Formal techniques for software and hardware verification

Lecturers:

Vladimir Zakharov

Vladislav Podymov

e-mail:

valdus@yandex.ru

2020, fall semester

Seminar 4

NuSMV

tool overview

Model checking problem

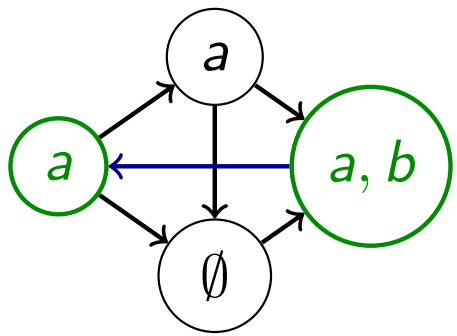
Broad formal statement

Given

- a Kripke structure $M = (S, S_0, R, L)$ and
- a formula φ

Compute the set of states

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



Model checking problem

Narrow formal statement

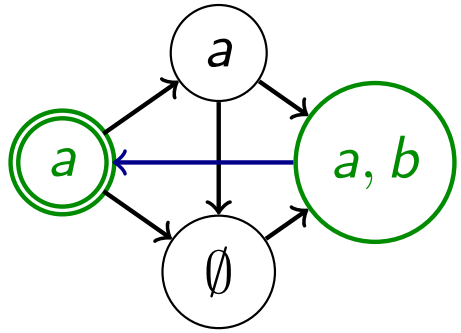
Given

- a Kripke structure $M = (S, S_0, R, L)$ and
- a ctl-formula φ

Check whether the model M satisfies the formula φ

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

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



Model checking problem

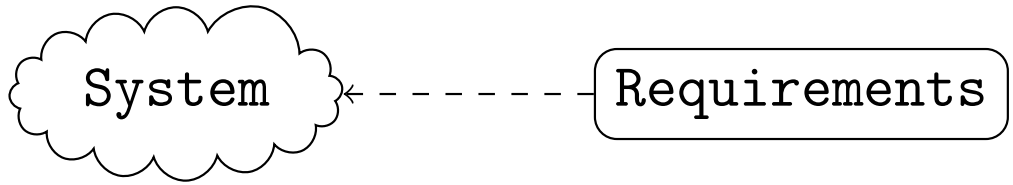
Informal statement

Given

- an informal system description, and
- an infomally written requirements

Check whether

the system satisfies all the requirements



The main goal of all the remaining seminars is to learn how to solve this informally stated PROBLEM

