованный")

next(<*имя переменной*>) — это штрихованная версия переменной (*в нотации лекции 3*)

Синтаксис NuSMV: переходы автомата

Пример

```
MODULE bird
VAR
  satiety : {FED, HUNGRY, DEAD};
  flying : boolean;
INIT ! (satiety = DEAD);
TRANS satiety = FED -> next(satiety) = HUNGRY;
TRANS satiety = HUNGRY ->
      next(satiety) in {HUNGRY, DEAD};
TRANS satiety = DEAD ->
      next(satiety) = DEAD & !next(flying);
```

И какие же переходы описываются такой совкупностью TRANS-выражений?

Синтаксис NuSMV: переходы автомата

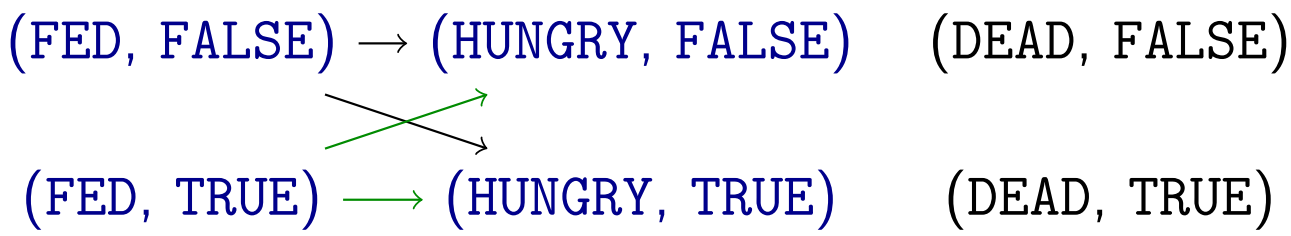
```
TRANS satiety = FED -> next(satiety) = HUNGRY;
TRANS satiety = HUNGRY ->
      next(satiety) in {HUNGRY, DEAD};
TRANS satiety = DEAD ->
      next(satiety) = DEAD & !next(flying);
```

(FED, FALSE) (HUNGRY, FALSE) (DEAD, FALSE)

(FED, TRUE) (HUNGRY, TRUE) (DEAD, TRUE)

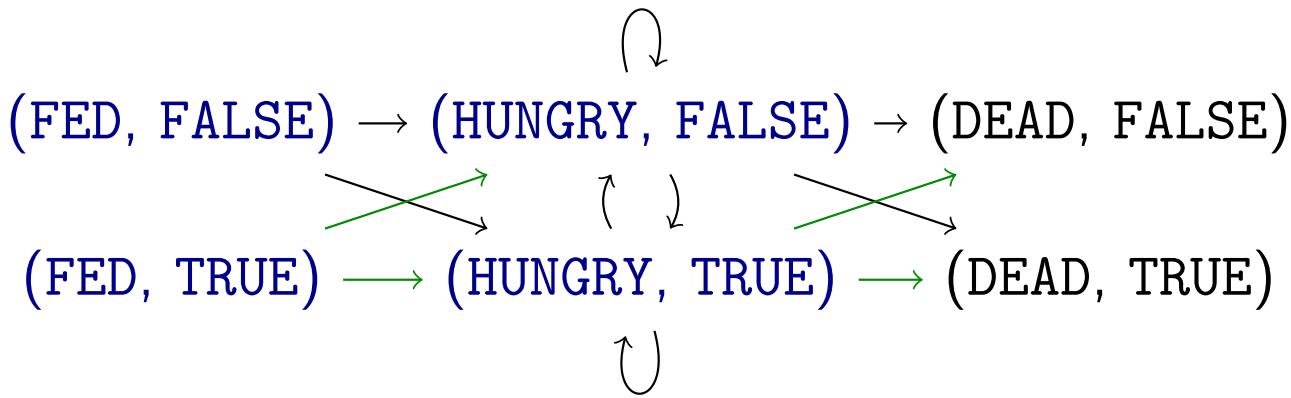
Синтаксис NuSMV: переходы автомата

```
TRANS satiety = FED -> next(satiety) = HUNGRY;
TRANS satiety = HUNGRY ->
      next(satiety) in {HUNGRY, DEAD};
TRANS satiety = DEAD ->
      next(satiety) = DEAD & !next(flying);
```



