Technische Universität
Kaiserslautern

o1o1seda o1o1oo

software engineering dependability

# Safety and Reliability of Embedded Systems

(Sicherheit und Zuverlässigkeit eingebetteter Systeme)

## Symbolic Model Checking

# Content

- Introduction
- Example: Elevating rotary table
- Excursus: Temporal logic (CTL)
- State space
- Proof of safety requirements
- Excursus: OBDDs
- Encoding of state machines with OBDDs
- Analysis of state machines with OBDDs
- Summary

# Symbolic Model Checking Introduction

- Symbolic model checking is a formal verification technique
  - It is mathematically founded
  - Formal documents are required as a starting point
  - It yields complete statements
  - The results are reliable

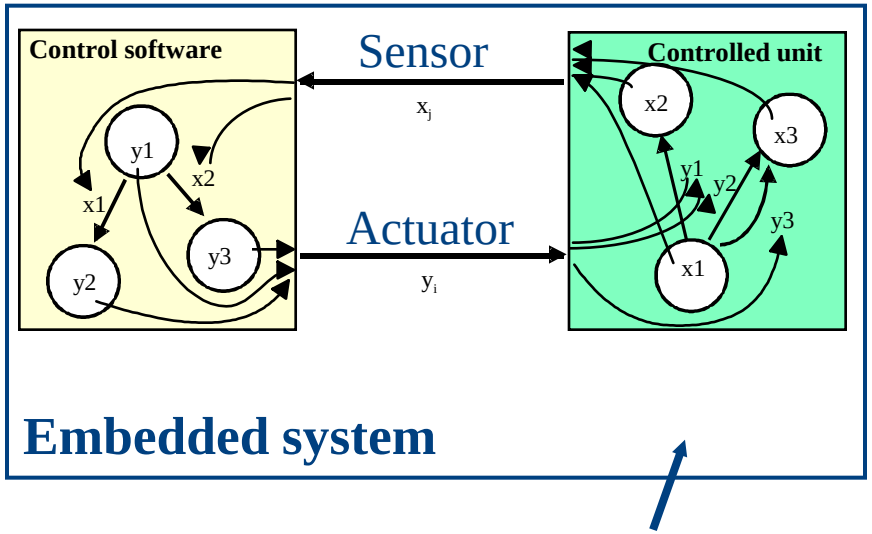
- Aims
  - Verification of defined properties of the item under consideration (software, specification) or ...
  - Generation of information, where the required property does not hold

# Symbolic Model Checking Introduction



- In the following, symbolic model checking will be explained by the verification of properties
  - ... of finite state behavioral descriptions
  - ... against formulas in temporal logic
- Finite state descriptions are interesting because they are common in software engineering and other disciplines (e.g. electrical engineering). Programming languages used in control engineering are normally based on state machines
- Temporal logic is necessary because information about the sequence is required when traversing a state machine

- Finite state control software
- Formal description of the unit under control
- Formal specification of the (safety) requirements in temporal logic (e.g. CTL) for the embedded system which consists of the control software and the unit under control

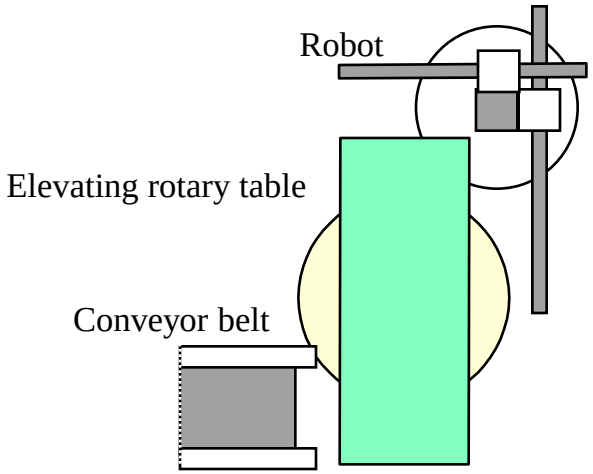


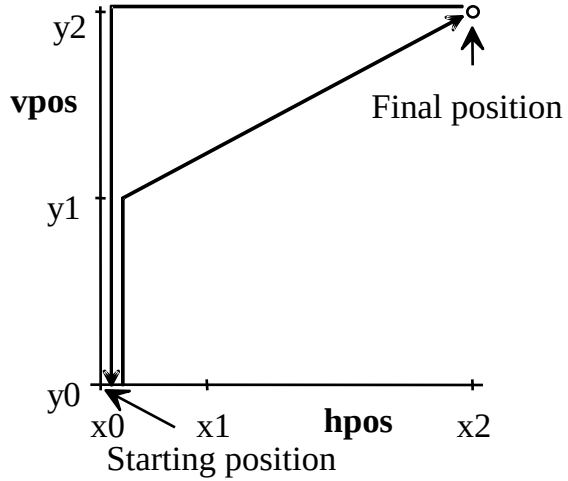
*A manufacturing cell processes raw metal parts, which are fed to a press by a robot. The robot takes the raw part from an elevating rotary table, to which it was delivered by a conveyor belt. The task of the elevating rotary table is to place the raw part, by changing its horizontal and vertical position, in such a way, that it can be grabbed by the robot. The table has two drives (actuators) – one for the horizontal and one for the vertical direction – and 3 sensors – respectively one for the horizontal and one for the vertical position as well as one for the detection of a part on the table.*

| Actuator | Name | Values |
|---|---|---|
| Horizontal movement | *hmov* | *stop, plus, minus* |
| Vertical movement | *vmov* | *stop, up, down* |

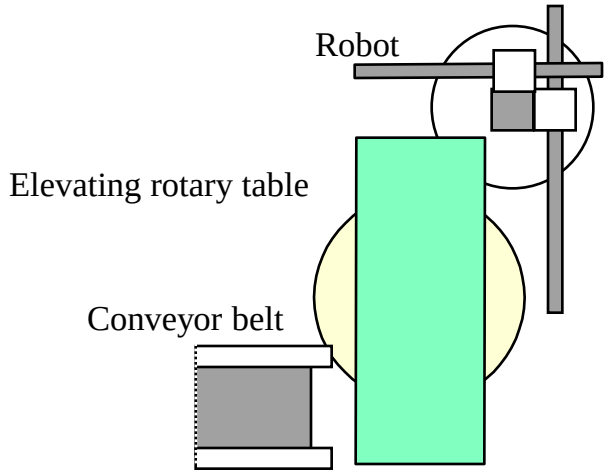
| Sensor | Name | Values |
|---|---|---|
| Horizontal position | *hpos* | *x0, x1, x2* |
| Vertical position | *vpos* | *y0, y1, y2* |
| Part on the table | *part_on_table* | *no, yes* |

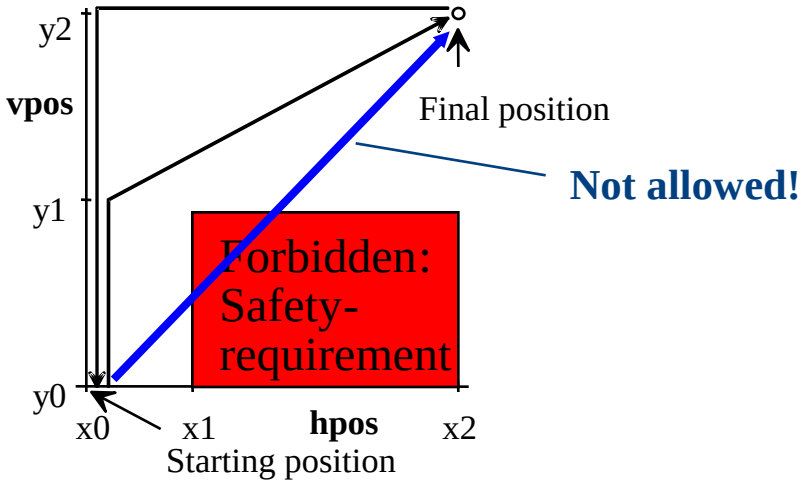
*The starting position of the table is hpos=x0, vpos=y0. All motions have stopped (hmov=stop, vmov=stop). The vertical motion is started (vmov=up) when a part is put on the table (part_on_table=yes). If vpos=y1 is reached, the horizontal motion is started as well (hmov=plus). The robot can grab the part when the table has reached the final position (hpos=x2, vpos=y2). The horizontal motion (hmov=minus) starts when the part has left the table (part_on_table=no) and continues until the position hpos=x0 is reached. The downward motion starts (vmov=down) and the horizontal motion is stopped (hmov=stop). The downward motion is also stopped (vmov=stop) when the table reaches the starting position.*



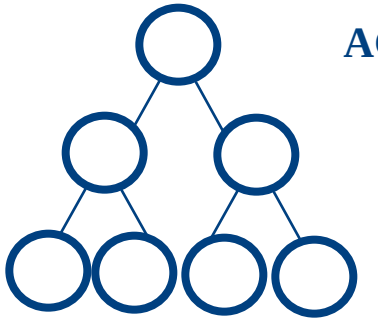
*A safety requirement has to be considered. The table is not allowed to move into the forbidden area. This is ensured, if the states where [vpos=y0 /\ (hpos=x1 \/ hpos=x2)] is true are not reachable. For this reason, during the upward movement the horizontal motion is started at the vertical position y1, and not before. This applies analogously for the downward motion.*



