Advanced Topics in Software Engineering (02265)

Ekkart Kindler

DTU Compute
Department of Applied Mathematics and Computer Science

Slides 82-151 provide the formalization of the concepts, which however is not presented in the lecture in detail.
V. Formalisation and Analysis
Model checking is a technology for the fully automatic verification of reactive systems with a finite state space.
Validation

requirements

(formal) specification

validation

verification

system

refinement / implementation

or code generation!

model

Ekkart Kindler

ATSE (02265), L08: Formalisation and Analysis
5.2. Main Concepts and Ideas

- Kripke structures (defining the system/model)
- CTL (specifying the properties)
- algorithms (only basic idea)
- complexity
Systems and Requirements

system

meets

requirements
Model und Specification

model $M$

Kripke structure

specification $A$

$\text{Computation Tree Logic (CTL)}$

$\begin{align*}
\text{AG ( } a \Rightarrow AF b \text{ )}
\end{align*}$
A Kripke structure consists of:

- a set of **states**, 
- with distinguished **initial states**, 
- a **total transition relation**, and 
- a **labelling** of states with a set of **atomic propositions**.
The behaviour at a state can be represented as a computation tree:

Note that all paths are infinite!
That is a consequence of the totality of the transition relation.
CTL-Formulas $a, b, \ldots$

\[\wedge, \, \vee, \, \neg, \ldots\]

\[\text{EX} \, , \, \text{EG} \, , \, E[ \, . \, U \, . \, ], \ldots\]

**CTL-formulas** are inductively defined:

- atomic propositions are CTL-formulas
- CTL-formulas combined with a Boolean operator are CTL-formulas
- CTL-formulas combined with temporal operators are CTL-formulas
there exists an (immediate) successor in which \( p \) holds true:
there exists an infinite path on which $p$ holds in each state:
there exists a reachable state in which $b$ holds true, and up to this state $p$ holds true:
Abbreviations

**AX** \( p \equiv \neg \text{EX} \neg p \)
for all immediate successors, \( p \) holds true

**EF** \( p \equiv E [ \text{true U} p ] \)
in some reachable state, \( p \) holds true

**AG** \( p \equiv \neg \text{EF} \neg p \)
in all reachable states, \( p \) holds true

**AF** \( p \equiv \neg \text{EG} \neg p \)
on each path, there exists a state in which \( p \) holds true
A CTL-formula **holds** for a Kripke structure if the formula holds in each initial state.
Example

model $M$                                    specification $p$

$$\begin{array}{c}
\quad \\
\quad
\end{array}$$

$AG ( a \Rightarrow AF b )$

How do we prove it?
Algorithms

For each sub-formula, we inductively calculate the set of states, in which this sub-formula is true:

- atomic propositions
- Boolean operators
- temporal operators
Given:
The set of states in which \( p \) holds: \( S_p \)

Wanted:
The set of states in which \( \mathbf{EX} p \) holds: \( S_{\mathbf{EX} p} \)

We also write \( \mathbf{EX}(S_p) \) for \( S_{\mathbf{EX} p} \)
Algorithm for $E[p \cup q]$

As stated here, this algorithm is quite inefficient. There are more efficient ways to do this.

But, even this inefficient algorithm turns out to be quite efficient when used with the right data structure (ROBDDs, see 5.4).

Given: $S_p$ und $S_q$

Wanted: $S_E[p \cup q]$

\[
\begin{align*}
S_0 &= \emptyset \\
S_1 &= S_q \cup (S_p \cap EX(S_0)) \\
S_2 &= S_q \cup (S_p \cap EX(S_1)) \\
&\quad \vdots \\
S_{i+1} &= S_q \cup (S_p \cap EX(S_i)) \\
\text{until } S_{i+1} &= S_i = S_{E[p \cup q]}
\end{align*}
\]
Algorithm for $\text{EG}_p$

**Given**: $S_p$

**Wanted**: $S_{\text{EG}_p}$

\[
S_0 = S \\
S_1 = S_p \cap \text{EX}(S_0) \\
S_2 = S_p \cap \text{EX}(S_1) \\
\vdots \\
S_{i+1} = S_p \cap \text{EX}(S_i)
\]