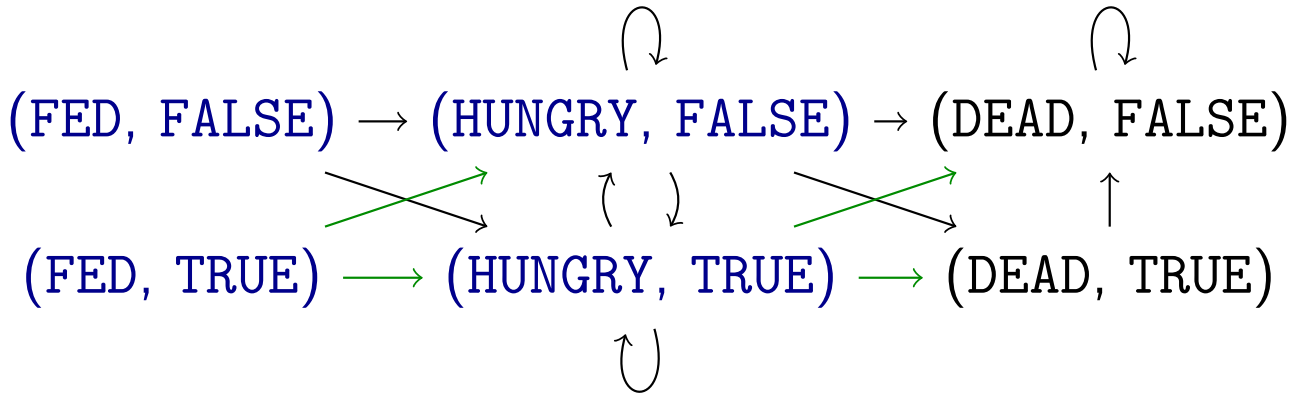
Синтаксис NuSMV: переходы автомата

```
TRANS satiety = FED -> next(satiety) = HUNGRY;
TRANS satiety = HUNGRY ->
      next(satiety) in {HUNGRY, DEAD};
TRANS satiety = DEAD ->
      next(satiety) = DEAD & !next(flying);
```



Синтаксис NuSMV: переходы автомата

```
TRANS satiety = FED -> next(satiety) = HUNGRY;
TRANS satiety = HUNGRY ->
      next(satiety) in {HUNGRY, DEAD};
TRANS satiety = DEAD ->
      next(satiety) = DEAD & !next(flying);
```



А почему новые переходы **объединялись** с построенными ранее, несмотря на то что по семантике множество переходов описывается **конъюнкцией** выражений?

Синтаксис NuSMV: переходы автомата

Другой пример

```
MODULE bird
VAR
  satiety : {FED, HUNGRY, DEAD};
  flying : boolean;
TRANS satiety = HUNGRY;
```

Так тоже можно писать в NuSMV

Похожа ли эта система на ненасытную бессмертную птицу?

Как будет выглядеть модель Крипке и почему?

```
MODULE bird
VAR
  satiety : {FED, HUNGRY, DEAD};
  flying : boolean;
INIT satiety = HUNGRY;
TRANS satiety = HUNGRY;
```

А здесь всё хорошо?