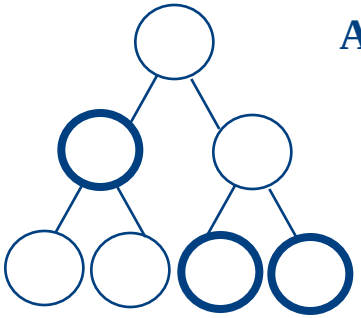
**CTL** (**C**omputation **T**ree **L**ogic)

- Forward-time operators
  - G: **g**lobally, invariantly
  - F: sometime in the **f**uture
  - X: ne**x**t time
  - U: **U**ntil
- Path quantifiers
  - A: **a**ll computation paths
  - E: some computation path (**e**xists)
- In CTL, a *path quantifier* has to be placed right before a *forward-time operator* (e.g. **AG** *f* or **EF** *f)*



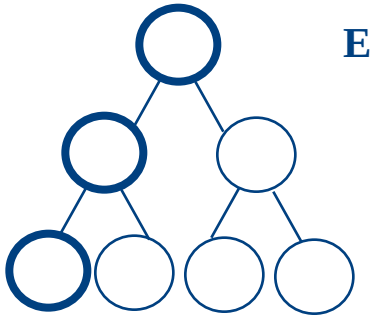
**AG *f***

All states on all paths fulfill *f*



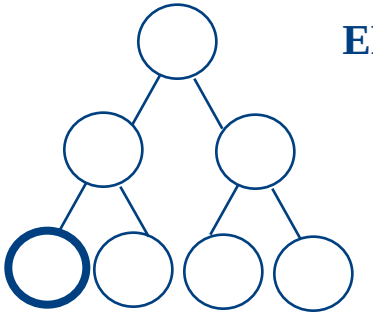
**AF *f***

At least one state on every path fulfills *f*



**EG *f***

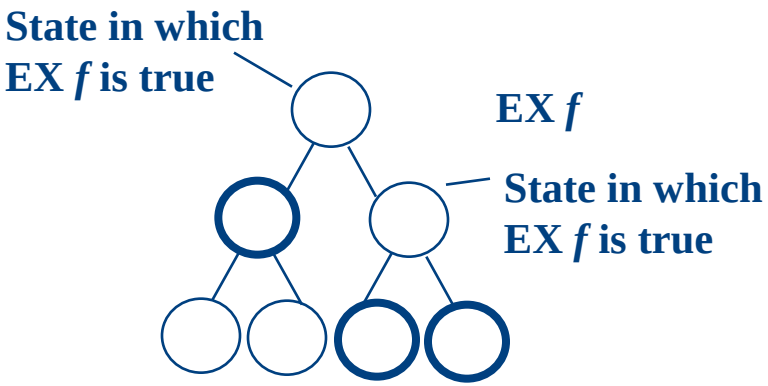
There exists at least one path on which all states fulfill *f*



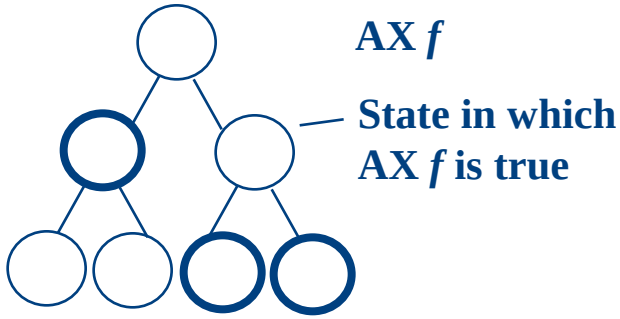
**EF *f***

There exists at least one path on which at least one state fulfills *f*

**State in which *f* is fulfilled**



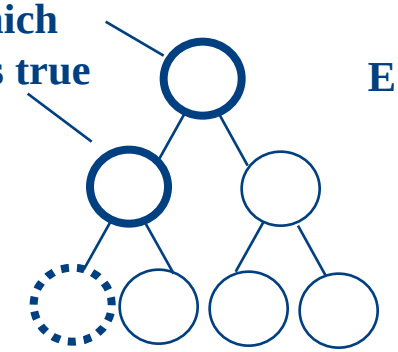

There has to be a directly following state in which *f* is fulfilled




*f* has to be fulfilled in every directly following state

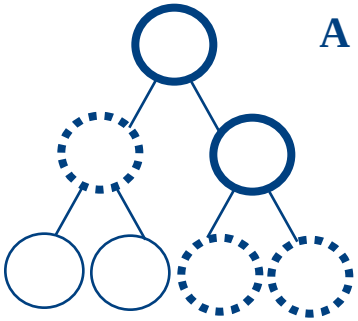
**State in which *f* is fulfilled**

**State in which E [*f* U *g*] is true**



**E [*f* U *g*]**

*f* is true on a path until *g* is fulfilled



**A [*f* U *g*]**

*f* is true along all paths until *g* occurs

**State in which *f* is fulfilled**

**State in which *g* is fulfilled**

- Safety requirement
  - It is not allowed that a state occurs where the following is true

$$\mathbf{[vpos=y0 \wedge (hpos=x1 \vee hpos=x2)]}$$

  - In CTL

$$\mathbf{AG \sim[vpos=y0 \wedge (hpos=x1 \vee hpos=x2)]}$$

or equivalently

$$\mathbf{\sim EF\ [vpos=y0 \wedge (hpos=x1 \vee hpos=x2)]}$$

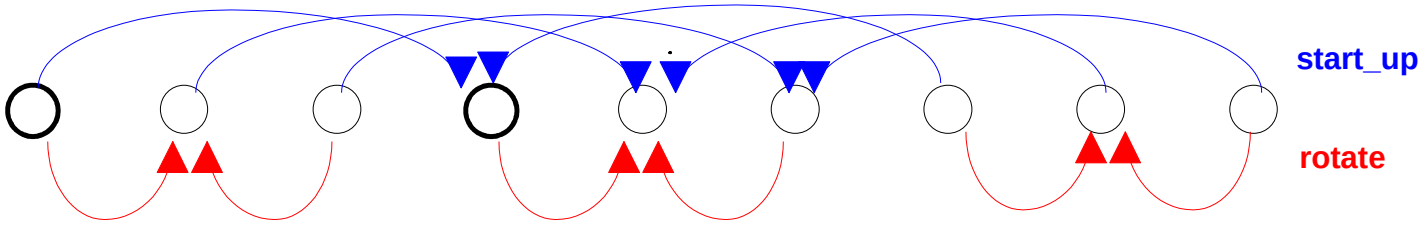
~: Negation

- Control software written in the programming language CSL (Control Specification Language), which is based on state machines

```
StateVariables
        input  vpos : [y0, y1, y2] default y0;
        input  hpos : [x0, x1, x2] default x0;
        input  part_on_table  : [no, yes]  default no;
        output vmov: [stop, up, down] default stop;
        output hmov: [stop, plus, minus] default stop;

Transitions
        start_up:= (part_on_table = yes /\ vpos = y0) ==> (** vmov = up);
        rotate:= (part_on_table = yes /\ vpos = y1 /\ hpos < x2) ==> (** hmov = plus);
        stophigh:= (part_on_table = yes /\ vpos = y2) ==> (** vmov = stop);
        stop_rot:= (part_on_table = yes /\ hpos = x2) ==> (** hmov = stop);
        rot_back:= (part_on_table = no /\ vpos = y2 /\ hpos = x2) ==> (** hmov = minus);
        start_down:= (part_on_table = no /\ hpos = x0 /\ vpos = y2)
                        ==> (** hmov = stop /\ ** vmov =      down);
        stoplow:= (part_on_table = no /\ vpos = y0) ==> (** vmov = stop);
```

- The state space of the controller results from the combination of actuator values. Sensor readings trigger state transitions in the state space
- The transitions *start_up* and *rotate* are depicted in the state space below

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **vmov** | – stop | stop | stop | up | up | up | **down** | down | down |
| **hmov** | _ stop | plus | minus | stop | plus | minus | **stop** | plus | minus |




```
start_up :=  (part_on_table = yes /\ vpos = y0) ==> (** vmov = up);
rotate:=  (part_on_table = yes /\ vpos = y1 /\ hpos < x2) ==> (** hmov = plus);
stophigh :=  (part_on_table = yes /\ vpos = y2)  ==> (** vmov = stop);
stop45 :=  (part_on_table = yes /\ hpos = x2)  ==> (** hmov = stop);
rot_back := (part_on_table = no /\ vpos = y2  /\ hpos = x2) ==> (** hmov = minus);
start_down :=  (part_on_table = no /\ hpos = x0 /\ vpos = y2) ==> (** hmov = stop /\ ** vmov = down);
stoplow :=  (part_on_table = no /\ vpos = y0) ==> (** vmov = stop);
```