**until** $S_{i+1} = S_i = S_{\text{EG}_p}$
Algorithms Summary

CTL model checking ~ marking algorithm + iteration

- $\textbf{EX} \; p$

- $\textbf{E}[ \ p \ U \ q \ ]$

- $\textbf{EG} \; p$
When implemented in an efficient way, the marking algorithm for each operator is linear in the number of states of the system:

$$O(|M| \cdot |p|)$$
When implemented in an efficient way, the marking algorithm for each operator is linear in the number of states of the system:

$$O(|M|)$$
State Space Explosion

- The number of states of a system is exponential in the number of its variables

- Therefore, naive model checking algorithms are doomed to fail in practice:
  - more efficient data structures
  - improved algorithms
  - partial investigation of state space
  - ...
State Space Explosion

The main issue in model checking is:

How to avoid or at least to restrict the negative effect of the state space explosion?
5.3. System model

- Kripke Structures
- Syntactic Representation
- Examples
Kripke Structures

- Motivation
- Definition
- Computation paths
- Transition systems
Motivation

There are many different notations for reactive systems; the choice depends on the application area and the purpose of the model.

Most model checking techniques are independent from the particular notation. Therefore, we do not fix a notation.

Rather we define Kripke structures as a common underlying semantic model.
A Kripke structure $M$ consists of

- a finite set of states: $S$,
- a set of initial states: $S_0 \subseteq S$,
- a total transition relation: $R \subseteq S \times S$,
- a labelling of the states with a set of atomic propositions $AP$: $L: S \rightarrow 2^{AP}$
The set of all paths of $M$ in a state $s$ can be represented as an infinite tree, the **computation tree** of $M$ in $s$:

Example:

Since the transition relation $R$ is total, all paths (branches) of the tree are infinite!
Systems and Kripke Structures

The corresponding Kripke structure

A Petri net

\[ (1,0,1,0) \quad (0,1,1,0) \]

\[ (1,0,0,1) \quad (0,1,0,1) \]
The information on related transitions is lost in the Kripke structure!
Extensions

- Labelling of transitions: **Transition systems**
- Instead of a single transition relation, there are many transition relations (in our example for every Petri net transition).

  This is also important for efficiency reasons!
Syntactic Representation

- Motivation & Example
- States
- Initial states
- Transitions
- Labels
**Formula representation**

- **Boolean variables:**
  \[ V = \{ \ a, \ c \ \} \]

- **Initial formula:**
  \[ S_0 \equiv \neg c \]

- **Transition formula:**
  \[ \mathcal{R} \equiv (a' = \neg a \land c' = c) \lor (a' = a \land c' = \neg c) \]

- **Implicit labelling:**
  \[ AP = V \]

\[ S = \{ (0,0), (0,1), (1,0), (1,1) \} \]

\[ S_0 = \{ (0,0), (1,0) \} \]

\[ R = \{ ((0,0),(1,0)), ((1,0),(0,0)), ((0,1),(1,1)), ((1,1),(0,1)), ((0,0),(0,1)), ((0,1),(0,0)), ((1,0),(1,1)), ((1,1),(1,0)) \} \]
As a Transition System

- **Boolean variables:**
  \[ V = \{a, c\} \]

- **Initial formula:**
  \[ S_0 \equiv \neg c \]

- **Transition formula:**
  \[ T \equiv \{ (a' = \neg a \land c' = c), (a' = a \land c' = \neg c) \} \]

**Implicit labelling:**
\[ AP = V \]

This equality is often implicit for variables that do not occur “primed”.

For example in MCiE (important for efficiency).
More Examples

In this section, we show by the help of two examples how to represent different kinds of systems as Kripke structures represented by formulas.

- Synchronous circuit (hardware)
- Concurrent processes
- Petri nets
Combinatorial Circuit

\[
b' = (a \land b \lor \lnot c) \\
c' = \lnot c
\]
Sequential Synchronous Circuit

\[
(b' = (a \land b \lor \neg c)) \land (c' = \neg c) \land (a' = 0 \lor a' = 1)
\]
Concurrent Processes

\[
\text{loop forever}
\]
\[
pca = 0 \quad x := 0; \\
pca = 1 \quad y := 0; \\
        \quad \quad \quad \text{||}
\]
\[
pcb = 0 \quad x := 1; \\
pcb = 1 \quad y := 1;
\]

\[
(pca = 0 \land pca' = 1 \land x' = 0 \land y' = y \land pcb' = pcb) \lor \\
(pca = 1 \land pca' = 0 \land y' = 0 \land x' = x \land pcb' = pcb) \lor \\
(pcb = 0 \land pcb' = 1 \land x' = 1 \land y' = y \land pca' = pca) \lor \\
(pcb = 1 \land pcb' = 0 \land y' = 1 \land x' = x \land pca' = pca)
\]
Petri nets