Присваивания

Нередко при описании систем в NuSMV используются блоки, напоминающие последовательность присваиваний:

```
ASSIGN
  <имя переменной> := <выражение>;
  init(<имя переменной>) := <выражение>;
  next(<имя переменной>) := <выражение>;
  ...
```

Как работает “`v := e;`”: (*точнее описано в документации*)

- если выражение имеет тип переменной, то это работает как “`TRANS v = e`” (*всё чуть иначе — см. “IVAR”*)
- если значение выражения — множество элементов того же типа, что и `v`, то это работает как “`TRANS v in e`”

ASSIGN-выражение с `next` работает точно так же

ASSIGN-выражение с `init` работает *почти* так же: эквивалентная конструкция содержит “`INIT`” вместо “`TRANS`”, и ключевое слово “`init`” снимается

Синхронная и асинхронная композиции

Система, описанная на языке NuSMV, обычно состоит из нескольких модулей, работающих *параллельно*

А какие виды параллельного выполнения систем вы знаете?

NuSMV поддерживает два вида параллельной композиции модулей: синхронную и асинхронную

Синхронная композиция:

- все модули, участвующие в композиции, одновременно совершают один переход, и это объявляется переходом композиции

Асинхронная композиция:

- произвольно выбирается один из модулей композиции
- выбранный модуль совершает переход, остальные модули не совершают перехода
- получившееся изменение состояния системы объявляется результатом выполнения перехода композиции

Синтаксис NuSMV: экземпляры модулей

Модуль может быть использован в качестве **типа переменной** в другом модуле
Оба типа композиции описываются похожим образом

```
MODULE s
VAR inst1 : m1; inst2 : m1; inst3 : m2;
```

В модуле `s` **синхронно** запущены два экземпляра модуля `m1` и один экземпляр модуля `m2`

`inst1`, `inst2`, `inst3` — имена этих экземпляров

```
MODULE a
VAR inst1 : process m1; inst2 : process m1;
```

В модуле `a` **асинхронно** запущены два экземпляра модуля `m1`

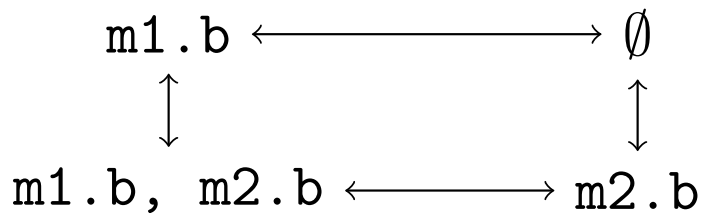
Синтаксис NuSMV: экземпляры модулей

Пример

```
MODULE room
VAR m1 : process mosquito; m2 : process mosquito;

MODULE mosquito
VAR buzzing : boolean;
ASSIGN next(buzzing) := !buzzing;
```

Так запущенные комары жужжат асинхронно:



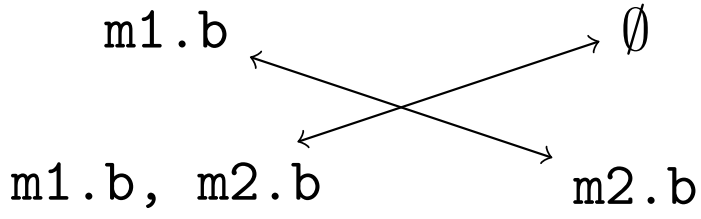
Синтаксис NuSMV: экземпляры модулей

Пример

```
MODULE room
VAR m1 : mosquito; m2 : mosquito;

MODULE mosquito
VAR buzzing : boolean;
ASSIGN next(buzzing) := !buzzing;
```

Так запущенные комары жужжат синхронно:



Синтаксис NuSMV: доступ к локальным переменным

Если в модуле `M` запущен экземпляр модуля `m`, то в модуле `M` можно использовать значения всех переменных этого экземпляра

Доступ к значениям переменных происходит так же, как доступ к значениям полей структуры в C:

```
MODULE M
VAR x : m; b : boolean;
ASSIGN next(b) := !x.b xor b;

MODULE m
VAR b : boolean;
ASSIGN next(b) := !b;
```

А как эта система работает?