- The relevant properties are not defined exclusively for the control automaton and are therefore not verifiable
- It is necessary to take the properties of the controlled system into consideration, i.e. the response of the controlled system to commands from the control software
- These result from in most cases relatively simple physical properties of the controlled system, e.g.
  - *If a drive is stopped the controlled system does not move in this axis*
  - *With opened inflow and closed outflow valve the level of a closed boiler does not decrease*
- The combination of control logic and the response of the controlled system determines the behavior of the whole system

  => A formal description of the behavior of the controlled system is necessary

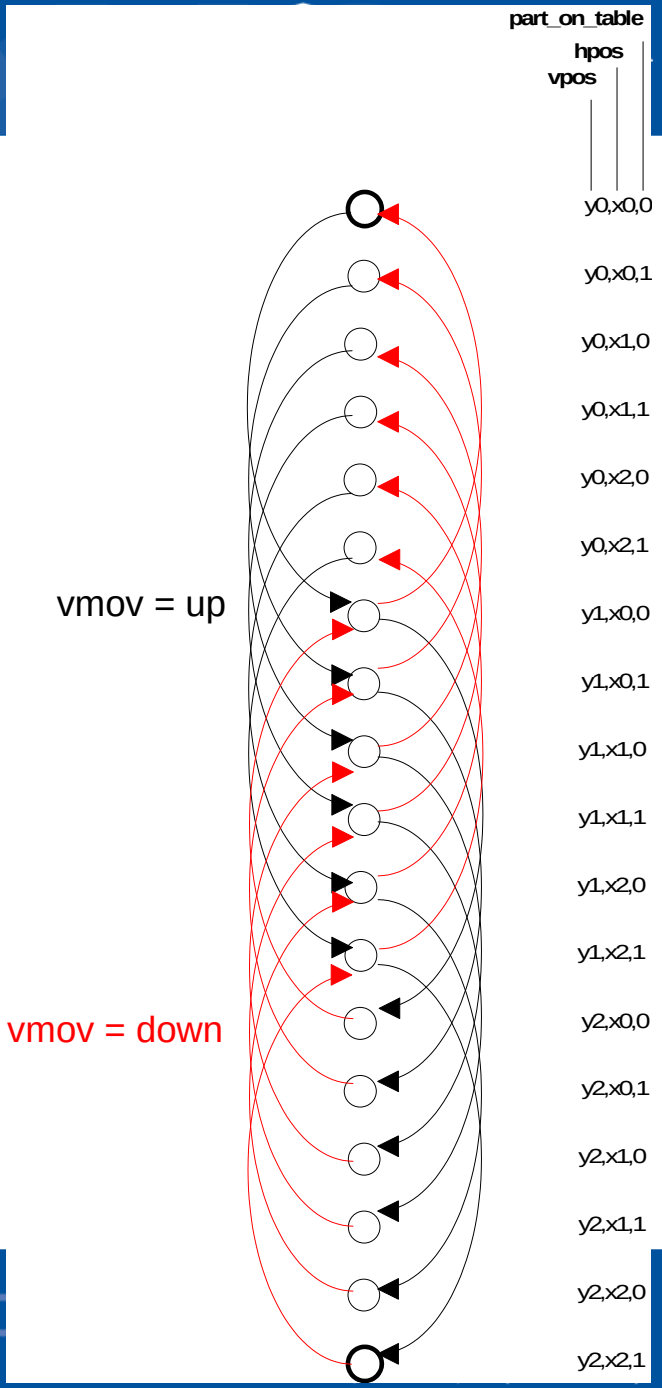
# Symbolic Model Checking
# Example: Elevating rotary table

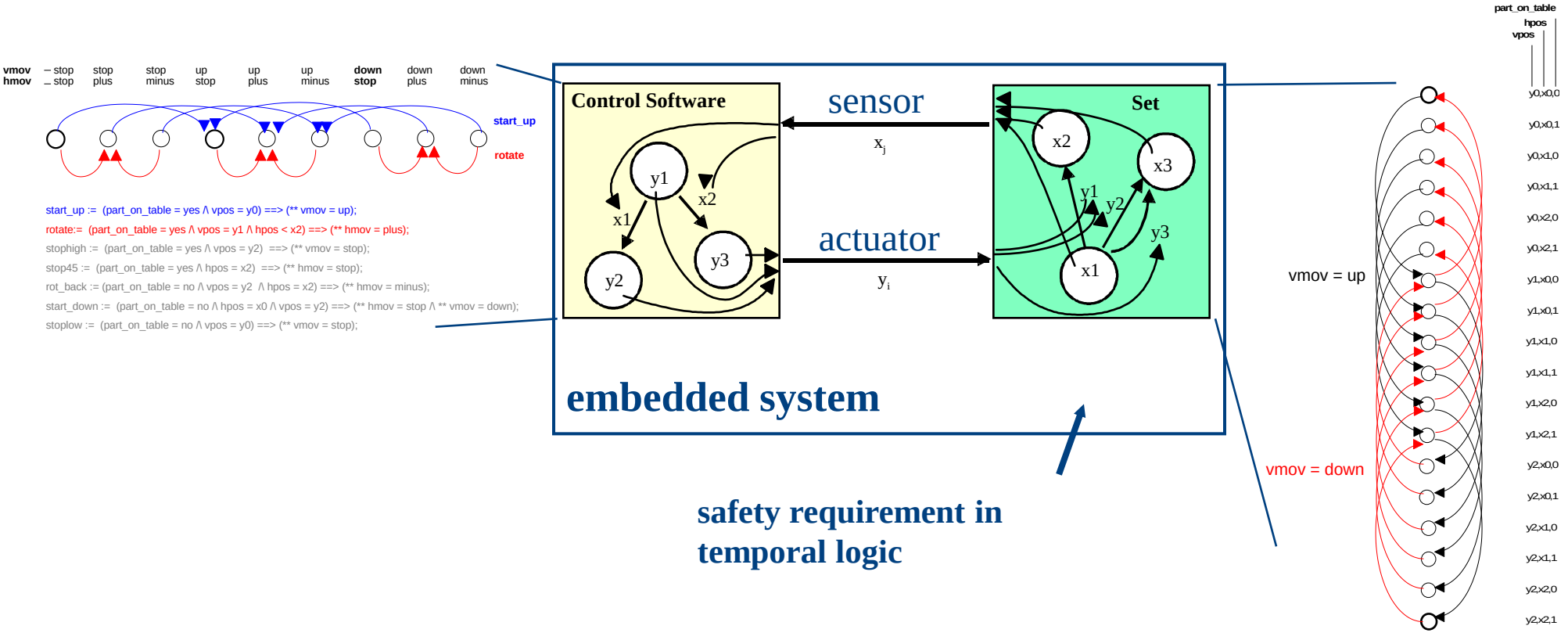
```
allMotorsStop:= ('table.hmov' = stop /\ 'table.vmov' = stop).                    Definitions
initialPosition:= ('table.hpos'=x0) /\ ('table.vpos'=y0).
finalPosition:= ('table.hpos'=x2) /\ ('table.vpos'=y2).
initialState:= initialPosition /\ ('table.part_on_table'=no).
finalState:= finalPosition /\ ('table.part_on_table'=yes).

fairprocess:=
 (
          (  In dependency of the moving direction the vertical positions have to occur in a certain order
           ('table.vmov' = stop /\ 'table.vpos' = y0) /\ x('table.vpos' = y0)
           \/ ('table.vmov' = stop /\ 'table.vpos' = y1) /\ x('table.vpos' = y1)
           \/ ('table.vmov' = stop /\ 'table.vpos' = y2) /\ x('table.vpos' = y2)

           \/ ('table.vmov' = up /\ 'table.vpos' = y0) /\ until('table.vpos' = y0, 'table.vpos' = y1)
           \/ ('table.vmov' = up /\ 'table.vpos' = y1) /\ until('table.vpos' = y1, 'table.vpos' = y2)
           \/ ('table.vmov' = up /\ 'table.vpos' = y2) /\ x('table.vpos' = y2)

           \/ ('table.vmov' = down /\ 'table.vpos' = y0) /\ x('table.vpos' = y0)
           \/ ('table.vmov' = down /\ 'table.vpos' = y1) /\ until('table.vpos' = y1, 'table.vpos' = y0)
           \/ ('table.vmov' = down /\ 'table.vpos' = y2) /\ until('table.vpos' = y2, 'table.vpos' = y1)
         )

...
```

```
 /\ (  In dependency of the moving direction the horizontal positions have to occur in a certain order
      ('table.hmov' = stop /\ 'table.hpos' = x0) /\ x('table.hpos' = x0)
      \/ ('table.hmov' = stop /\ 'table.hpos' = x1) /\ x('table.hpos' = x1)
      \/ ('table.hmov' = stop /\ 'table.hpos' = x2) /\ x('table.hpos' = x2)

      \/ ('table.hmov' = plus /\ 'table.hpos' = x0) /\ until('table.hpos' = x0, 'table.hpos' = x1)
      \/ ('table.hmov' = plus /\ 'table.hpos' = x1) /\ until('table.hpos' = x1, 'table.hpos' = x2)
      \/ ('table.hmov' = plus /\ 'table.hpos' = x2) /\ x('table.hpos' = x2)

      \/ ('table.hmov' = minus /\ 'table.hpos' = x0) /\ x('table.hpos' = x0)
      \/ ('table.hmov' = minus /\ 'table.hpos' = x1) /\ until('table.hpos' = x1, 'table.hpos' = x0)
      \/ ('table.hmov' = minus /\ 'table.hpos' = x2) /\ until('table.hpos' = x2, 'table.hpos' = x1)
    )
  /\ (
      (initialState /\ allMotorsStop /\ x('table.part_on_table' = yes))  Parts appear in the initial state
      \/ (finalState /\ allMotorsStop /\ x('table.part_on_table' = no))  Parts disappear in the final state
      \/ (        There is no change at the availability of the parts in every other state
                ~(initialState /\ allMotorsStop \/ finalState /\ allMotorsStop)
                /\ ('table.part_on_table' = no /\ x('table.part_on_table' = no)  \/ 'table.part_on_table' = yes /\
                                        x('table.part_on_table' = yes))
         )
    )
)
```