How do the formulas look for Petri nets as a transition system?
5.4 ROBDDs

- Motivation
- Definition
- Operations on ROBDDs
- Quantified Boolean formulas (QBF)

Reduced Ordered Binary Decision Diagrams; for simplicity often just called Binary Decision Diagrams (BDDs).
Motivation

- The number of states of realistic systems is gigantic.

⇒ Representing sets of states by enumerating every state explicitly is a bad idea.

- Sets could be represented “symbolically”, e.g. by formulas (see next slide)
Sets as formulas

- $\neg c$
- $a$
- $a \leftrightarrow c$
- $\neg a \land c$

Boolean formulas representing sets of states
Some operations on sets can be efficiently executed for sets that are represented as formulas:

- union: \( p \lor q \)
- disjunction: \( p \land q \)
- complement: \( \neg p \)
- set difference: \( p \land \neg q \)

Problem:

- the same set can have different representations
- it is extremely inefficient to find out whether two formulas represent the same set (NP-complete).
- therefore, formulas are not a good representation for sets of states.

Checking for equality of sets is a very crucial operation in model checking! (BTW: why?) → slide 19/20 (78)
Goal

- Representation of sets such that
  - set operations and
  - check for equality

The answer will be Reduced Ordered Binary Decision Diagrams (ROBDDs)!

ATSE (02265), L09: Formalisation and Analysis (cntd.)
Binary Decision Trees

\[ a = b \land c = d \]

Problem: These will also be very big!
Identify same sub-trees

\[ a = b \land c = d \]
Identify same sub-trees

\[ a=b \land c=d \]
Identify same sub-trees

\[ a = b \land c = d \]

\[ a = b \land c = d \]
Identify same sub-trees

\[ a = b \land c = d \]
Delete redundant nodes

\[ a = b \land c = d \]
Delete redundant nodes

\[ a = b \land c = d \]
Identify same sub-trees

\[ a=b \land c=d \]
Identify same sub-trees

\[ a = b \land c = d \]
\[ a = b \land c = d \]
"Prettified result": ROBDD

\[ a = b \land c = d \]

- All variables on the paths occur in the same order (we had that from the start)
- No identical sub-graphs anymore
- No redundant nodes anymore

\[ \Rightarrow \text{Reduced Ordered Binary Decision Diagram} \]
Observations

- For every set (and a fixed variable order) there exists exactly one ROBDD representing it!

- For many practically relevant sets, the ROBDDs representing them are small.

- The size of the ROBDDs depends on the chosen variable order (on the paths):
  
  For example, the ROBDD for the set characterized by \( a=b \land c=d \) is small with variable order \( a < b < c < d \); it is bigger with variable order \( a < c < d < b \).
Observations

- There are sets for which the ROBDD will be big for any variable order (multiplication)

- Finding good or even optimal variable orders is one of the challenges of symbolic model checking

- There is no efficient way to find an optimal variable order in general (results from complexity theory)

- But, there are heuristics:
  - Variables that are „somehow related“ should be close to each other
  - Local optimisations by switching two variables
Question

- How do we generate an ROBDD?

Answer: Start with full tree and reduce it!
Question

- How do we generate an ROBDD?

Answer: Start with full tree and reduce it!

This is a very bad idea!

Rather, we build them bottom up from formulas with operations on ROBDDs.
Operationen in ROBBDs

- Boolean variable
- Negation
- Restriction and Shannon expansion
- Binary operations
- ROBDDs and Kripke structures
Boolean variable

The set represented by variable $a$ is represented by the ROBBD:
Negation

Complexity: $O(1)$

In practice, this is done a bit differently. Existing ROBDDs are never changed!
Restriction in ROBDDs

- For a ROBDD representing a Boolean function $p$, the ROBDD for the $p|_v \leftarrow t$ can be obtained as follows:

- Subsequently: systematic reduction of the resulting ROBDD.

Remember:
Existing ROBDDs are never changed!

In practice, this is done a bit different.