Синтаксис NuSMV: аргументы модуля

Чтобы экземпляр модуля мог использовать значения переменных, описанных вовне, в его определении можно дописать аргументы:

```
MODULE m(a1, a2, a3) ...
```

Значения аргументов могут использоваться в модуле так же, как и остальные переменные (но без `init` и `next`)

При определении экземпляра модуля на местах всех аргументов прописываются **выражения**:

```
VAR
  b1 : boolean;
  x : m(TRUE, b1, y.b);
  y : m(b1 & x.b, x.b, b1);
```

Синтаксис NuSMV: главный модуль

Как NuSMV поймёт, какой из модулей **главный**: описывает ту самую систему, которую мы хотим исследовать?

Правило очень простое:

- главный модуль обязан называться `main`
- главный модуль обязан не содержать аргументов

Семантика модуля без аргументов — (*в чистом виде, без всяких оговорок*) модель Крипке

В модели Крипке можно проверить выполнимость CTL-формулы

NuSMV умеет это делать (*и даже больше, но сейчас достаточно этого*)

Синтаксис NuSMV: CTL-спецификации

В главном модуле можно явно написать набор CTL-формул, выполнимость которых мы хотим проверить

Синтаксис очень простой:

- описание спецификации — тоже часть тела модуля
- спецификация предваряется словом `CTLSPEC`
- спецификация — это **выражение**, дополненное связками `AF`, `EF`, `AG`, `EG`, `AX`, `EX`, `A[...U...]`, `E[...U...]`
- нетемпоральные выражения, стоящие непосредственно под темпоральными операторами, должны иметь булев тип

Пример

```
MODULE room
VAR m1 : mosquito; m2 : mosquito
CTLSPEC EF AG !(m1.buzzing & m2.buzzing)
```

Да когда же эти комары помрут наконец?

Полноценный пример

```
MODULE main
  VAR b : bird(s.location = NEAR); s : swarm;
  CTLSPEC AG AF (b.satiety = FED);
MODULE bird(food)
  VAR satiety : {FED, HUNGRY};
  ASSIGN next(satiety) := case
    satiety = HUNGRY & food : FED;
    TRUE : HUNGRY;
  esac;
MODULE swarm
  VAR location : {NEAR, FAR};
```

Верно ли, что сколько бы бессмертная птица ни прожила, она обязательно перекусит насекомыми ещё раз?

Синтаксис NuSMV: справедливость

NuSMV поддерживает два способа задания справедливости
Оба способа являются элементами тела модуля

Первый способ: `JUSTICE` <*выражение*>
Рассматриваются только такие поведения системы, в которых выражение в экземпляре модуля становится истинным бесконечно часто

Второй способ: `COMPASSION(`<*выражение*>`,` <*выражение*>`)`
Рассматриваются только такие поведения системы, для которых верно: если выражение в первом аргументе становится истинным бесконечно часто, то и выражение во втором аргументе будет становиться истинным бесконечно часто

Дополнительная возможность:
Для упрощения написания систем в каждом экземпляре, вызванном с помощью слова `process`, предопределена переменная `running`, истинная тогда и только тогда, когда процесс выбран при асинхронном переходе

Полноценный пример, дополненный

```
MODULE main
  VAR b : bird(s.location = NEAR); s : swarm;
  CTLSPEC AG AF (b.satiety = FED);
MODULE bird(food)
  VAR satiety : {FED, HUNGRY};
  ASSIGN next(satiety) := case
    satiety = HUNGRY & food : FED;
    TRUE : HUNGRY;
  esac;
MODULE swarm
  VAR location : {NEAR, FAR};
  JUCTICE location = NEAR;
```

А теперь всё нормально?

Насколько “справедлива” такая справедливость?