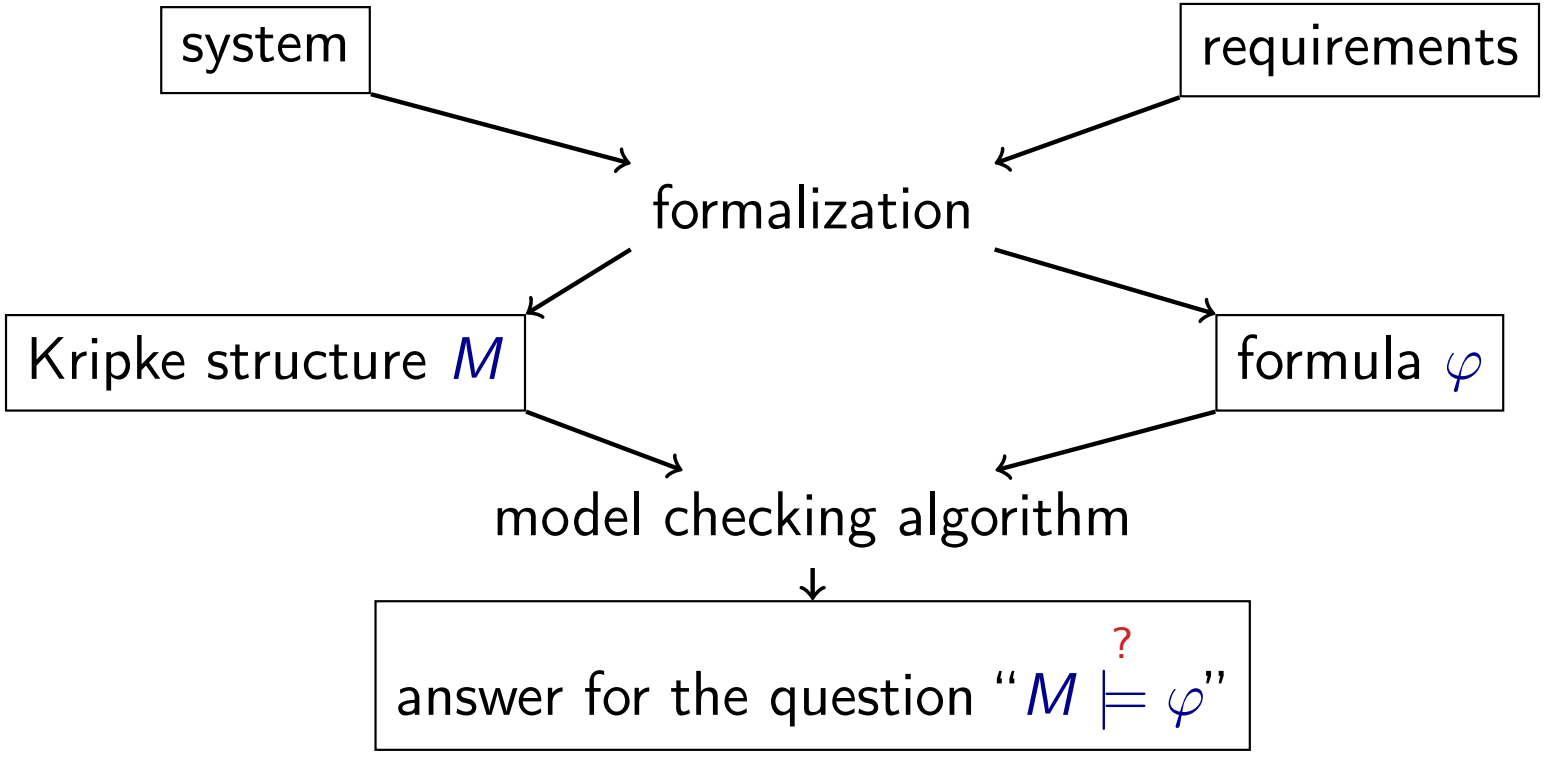
Model checking problem

The PROBLEM will be discussed in conjunction with program tools for formal verification

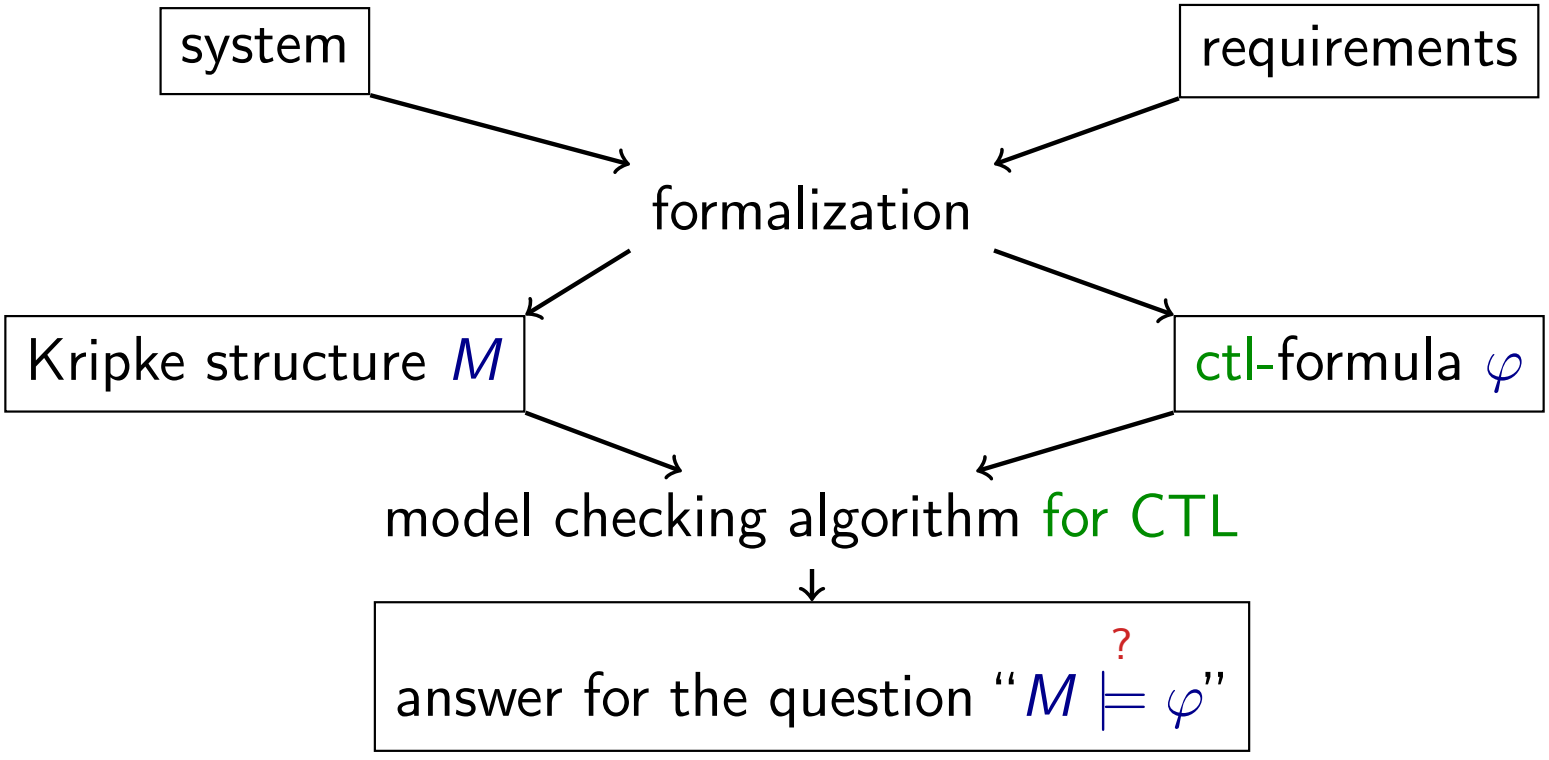
Each of the tools to be discussed takes a system and requirements written in some special language which