Complexity:
$O(|p|)$

Size of the ROBDDs for $p$

Complexity:
$O(|p| \cdot \log (|p|))$
An important special case is the restriction to the first variable $v_1$ of the ROBDD:

$p|_{v_1} \leftarrow 0 \text{ bzw. } p|_{v_1} \leftarrow 1$

Compexity: $O(1)$
The binary Boolean operations can be formulated recursively by the help of the Shannon expansion:

- \( p \land q = \)  
  \[
  (\neg v \land (p|_v \leftarrow 0 \land q|_v \leftarrow 0)) \lor \\
  (v \land (p|_v \leftarrow 1 \land q|_v \leftarrow 1))
  \]

- \( p \lor q = \)  
  \[
  (\neg v \land (p|_v \leftarrow 0 \lor q|_v \leftarrow 0)) \lor \\
  (v \land (p|_v \leftarrow 1 \lor q|_v \leftarrow 1))
  \]

- \( p \circ q = \)  
  \[
  (\neg v \land (p|_v \leftarrow 0 \circ q|_v \leftarrow 0)) \lor \\
  (v \land (p|_v \leftarrow 1 \circ q|_v \leftarrow 1))
  \]

This works for arbitrary binary operators!
Binary Boolean operations

ROBDD for $p \oplus q$ from ROBDDs for $p$ and $q$:

- Generate ROBDDs for $p|_v \leftarrow 0$, $q|_v \leftarrow 0$, $p|_v \leftarrow 1$, and $q|_v \leftarrow 1$
- Construct recursively $p|_v \leftarrow 0 \oplus q|_v \leftarrow 0$ and $p|_v \leftarrow 1 \oplus q|_v \leftarrow 1$
- The OBDD for $p \oplus q$ is:

- Reduce the OBDD systematically to an ROBDD.

Overall complexity (if cleverly implemented): $O(|p| \cdot |q|)$
ROBDDs: Summary

- As long as all involved ROBDDs remain small, all operations on ROBDDs are efficient.

- There are many libraries implementing ROBDDs and the operations on them (often with clever algorithms for optimizing the variable order). MCiE is a very simple implementation.

- In practice, all ROBDDs in the same context are maintained in a single data structure (as a „forest“ of ROBDDs and hash tables for avoiding duplicate nodes). Then, equality of ROBDDs can be decided in constant time (same pointer).
Quantified Boolean formulas

- For model checking, we need **Boolean formulas** with quantification of Boolean variables \( \nu \) (QBF):
  \[ \exists \nu \cdot p \]

- \( \exists \nu \cdot p \) is just an abbreviation for \( p|_\nu \leftarrow 0 \lor p|_\nu \leftarrow 1 \)

- \( \exists \nu \cdot p \) is an abbreviation for
  \[ \exists \nu_1 \cdot ( \exists \nu_2 \cdot ( \ldots ( \exists \nu_n \cdot p ) \ldots ) ) \]

- Respectively, \( \forall \nu \cdot p \) stands for \( p|_\nu \leftarrow 0 \land p|_\nu \leftarrow 1 \)

- And \( \forall \nu \cdot p \) stands for
  \[ \forall \nu_1 \cdot ( \forall \nu_2 \cdot ( \ldots ( \forall \nu_n \cdot p ) \ldots ) ) \]
Relation product

- For a formula, \( p(u, v) \) over variables \( U \) and \( V \) and a formula \( q(v, w) \) over variables \( V \) and \( W \), we call

\[ \exists v . p(u, v) \land q(v, w) \]

the relation product of \( p(u, v) \) and \( q(v, w) \).

- The ROBDD for the relation product can be realized with the above abbreviations by the Boolean operations. That, however, is a bit inefficient.

- In practice, the relation product is implemented directly. The worst case complexity is exponential; but, it works reasonably well in many practical setting.
5.5 Symbolic model checking

Represent everything, i.e. initial condition, transition relation as well as the result, as ROBDDs:

**Given:**
- $S_0$ and $\mathcal{R}$ as ROBDDs over $V$ resp. $V \cup V'$
- a CTL-Formula $p$.

**Wanted:**
- The ROBDD for the set of states $S_p$ (the set of states in which $p$ is true).
Algorithms for CTL

- We assume that we have calculated the ROBDDs for the sets $S_p$ and $S_q$ already

- Next we give the algorithms for calculating the ROBDDs for the sets
  - $S_p \lor q$, $S_p \land q$ and $S_{\neg p}$,
  - $S_{EX\ p}$,
  - $S_{EG\ p}$ and
  - $S_{E[p \cup q]}$

These are the Boolean operations.