# Symbolic Model Checking
## Example: Elevating rotary table – State space

- The state space of the elevating rotary table is defined as the combination of the sensor values. Actuator data is triggering transitions in this state space
- The diagram shows those transitions that represent the correlation between vertical movement and vertical position



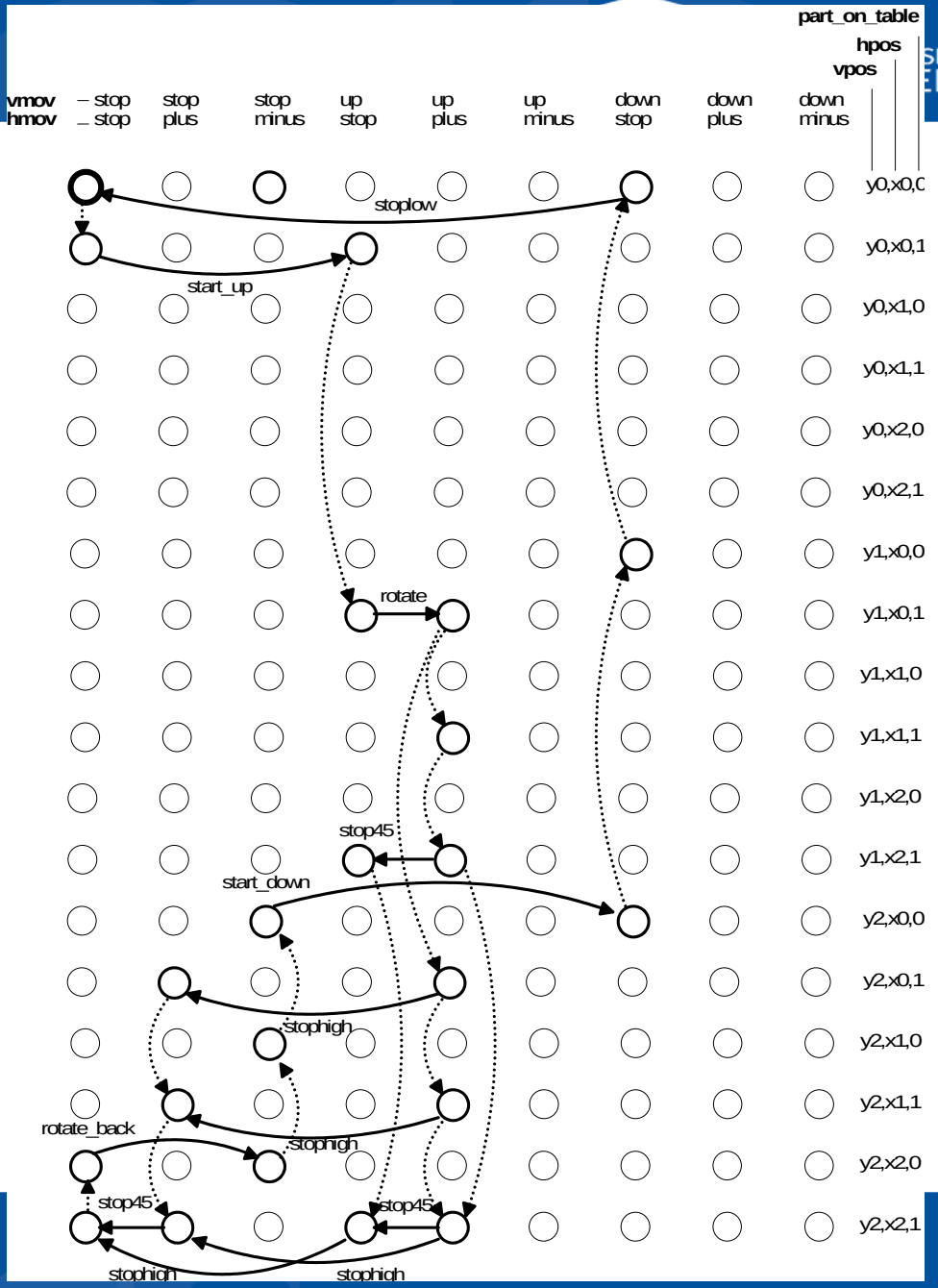



**safety requirement in temporal logic**

- Next Step: Combination of the state machine of the controller and the state machine of the elevating rotary table to a so-called product fsm (finite state machine)

# Symbolic Model Checking
# Example: State space

- The number of states of the product automaton is the product of the number of states of the control automaton and of the automaton of the controlled system
- Here, those transitions are displayed that can actually be passed through in the product automaton. Steps of the controller are drawn as solid lines and steps of the controlled system are represented as dashed lines



# Symbolic Model Checking
# Example: State space

- Safety requirement:

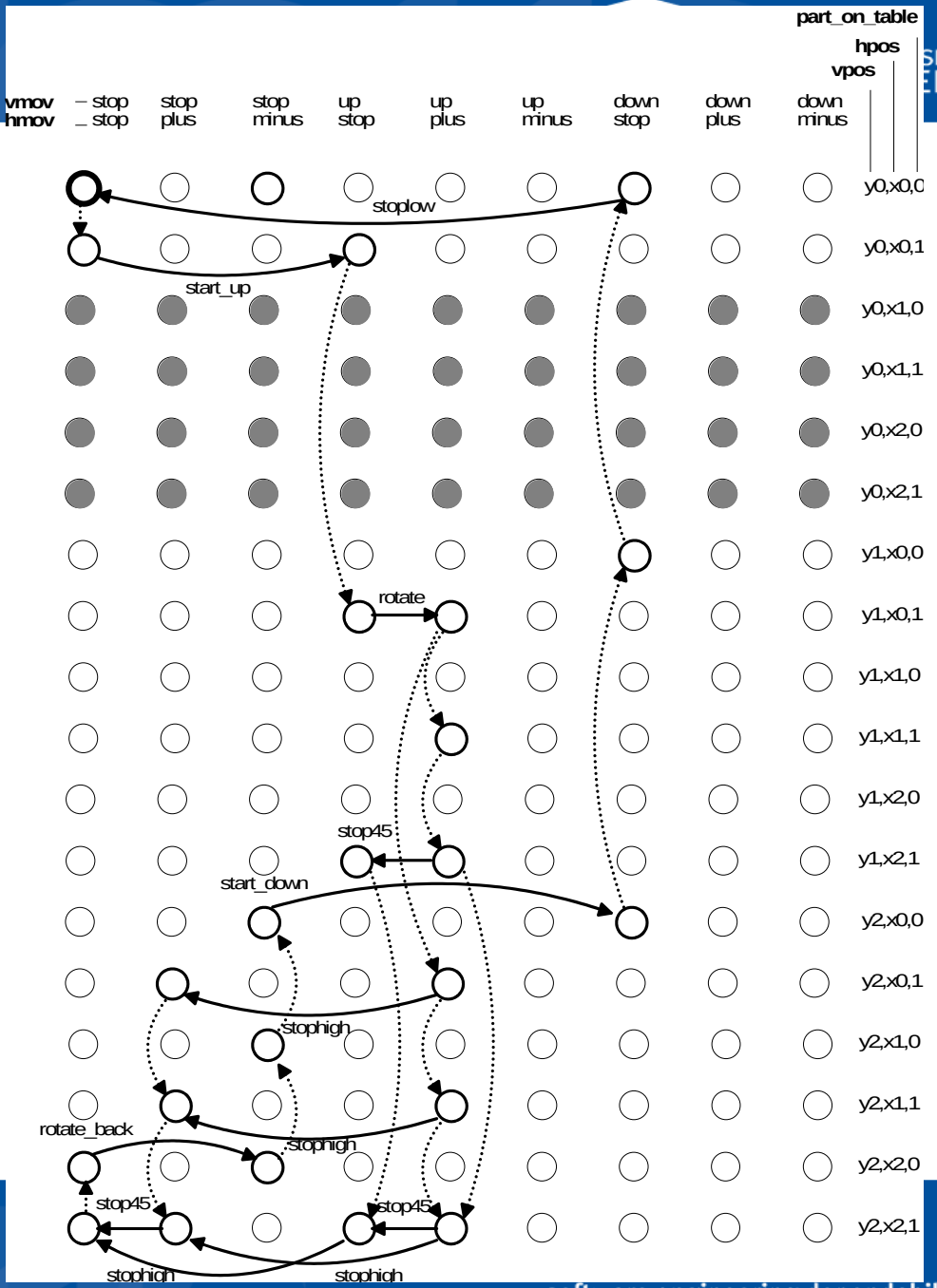
**AG ~[vpos=y0 ∧ (hpos=x1 ∨ hpos=x2)]**

resp.:

**~EF [vpos=y0 ∧ (hpos=x1 ∨ hpos=x2)]**

- States for which [vpos=y0 ∧ (hpos=x1 ∨ hpos=x2)] holds are marked in grey. The safety requirement demands that none of the possible paths contains such a state

=> Reachability analysis



- Safety requirements can often be checked by performing reachability analyses of the state space: “A system is safe, if unsafe states are not reachable.”
- Algorithm:

*Initialize the set of reachable states E with the initial state Z*

*Calculate F as the set of direct successor states of set E.*

$E = E \cup F$

*until* ***E*** *stays unchanged (so-called fix point iteration)*

*Calculate the intersection S of E and the set of the unsafe states* ***U;*** $S = E \cap U$

| $S = \emptyset$ | |
|---|---|
| T | F |
| *system is safe* | *system is unsafe* |
| | *Determine a path from the initial state into an unsafe state as an example for unsafe behavior* |

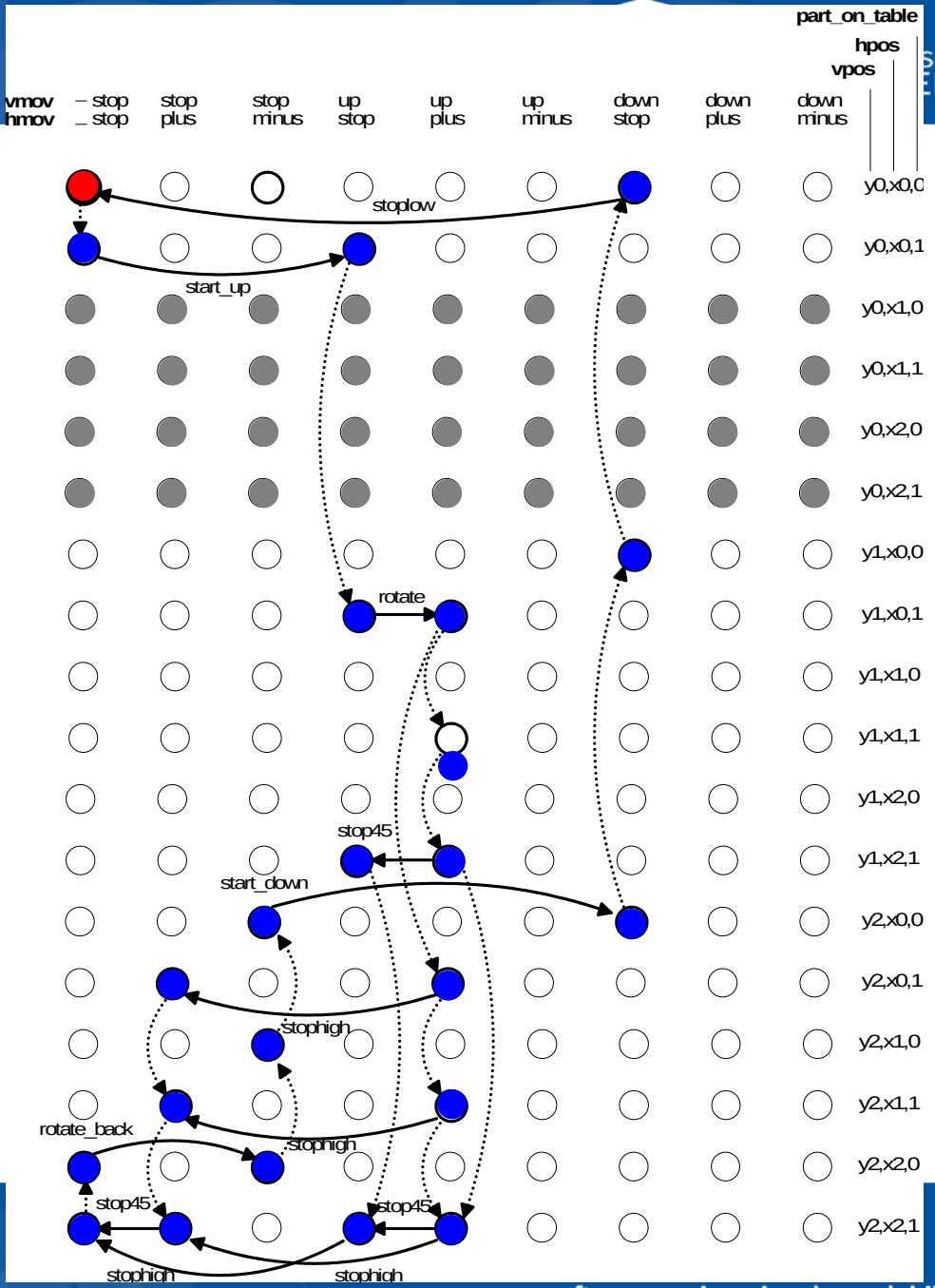
# Symbolic Model Checking
# Proof of safety requirements

- Algorithm for safety analysis:

**E**= {(y0,x0,0,stop,stop); (y0,x0,1,stop,stop);
(y0,x0,1,up,stop); (y1,x0,1,up,stop);
(y1,x0,1,up,plus); (y1,x1,1,up,plus);
(y1,x2,1,up,plus); (y2,x2,1,up,plus);
(y2,x0,1,up,plus); (y2,x1,1,up,plus);
(y1,x2,1,up,stop); (y2,x2,1,up,stop);
(y2,x0,1,stop,plus); (y2,x1,1,stop,plus);
(y2,x2,1,stop,plus); (y2,x2,1,stop,stop);
(y2,x2,0,stop,stop); (y2,x2,0,stop,minus);
(y2,x1,0,stop,minus); (y2,x0,0,stop,minus);
(y2,x0,0,down,stop); (y1,x0,0,down,stop);
(y0,x0,0,down,stop)}

**U**= {(y0,x1,-,-,-); (y0,x2,-,-,-)}

**E** ∩ **U**= ∅ => safety requirement
is fulfilled

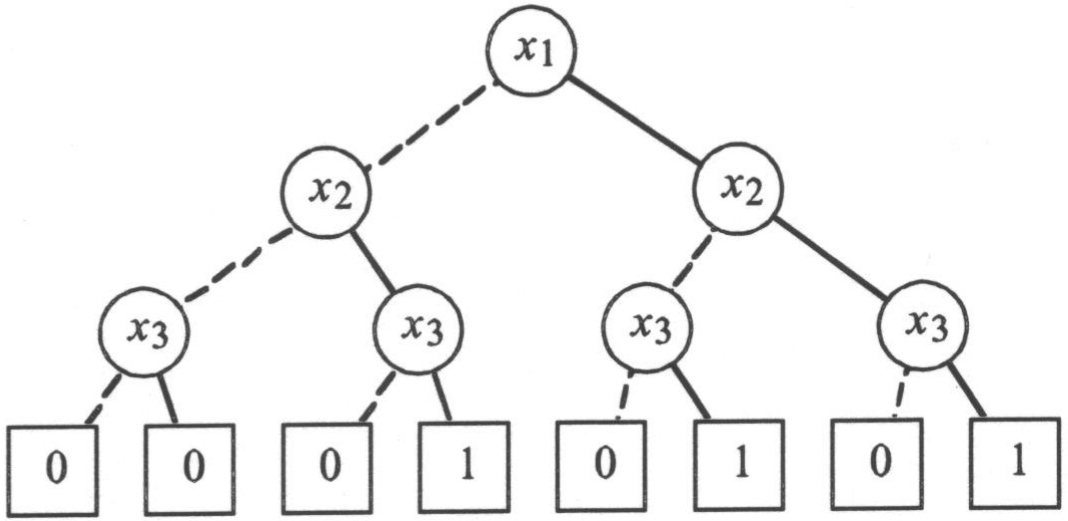


# Symbolic Model Checking
# Efficient implementation

- The size of the state space grows exponentially with the number of state variables => an efficient implementation is necessary in order to apply model checking to real, large systems
- Common implementation of symbolic model checking: Use of so-called OBDDs *(Ordered Binary Decision Diagrams)*
  - Often very compact representation of the so-called characteristic function of those state machines that occur in practical applications
  - Efficient evaluation algorithms