- provides readable descriptions of big systems and complex requirements
- has strict but comprehensible semantics closely related to the narrow formal statement of the model checking problem

General solution scheme for the PROBLEM

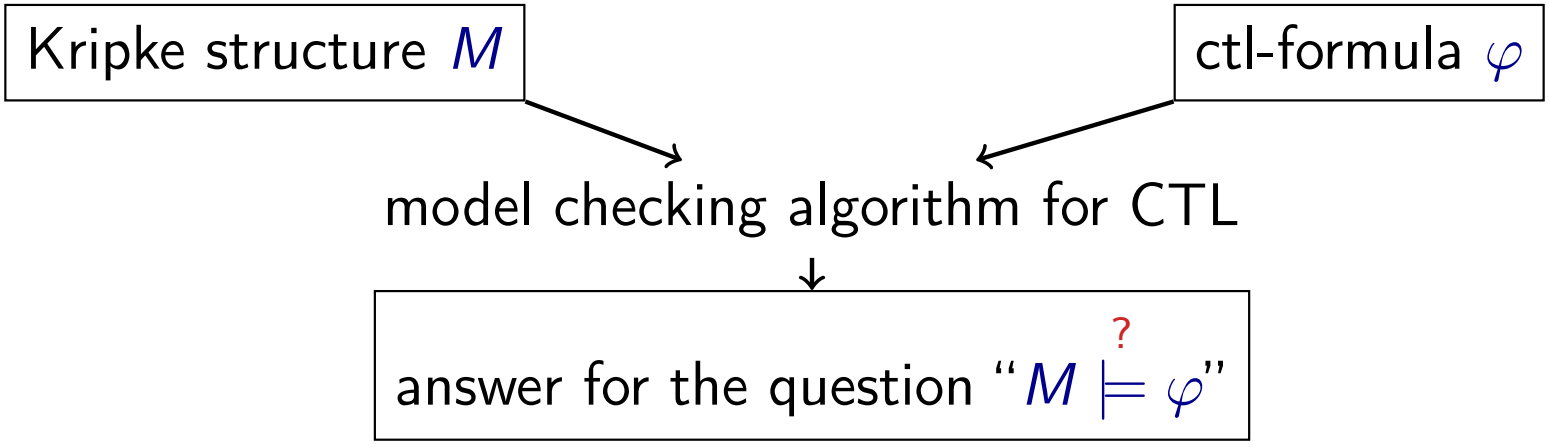


General solution scheme for the PROBLEM



NuSMV-related seminars are all about computational tree logic

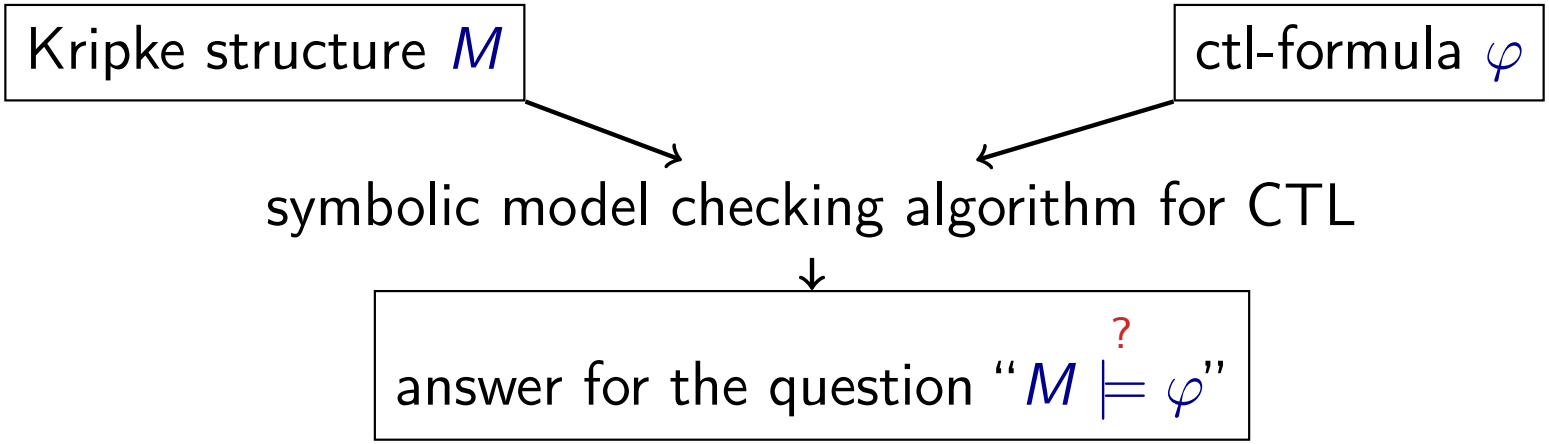
General solution scheme for the PROBLEM



You already know (*at least*) two model checking algorithms for CTL:

- **tableau-based**
  - clear and simple
  - lies in the base of other algorithms
  - inefficient
- **symbolic**
  - intricate and difficult to understand
  - much more efficient than the tableau-based (**?**)

General solution scheme for the PROBLEM



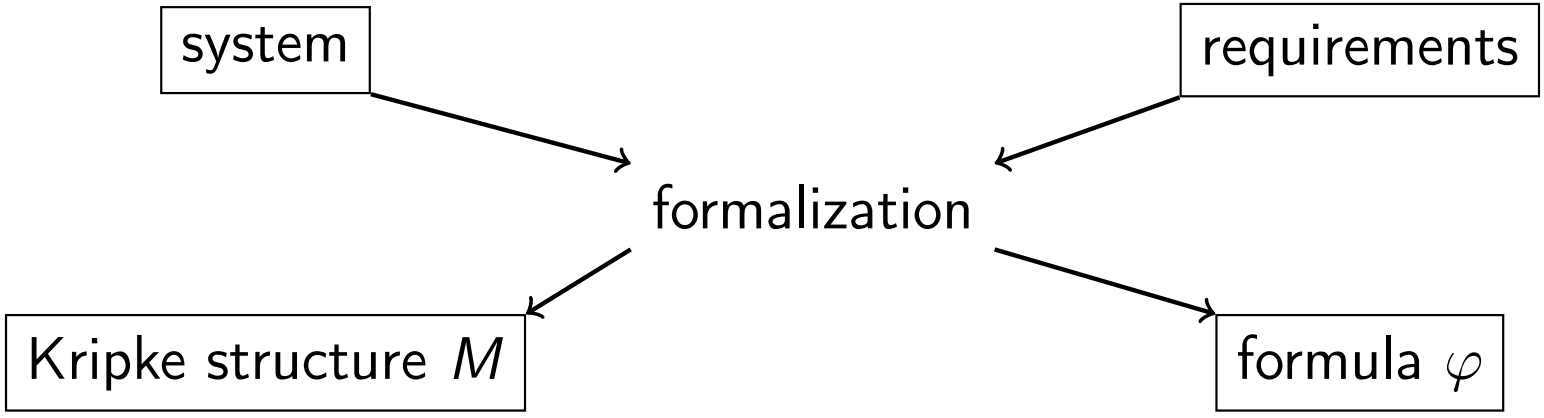
Efficiency of the symbolic algorithm depends on efficiency of tool for boolean function management

There exists a whole lot of such tools, and usually such a tool belongs to one of two classes:

1. a BDD management library
2. SAT/SMT-solver (a satisfiability checking tool for boolean and other formulae)

A verification tool usually has a fully implemented automatic model checking algorithm, which means that
"to use the verification algorithm" =
"to push the required button in the tool"

General solution scheme for the PROBLEM



The system and the requirements are usually given in completely- or partially-informal notions

System and requirement formalization (as a Kripke structure and a temporal formula, or in a verification tool language) is quite nontrivial process requiring some programming and mathematical skills

Such formalization is the main part of the remaining seminars

General solution scheme for the PROBLEM

For those interested, here are some tools capable of ctl-formulae against *some* models:

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

Disclaimer: *these are just several tools randomly taken from a related Wikipedia page some time ago*

We will focus on the tool NuSMV:

- it is open-source and free
- it is more-or-less popular
- its language is more-or-less simple, and based on boolean formulae just like in the symbolic algorithm

(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) Modules

```
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);
```

A module is a description of a Kripke structure together with requirements

A module named "`name`" (*with no parameters*) starts with

`MODULE name`

Acceptable identifiers contain the symbols "A-Za-z0-9_$#-" and start with "A-Za-z_"

For clarity, $M[m]$ will be used to denote Kripke structure defined by a module m

A main module is a module *with no parameters* named "main", and it is the module to be verified by default

(ν) Variables

```
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);
```

A state of $M[md]$ *in simple cases* is a set of values for all variables declared in *md* with the `VAR` kayword:

`VAR` *declaration*`;` *declaration*`;` ... *declaration*`;`

declaration ::= *identifier* : *type*

`boolean` is a type with the domain $\{TRUE, FALSE\}$

Example: the module $M[main]$ has two states:

b/*FALSE*

b/*TRUE*

(ν) Initial states

```
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);
```

Initial states of $M[md]$ are all states which satisfy **every** formula in *md* after the `INIT` keyword

A formula (also called a simple expression) is a *boolean expression* over variables declared in *md*

Example

If line 3 is deleted from *main*, then each state of $M[main]$ becomes initial:

If line 3 remains in place, then $M[main]$ contains exactly one initial state:

(ν) Transitions

```
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);
```

To define transitions, two sets of variables are used in $M[md]$:

1. variables declared in *md*
   - these variables correspond to values in a **current** state (transition source)
2. variables of the form `next(x)`, where `x` is a variable declared in *md*
   - these variables correspond to values in a **next** state (transition target)
   - such variables will be called next-variables

(ν) Transitions

```
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);
```

A `next`-formula is a boolean *next-expression*, which is an expression over variables and next-variables

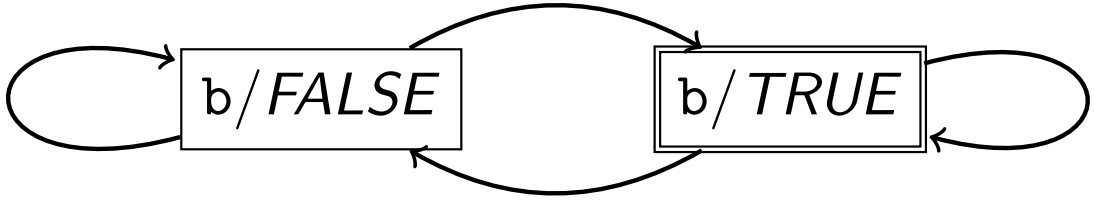
A transition exists in $M[md]$ iff
source and target variable values of the transition satisfy **every** next-formula after the `TRANS` keyword

(ν) Transitions

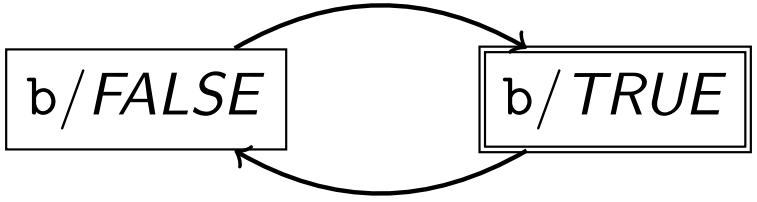
```
1 MODULE main
2 VAR b : boolean;
3 INIT b;
4 TRANS next(b) = !b;
5 CTLSPEC AG b;
6 CTLSPEC AG (!b -> AX b);
```

Example

If line 4 is deleted from *main*, then $M[main]$ is the following structure:



If line 4 remains in place, then $M[main]$ is the following structure:



(ν) Requirements

```
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);
```

All requirements for $M[md]$ can be written directly in md

We focus on requirements written in the language of CTL

Such requirements are written after the `CTLSPEC` keyword

BNF for NuSMV ctl-formulae (Φ) syntax:

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]

(I guess that the syntax is quite clear,
and no clarifications are needed)

(ν) Data types

- the boolean type: `boolean`, the domain is $\{TRUE, FALSE\}$
- an enumeration, or a set: $\{val_1, \ldots, val_k\}$, where val_i is either an integer or an identifier
- an interval `i..j` is a set of integers from `i` to `j` (including `i` and `j`), where `i` and `j` are constant expressions
- integers if a given width `i` (of a binary code):
  - unsigned: `unsigned word [i]`
  - signed: `signed word [i]`
- an array `array i..j of T` is a collection of variables of a type `T` indexed by integers from `i..j`
  - nested arrays are allowed, e.g.
    `array 0..2 of array 3..7 of boolean`

(ν) Constants

- `boolean` constants: `TRUE`, `FALSE`
- integer constants: $0, 1, 2, \ldots$
  (*these constants have some usage restrictions*)
- symbolic constants: these are the names used in enumerations
- `word`-constants in some numeral system — binary (`b`), octal (`o`), decimal (`d`), hexadecimal (`h`) — by example:
  - `0ub5_10011` and `0b_10011` is 19 written in 5 bits
  - `0so_77` is -1 written in 6 bits

(ν) Expressions

Expressions are statically typed with limited implicit type conversions (*see manuals*) over

- constants, declared variables, parentheses
- `next`-variables corresponding to declared variables (for `next`-expressions)
- boolean operations: `!`, `&`, `|`, `xor`, `xnor`, `->`, `<->`
- aritmetical operations: `+`, `-`, `*`, `/`, `mod`, `abs()`, `max(,)`, `min(,)`
- arithmetical relations: `<`, `<=`, `>`, `>=`, `=`, `!=`
- bitwise operations: `<<`, `>>`, `::` (*concatenation*)
- indexing operations: `[i]` (*array indexing*), `[i:j]` (*subword indexing for words*)

(ν) Expressions

Expressions are statically typed with limited implicit type conversions (*see manuals*) over

- set operations: $\{e_1, \ldots, e_k\}$, `union`, `in`, `e1..e2`
- the ternary operator: `?:`
- the case-operator:

  `case` *`alternative`* `;` `...` *`alternative`* `esac`

  - *`alternative`* `::=` *`formula`* `:` *`expression`*
  - the first alternative with a formula evaluated to `TRUE` is picked
  - the case-expression result equals to the result of the picked alternative
- ...