Algorithms on the following slides!
Algorithm for $\mathbf{EX} \ p$

Observation:

- $\mathbf{EX} \ p \equiv \exists v’. \ R(v, v’) \land p(v’)$

- Given answer in ROBDD

- $p(v)$ given as ROBDD

Relation product on ROBDDs
The only thing left to do is to produce an ROBDD for $p(v')$ from an ROBDD for $p(v)$:

In practice, this renaming is done on the fly (and only temporarily) when the relation product is calculated.
Reminder: $E[p \cup q]$

Given: $S_p$ and $S_q$

Wanted: $S_{E[p \cup q]}$

\[
S_0 = S_q \\
S_1 = S_q \cup (S_p \cap \text{EX}(S_0)) \\
S_2 = S_q \cup (S_p \cap \text{EX}(S_1)) \\
\ldots \\
S_{i+1} = S_q \cup (S_p \cap \text{EX}(S_i)) \\
\text{until } S_{i+1} = S_i = S_{E[p \cup q]}
\]
Algorithm for $E[p \cup q]$

- In this algorithm, the following operations on sets (ROBDDs) occur:
  - test for equality
  - union
  - intersection
  - $\text{EX}(S)$

- For all these operations, we have algorithms already (more or less efficient)

- If the iteration does not change anything (check for equality), this is the ROBDD for $S_{E[p \cup q]}$. 
Procedure \texttt{checkEU}(S_p, S_q)

\begin{align*}
S &:= S_p; \quad // \text{represented as ROBDD} \\
\text{repeat} \\
& \quad S' := S; \\
& \quad S := S_q \lor (S_p \land \text{checkEX}(S)); \\
\text{until} \quad S = S'; \\
\text{return} \quad S;
\end{align*}
Reminder: \( \mathbf{EG} \ p \)

**Given:** \( S_p \)

**Wanted:** \( S_{\mathbf{EG} \ p} \)

\[
S_0 = S_p \\
S_1 = S_p \cap \mathbf{EX}(S_0) \\
S_2 = S_p \cap \mathbf{EX}(S_1) \\
\vdots \\
S_{i+1} = S_p \cap \mathbf{EX}(S_i) \\
\text{until } S_{i+1} = S_i = S_{\mathbf{EG} \ p}
\]

This is the inefficient algorithm from the introduction.

With the help of ROBDDs it becomes reasonably efficient.
Procedure \texttt{checkEX}(S_{p})

\begin{align*}
S &:= S_{p}; \quad // \text{represented as ROBDD} \\
\text{repeat} & \\
\quad S' &:= S; \\
\quad S &:= S_{p} \land \text{checkEX}(S); \\
\text{until} & \quad S = S' \\
\text{return} & \quad S;
\end{align*}

Check for equality

ROBDD operation

procedure for EX(S)
Symbolic model checking

- The use of ROBDDs for the representation of sets of states is called **symbolic model checking** (as in contrast to explicit model checking).
- Symbolic model checking contributed to the initial success of model checking (SMV and today NuSMV)!
- Though it uses more inefficient algorithms as one would use with explicit sets, symbolic model checking is sometimes more efficient (but that depends!).
- It does not work always (for bigger examples).
- There are many other techniques for model checking!
- To date, applying model checking for realistic systems requires much experience.
The following slides are covering the mathematical formalisation and some additional details; The are not shown in the lecture, but are included For completness sake.
5.3. System model (details)

- Kripke Structures
- Syntactic Representation
- Examples

Rather, we build them bottom up from formulas with operations on ROBDDs.
Kripke Structures

- Motivation
- Definition
- Computation paths
- Transition systems
Motivation

There are many different notations for reactive systems; the choice depends on the application area and the purpose of the model.

Most model checking techniques are independent from the particular notation. Therefore, we do not fix a notation.

Rather we define Kripke structures as a common underlying semantic model.
Kripke Structures

A Kripke structure $M$ consists of

- a finite set of states: $S$,
- a set of initial states: $S_0 \subseteq S$,
- a total transition relation: $R \subseteq S \times S$

- a labelling of the states with a set of atomic propositions $AP$: $L: S \rightarrow 2^{AP}$
Kripke Structures

We call $M = (S, S_0, R, L)$ a Kripke structure over the atomic propositions $AP$.