Полноценный пример, исправленный

```
MODULE main
  VAR b : bird(s.location = NEAR); s : swarm;
  CTLSPEC AG AF (b.satiety = FED);
MODULE bird(food)
  VAR satiety : {FED, HUNGRY};
  ASSIGN next(satiety) := case
    satiety = HUNGRY & food : FED;
    TRUE : HUNGRY;
  esac;
MODULE swarm
  VAR location : {NEAR, FAR};
  ASSIGN next(location) := location = NEAR ? FAR : NEAR;
```

А теперь где подвох?

Полноценный пример, дважды исправленный

```
MODULE main
  VAR b : process bird(s.location = NEAR);
      s : process swarm;
  CTLSPEC AG AF (b.satiety = FED);
MODULE bird(food)
  VAR satiety : {FED, HUNGRY};
  ASSIGN next(satiety) := case
    satiety = HUNGRY & food : FED;
    TRUE : HUNGRY;
  esac;
MODULE swarm
  VAR location : {NEAR, FAR};
  ASSIGN next(location) := location = NEAR ? FAR : NEAR;
```

А теперь птица наконец-таки покормится?

А должна?

А можно ли придумать “разумную” справедливость, при которой всё заработает как надо?

Полноценный пример, трижды исправленный

```
MODULE main
  VAR b : process bird(s.location = NEAR);
      s : process swarm(b.satiety = HUNGRY);
  CTLSPEC AG AF (b.satiety = FED);
MODULE bird(food)
  VAR satiety : {FED, HUNGRY};
  ASSIGN next(satiety) := case
    satiety = HUNGRY & food : FED;
    TRUE : HUNGRY;
  esac;
MODULE swarm(hunting)
  VAR location : {NEAR, FAR};
  ASSIGN next(location) :=
          location = NEAR & !hunting ? FAR : NEAR;
```

А так лучше?

Полноценный пример, трижды исправленный и дополненный

```
MODULE main
  VAR b : process bird(s.location = NEAR);
      s : process swarm(b.satiety = HUNGRY);
  CTLSPEC AG AF (b.satiety = FED);
MODULE bird(food)
  VAR satiety : {FED, HUNGRY};
  ASSIGN next(satiety) := case
    satiety = HUNGRY & food : FED;
    TRUE : HUNGRY;
  esac;
  JUSTICE running;
MODULE swarm(hunting)
  VAR location : {NEAR, FAR};
  ASSIGN next(location) :=
          location = NEAR & !hunting ? FAR : NEAR;
  JUSTICE running;
```

А такое описание лучше всех предыдущих?

Небольшое замечание

Разработчики NuSMV объявили ключевое слово `process`, переменную `running` и асинхронное исполнение в целом устаревшими особенностями, которые использовать нехорошо

Взамен они предлагают самим моделировать асинхронное исполнение компонентов системы, первоначально имея **только** синхронное исполнение

А как это сделать?

Но так как в NuSMV версии 2.6.0 асинхронное исполнение всё ещё есть, то будем его использовать

Заключение: как запускать NuSMV

my.smv

```
MODULE main
  VAR b : process bird(s.location = NEAR); s : process swarm(b.satiety = HUNGRY);
  CTLSPEC AG AF (b.satiety = FED);
MODULE bird(food)
  VAR satiety : {FED, HUNGRY};
  ASSIGN next(satiety) := case
    satiety = HUNGRY & food : FED;
    TRUE : HUNGRY;
  esac;
  JUSTICE running;
MODULE swarm(hunting)
  VAR location : {NEAR, FAR};
  ASSIGN next(location) := location = NEAR & !hunting ? FAR : NEAR;
  JUSTICE running;
```

```
terminal> ./NuSMV my.smv
*** This is NuSMV 2.6.0 (compiled on Fri Sep 30 13:46:58 2016)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

WARNING *** Processes are still supported, but deprecated.      ***
WARNING *** In the future processes may be no longer supported. ***

WARNING *** The model contains PROCESSes or ISAs. ***
WARNING *** The HRC hierarchy will not be usable. ***
-- specification AG (AF b.satiety = FED)  is true
```