- OBDDs are a notation for the description of Boolean functions
- OBDDs are a so-called canonical representation (with a given variable order)

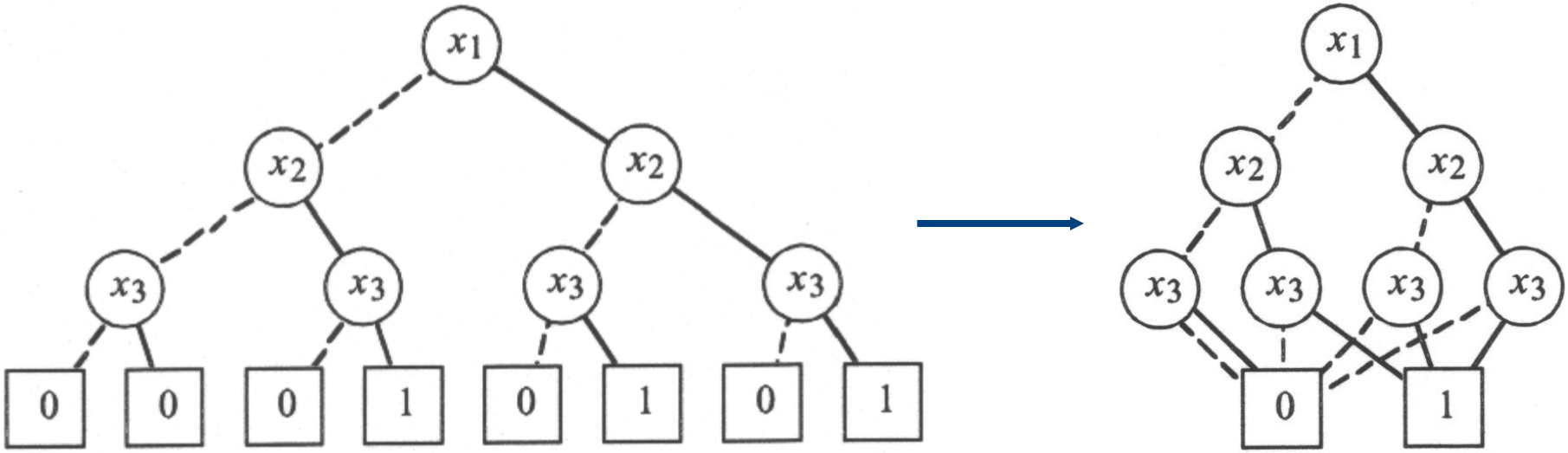
| $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |



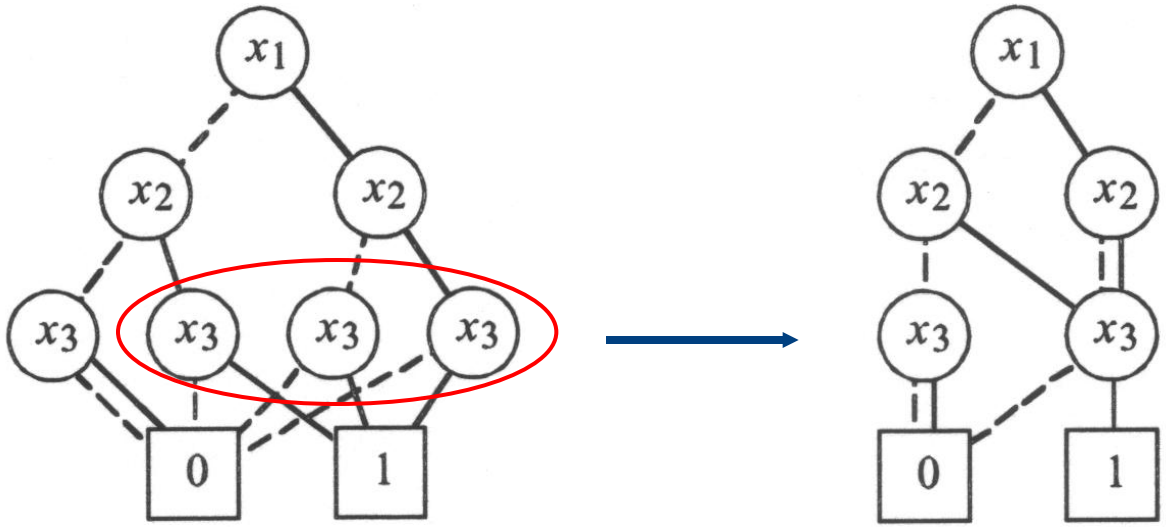

Decision table and decision tree as basis for the generation of an OBDD

0: dashed lines; 1: solid lines

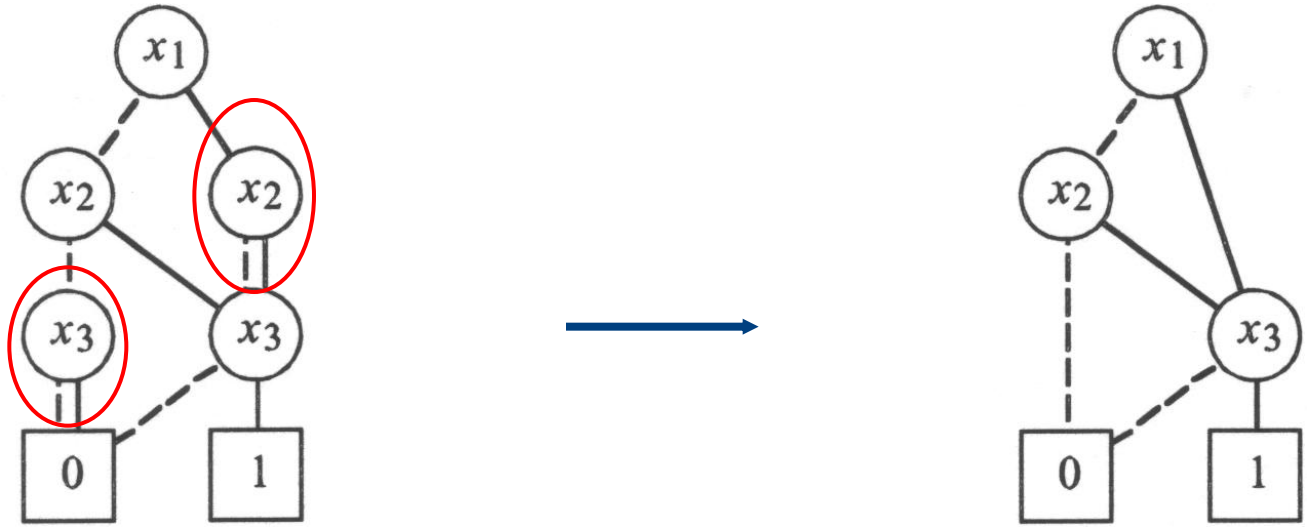
- Removal of redundant terminal nodes



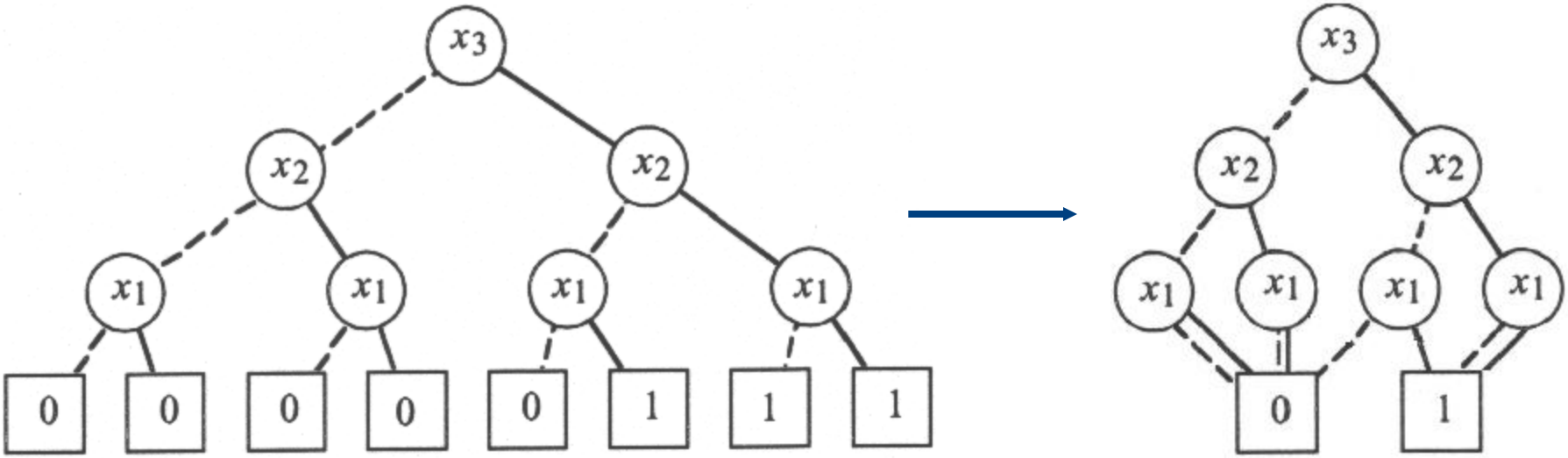
- Removal of identical non-terminal nodes