(ν) Module composition

```
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;
```

It is possible to define a system consisting of several modules

The name of a module can be used as a variable type

A variable declared with such a type is a module instance combined with rest of the system via synchronous composition: to execute a transition in the outer module means to execute transitions of all inner instances simultaneously

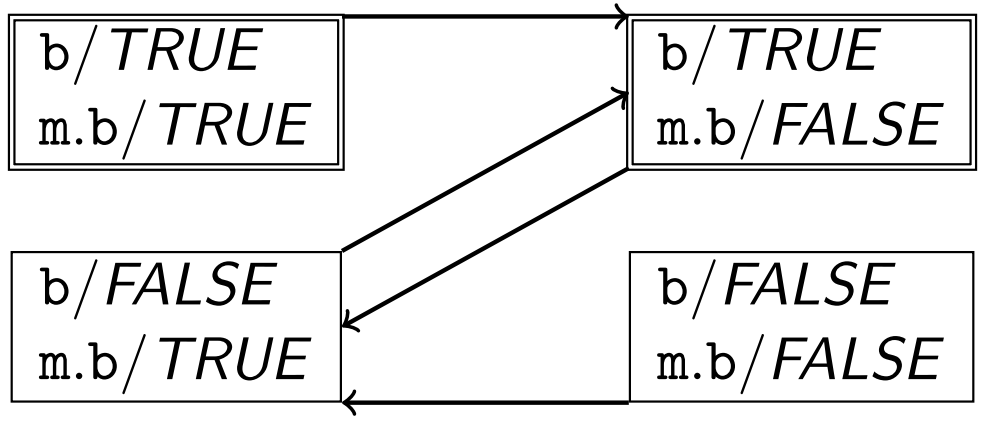
Local instance variables can be accessed from outer modules in the same way structure fields are accessed in C/C++

(ν) Module composition

```
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;
```

Example

$M[main]$ is the following structure:



(ν) Macros

```
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;
```

Macro definition is written as follows:

DEFINE *identifier* := *expression* ;

identifier is used in the same way as usual variables, and when the module is parsed, each *identifier* usage is replaced with the *expression* (surrounded by parentheses)

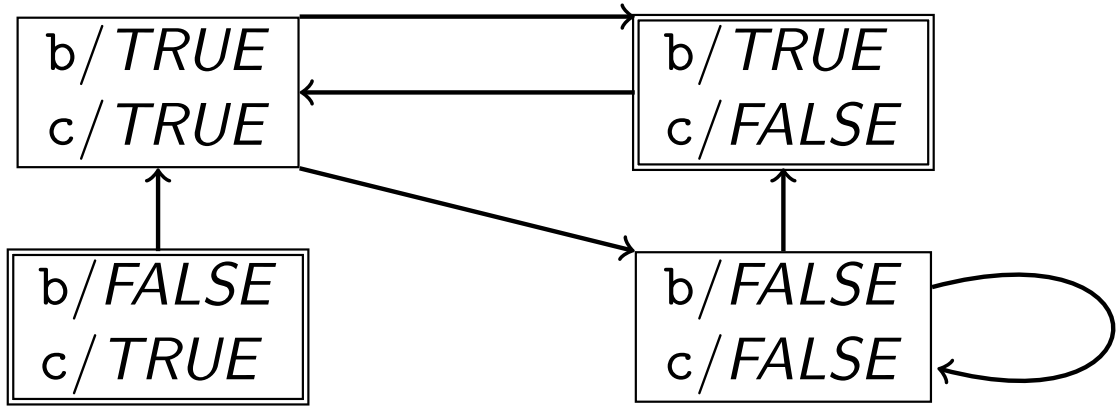
Macros whould be used for *identifiers* which are synonyms for some *expressions* and should not be included in the system state space

(ν) Macros

```
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;
```

Example

$M[main]$ is the following structure:



(ν) Parameterized modules

```
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;
```

A module declaration (*in general case*) can contain parameters — identifier names listed in parentheses after the module name, and separated with commas

A module parameter is similar to a macro:

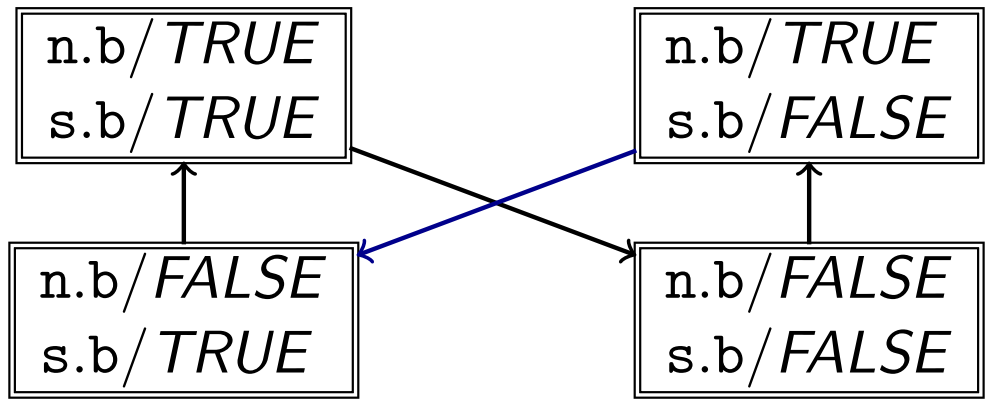
- a macro identifier is a parameter in the module declaration
- a macro expression ia a next-expression written in the corresponding place of an instance declaration

(ν) Parameterized modules

```
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;
```

Example

$M[main]$ is the following structure:



(ν) State invariants

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

Sometimes it is convenient to define a system state space as a set of general restrictions joined with an additional restriction

"exclude all states which do **not** satisfy a formula φ and all transitions connected to these states"

This additional restriction is written as follows:

INVAR φ;

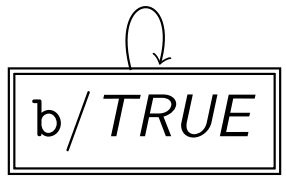
The restriction φ is added to other state space restrictions in two variants: for variables, and for corresponding next-variables

(ν) State invariants

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

Example

$M[main]$ is the following structure:



(ν) Special variables

A frozen variable is declared in the same way as a "usual" variable, but with the `FROZENVAR` keyword instead of `VAR`

A value of a frozen variable is defined in the initial state (e.g. under the `INIT` keyword) and remains unchanged during model execution

More detailed:

- each frozen variable is included into the state space
- usage of next-variables corresponding to frozen variables is forbidden
- for each frozen variable `x` the restriction `TRANS next(x) = x;` is implicitly included in the model

(ν) Special variables

An input variable is declared in the same way as a "usual" variable, but with the `IVAR` keyword instead of `VAR`

Input variable values mark transitions (**not** states) of a model

More detailed:

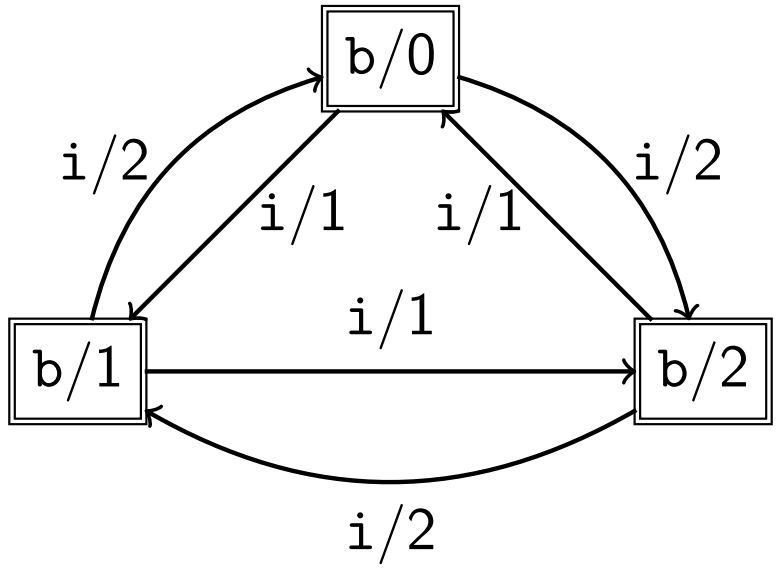
- input variables are **not** included into the state space
- usage of next-variables corresponding to input variables is forbidden, as well as usages not related to transition descriptions
- input variables can be used in `TRANS`-restrictions together with other variables

(ν) Special variables

```
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
```

Example

$M[main]$ is the following structure:



(ν) ASSIGN

Those who prefer "sequential" assignment form can use a unified `ASSIGN`-based restriction syntax:

- `ASSIGN` *`variable`* `:=` *`expression`* equals to
  - `INVAR` *`variable`* `in` *`expression`* if the expression result type is a set
  - `INVAR` *`variable`* `=` *`expression`* otherwise
- `ASSIGN init(`*`variable`*`) :=` *`expression`* equals to
  - `INIT` *`variable`* `in` *`expression`* if the expression result type is a set
  - `INIT` *`variable`* `=` *`expression`* otherwise
- `ASSIGN next(`*`variable`*`) :=` *`next-expression`* equals to
  - `TRANS next(`*`variable`*`) in` *`next-expression`* if the expression result type is a set
  - `TRANS next(`*`variable`*`) =` *`next-expression`* otherwise

(ν) Asynchronous composition

In `NuSMV 2.6.0` still exist (*deprecated*) means for asynchronous module composition based on *interleaving semantics*

In these course these means are allowed but not discussed — those who want to use them should **carefully** read manuals

Asynchronous composition in `NuSMV` can be implemented as a special case of a synchronous one:

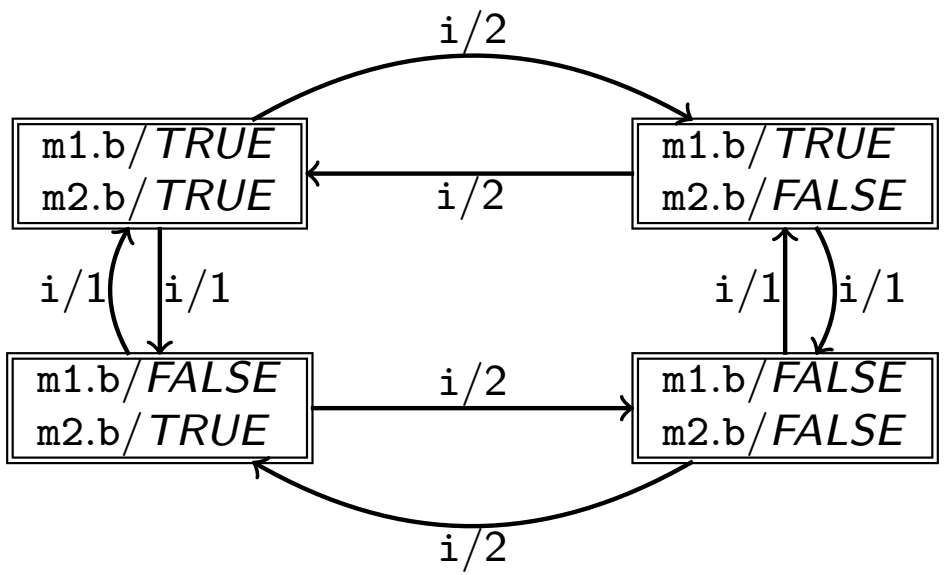
- Each module instance contains a boolean parameter "it is my turn"
- if "my turn" evaluates to `TRUE`, then instance variables should change as intended, otherwise — explicitly stated as unchanged
- an outer instance is used as an arbiter determining turns of inner instances

(ν) Asynchronous composition

```
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;
```

Example

$M[main]$ is the following structure:



(ν) Fairness

Reminder, lecture 4: **fairness constraints**

- divide all Kripke structure paths into fair and unfair
- modify semantics of the quantifiers:
  - $A\varphi$ = "for each **fair** path φ holds"
  - $E\varphi$ = "there exists a **fair** path,
    such that φ holds"

(ν) Fairness

Reminder, lecture 5: "classical" CTL fairness constraints are defined as follows:

- a primitive fairness constraint is a set of Kripke structure states
- a path π is fair w.r.t. a primitive fairness constraint $P \Leftrightarrow$ at least one state of P occurs infinitely often in π
- "fairness constraints" is the set of primitive fairness constraints
- a path π is fair w.r.t. fairness constraints $\mathcal{P} \Leftrightarrow \pi$ is fair w.r.t. each primitive fairness constraint of $\mathcal{P}$

(ν) Fairness

```
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` *formula* `;` is a definition of a "classical" primitive CTL fairness constraint: a path of $M[main]$ is fair $\Leftrightarrow$ a formula of **every** `JUSTICE`-restriction written inside of *main* or its instances evaluates to `TRUE` infinitely often

(ν) Fairness

```
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;
```

Example

$M[main]$ is the same structure as in the previous example, and *main* contains two primitive fairness constraints:

- `JUSTICE turn = 1;`
- `JUSTICE turn = 2;`

Informal meaning: **an unfair path is a path in which one of the processes becomes inactive forever**

(ν) Nontotal Kripke structures

According to previous lectures, a Kripke structure is a *total* graph: each state has at least one outgoing transition

Some verification tools (including `NuSMV`) allow descriptions of **nontotal** Kripke structures

These structures are usually considered **incorrect** except when manuals explicitly state the opposite

In particular, nontotal `NuSMV` models are incorrect unless core verification algorithms are completely changed using some options (*not discussed in the course and not recommended to use*)

To ensure model totality is a developer's problem

(ν) Nontotal Kripke structures

```
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
```