- Removal of redundant tests



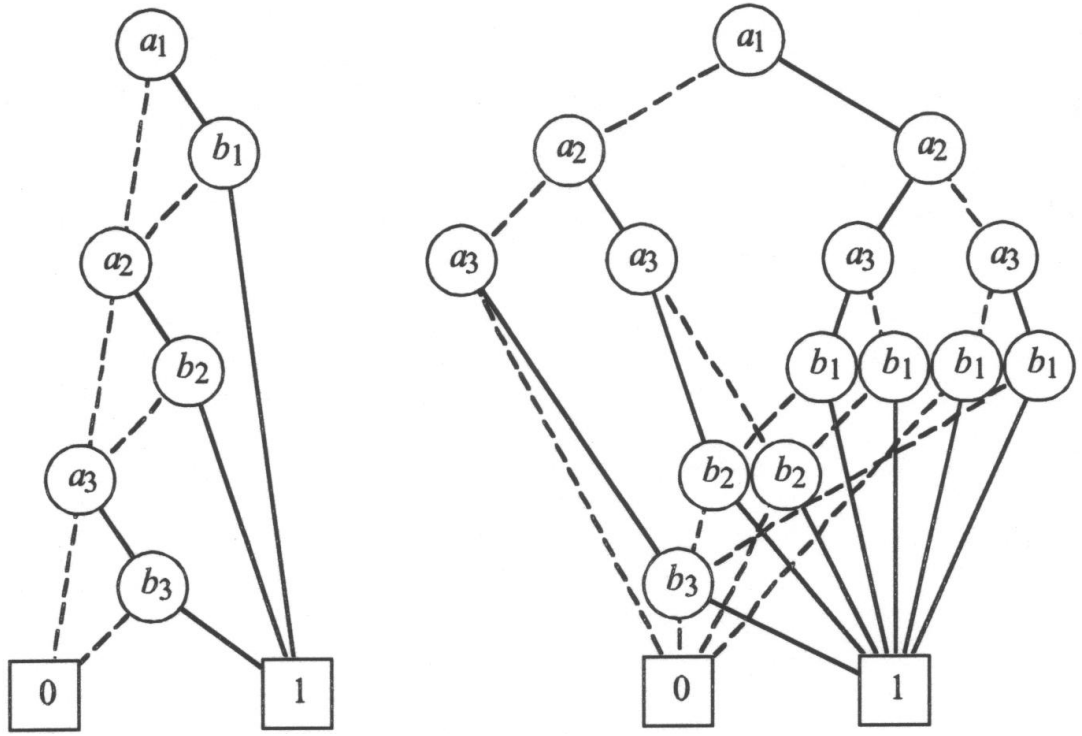
- Same function, different variable order



- Same function, different variable order


- Result: Different OBDD

- The variable order may have considerable influence on the size of the OBDDs
- The function $a_1 b_1 + a_2 b_2 + a_3 b_3$ with two different variable orders



- Representation of the characteristic function of the fsm as an OBDD
- Selection of a binary coding for all state variables and, if necessary, events, e.g.,:

| Y | yb | ya |
|---|---|---|
| y0 | 0 | 0 |
| y1 | 0 | 1 |
| y2 | 1 | 0 |
| - | 1 | 1 |

| X | xb | xa |
|---|---|---|
| x0 | 0 | 0 |
| x1 | 0 | 1 |
| x2 | 1 | 0 |
| - | 1 | 1 |

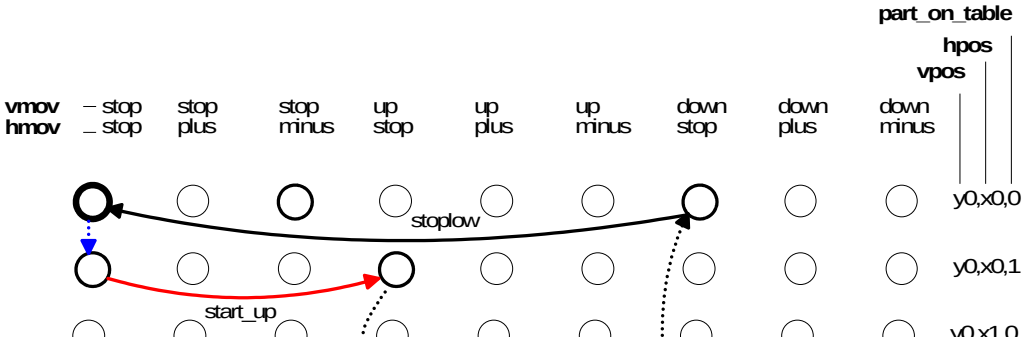
| part_on_table | p |
|---|---|
| no | 0 |
| yes | 1 |

| vmov | vb | va |
|---|---|---|
| stop | 0 | 0 |
| down | 0 | 1 |
| up | 1 | 0 |
| - | 1 | 1 |

| hmov | hb | ha |
|---|---|---|
| stop | 0 | 0 |
| minus | 0 | 1 |
| plus | 1 | 0 |
| - | 1 | 1 |

- Duplication of state variables for the description of present and successor (next) states, e.g. yb, ya (binary coding of the state variable Y in the present state); yb‘, ya‘ (binary coding of the state variable Y in the next state)

- If the state space is coded as described, the characteristic function has the value 1 if there exists a transition between two states under consideration. Otherwise it has the value 0




**Present state** **Next state (')**

(y0,x0,0,stop,stop) -> (y0,x0,1,stop,stop) =>

$$\overline{yb}\ \overline{ya}\ \overline{xb}\ \overline{xa}\ p\ \overline{vb}\ \overline{va}\ \overline{hb}\ \overline{ha}\ \overline{yb'}\ \overline{ya'}\ \overline{xb'}\ \overline{xa'}\ p'\ \overline{vb'}\ \overline{va'}\ \overline{hb'}\ \overline{ha'}$$

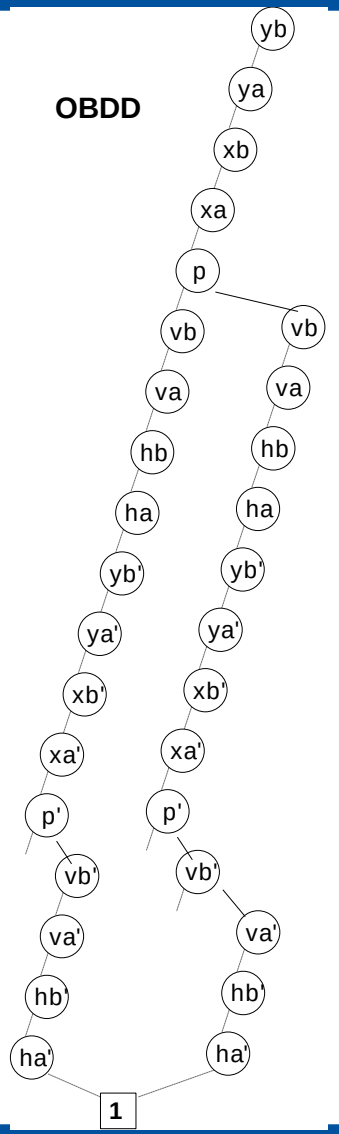
+

(y0,x0,1,stop,stop) -> (y0,x0,1,up,stop) =>

$$\overline{yb}\ \overline{ya}\ \overline{xb}\ \overline{xa}\ p\ \overline{vb}\ \overline{va}\ \overline{hb}\ \overline{ha}\ \overline{yb'}\ \overline{ya'}\ \overline{xb'}\ \overline{xa'}\ p'\ \overline{vb'}\ \overline{va'}\ \overline{hb'}\ \overline{ha'}$$

+

...



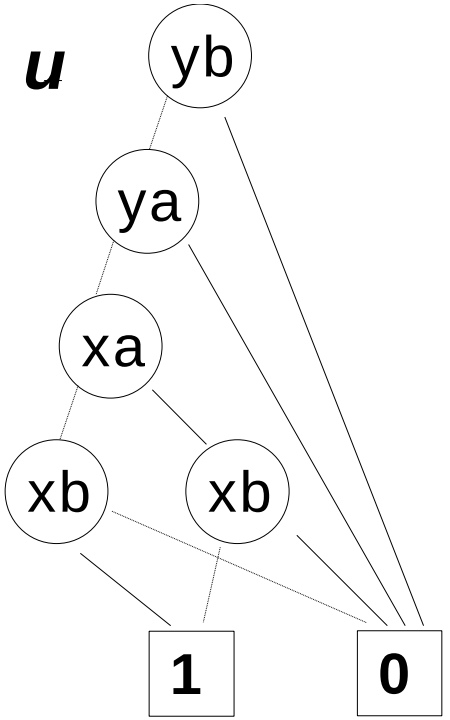
- Efficient analysis algorithms exist for OBDDs, e.g., the apply-function that combines two functions f and g given as OBDDs with an operator *op* to a new function: *f op g*
- The apply-function on OBDDs uses the following rule:

$$f\ \langle op \rangle\ g \quad = \quad \overline{x} \cdot \left( f|_{x \leftarrow 0}\ \langle op \rangle\ g|_{x \leftarrow 0} \right) + x \cdot \left( f|_{x \leftarrow 1}\ \langle op \rangle\ g|_{x \leftarrow 1} \right)$$

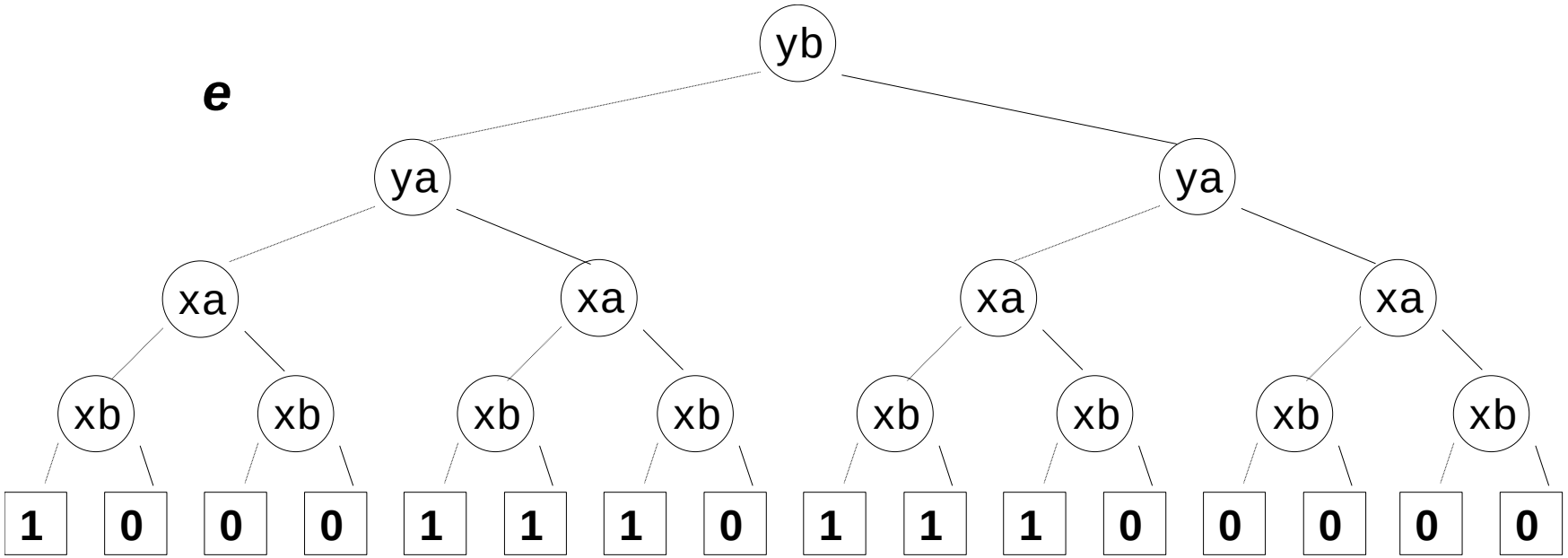
The application of the operation on the OBDDs can thus be recursively applied to the single nodes. If - doing this - a dominant terminal value occurs in one of the combined OBDDs (e.g., 1 with the OR-operator, 0 with the AND-operator), the appropriate terminal value can be directly inserted and the recursion stops

- Example: Checking the safety requirement of the elevating rotary table
  - The intersection of the reachable states and the unsafe states has to be empty, i.e.,
    - if *e* is a Boolean function represented as an OBDD, which has the value 1 for all reachable states **E** and
    - if *u* is a Boolean function represented as an OBDD, which has the value 1 for all unsafe states **U**,
    - (*e* AND *u*) must be reducible to the Boolean constant FALSE
  - i.e., the state is either reachable or unsafe, but not *reachable and unsafe*

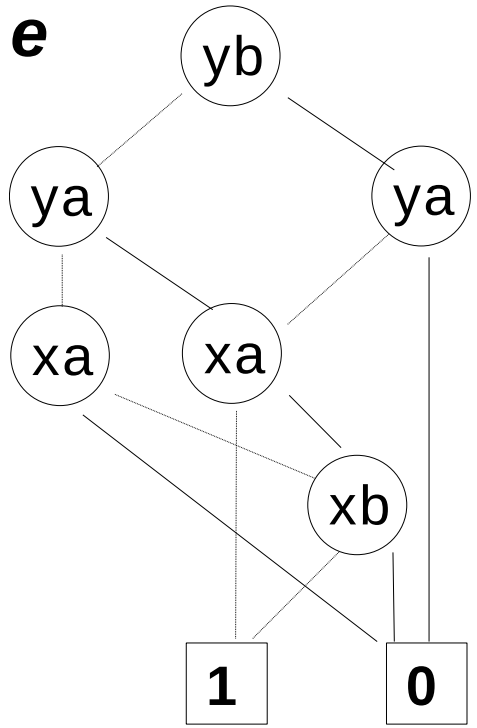
- Function *u* describing the set **U** of unsafe states
  - For unsafe states only position variables are interesting, all other variables can be seen as „don‘t cares“
  - Based on the given binary coding the following OBDD is generated:



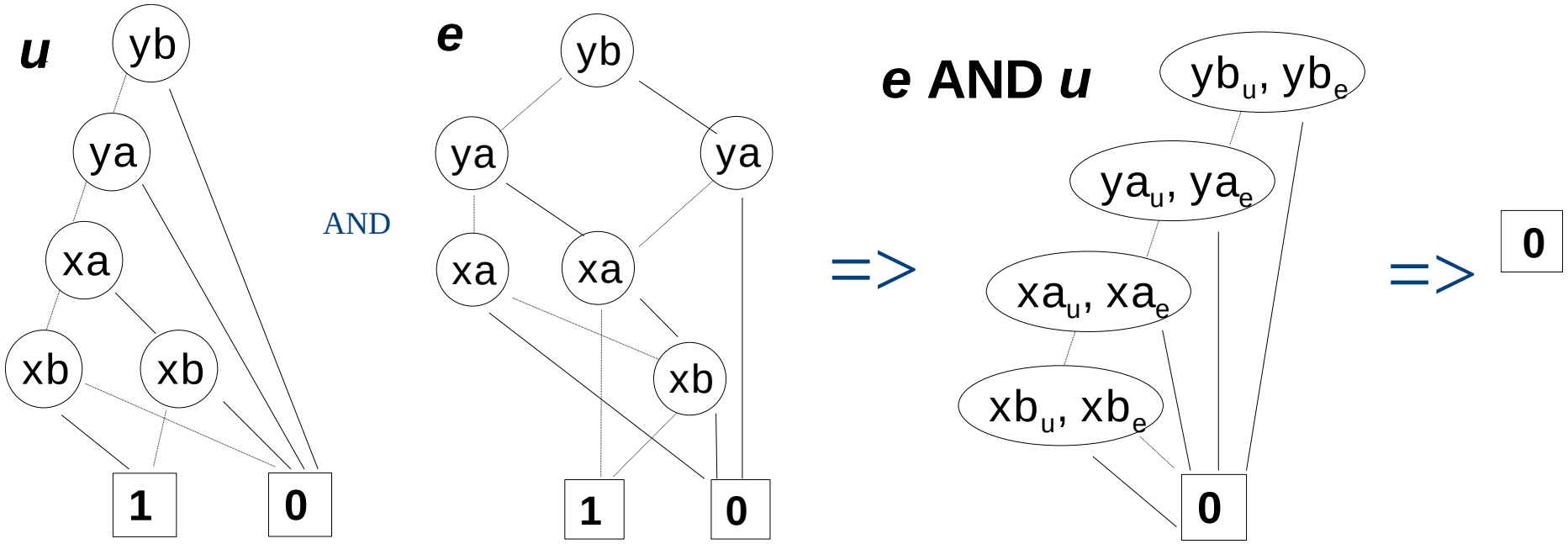
- Function *e* describing the set **E** of reachable states
  - Only position variables must be considered; all other variables are regarded as „don‘t cares“
  - Based on the given binary coding the following tree is generated:



- The following OBDD describes the set **E** (reachable states):



- The AND-operation:




- There are no reachable states that are unsafe, because the AND-operation applied to both sets of states produces the Boolean constant FALSE. The system is safe with regard to the defined safety requirement

# Symbolic Model Checking
# Summary

- Symbolic model checking is a powerful formal technique for proving properties, which requires a finite state description of the behavior
- In many practical applications, OBDDs are appropriate, efficient descriptions of state machines
- Symbolic model checking produces either a validation of required properties or a counter example
- Due to the widespread use of finite state machines in software engineering, the importance of symbolic model checking is growing