We say that

- proposition $a \in AP$ is valid in a state $s \in S$, if $a \in L(s)$, i.e. if $a$ is one of the labels of $s$.
- state $s' \in S$ is successor state of state $s \in S$, if $(s, s') \in R$. 


Kripke Structures

Remarks:

- For technical reasons, we require that the transition relation $R$ is total; i.e. for each state $s \in S$ there exists a successor state.

- In principle, we could avoid this restriction.
Paths

For a Kripke structure $M = (S, S_0, R, L)$ we call an infinite sequence over $S$

$$\pi = s_0 s_1 s_2 s_3 \ldots$$

a **path** of $M$ in $s_0$, if for each $i \in \mathbb{N}$ state is a successor of $s_i$; i.e. if $(s_i, s_{i+1}) \in R$

A path starting in an initial state of $M$ is called a **run** of $M$. 
Paths and Computation Trees

The set of all paths of $M$ in a state $s$ can be represented as an infinite tree, the **computation tree** of $M$ in $s$:

**Example:**

Since the transition relation $R$ is total, all paths (branches) of the tree are infinite!
Systems and Kripke Structures

- A system resp. a model of a system in another notation can be easily mapped to a Kripke structure (provided that the model is finite).
- Sometimes some information of the model will be lost.

→ Example on next slide
Systems and Kripke Structures

A Petri net

The corresponding Kripke structure
Systems and Kripke Structures

A Petri net

The information on related transitions is lost in the Kripke structure!
Extensions

- Labelling of transitions: **Transition systems**

- Instead of a single transition relation, there are many transition relations (in our example for every Petri net transition).

This is also important for efficiency reasons!
Syntactic Representation

- Motivation & Example
- States
- Initial states
- Transitions
- Labels
Motivation

- Kripke structures are a semantic model for reactive systems (a mathematical structure).
- For real (and large) systems, an explicit enumeration of all states and all transitions is tedious (→ state space explosion).
- Therefore, we use a notation from logic, for representing Kripke structures and transition systems in a more compact way.
Example

- Boolean variables:
  \[ V = \{ a, c \} \]

- Initial formula:
  \[ S_0 \equiv \neg c \]

- Transition formula:
  \[ \mathcal{R} \equiv (a' = \neg a \land c' = c) \lor (a' = a \land c' = \neg c) \]

- Implicit labelling:
  \[ AP = V \]

\[ 0 = \text{false} \]
\[ 1 = \text{true} \]
States

- Let $V = \{ v_1, \ldots, v_n \}$ be a set of Boolean variables.
- We call a mapping $\sigma : V \rightarrow B$ an assignment for variables $V$.
  - $B = \{ 0, 1 \}$ denotes the set of Booleans or truth values (with $0 = \text{false}$ and $1 = \text{true}$).
- Each assignment can be considered as a state.
- This way, the set $V$ implicitly defines a set of states $S = \{ \sigma \mid \sigma : V \rightarrow B \}$. 
The (propositional) formulas over variables $V$ are defined as usual.

Likewise, the validity of a formula $p$ under some assignment $\sigma$ is defined as usual; we write $\sigma \models p$, if $p$ is valid at $\sigma$.

A formula $S_0$ over $V$, the initial formula, defines the set of initial states:

$$S_0 = \{ \sigma \mid \sigma \models S_0 \}.$$
Transition relation

- For a set of variables we define the set of primed variables.

\[ V = \{ v_1, \ldots, v_n \}, \]
\[ V' = \{ v'_1, \ldots, v'_n \} \]

Idea:
- Assignment for \( V \): source state of the transition
- Assignment for \( V' \): target state of the transition
Transition relation

- An assignment for variables $V \cup V'$ can be represented as a pair of assignments $(\sigma, \sigma')$ for $V$:
  - $\sigma(v)$ defines the value for $v$
  - $\sigma'(v)$ defines the value for $v'$

- The validity of formula $p$ over $V \cup V'$ for a pair of assignments $(\sigma, \sigma')$ can be defined as usual: We write $(\sigma, \sigma') \models p$, if $p$ is valid for $(\sigma, \sigma')$
A formula $R$ over $V \cup V'$, the **transition formula**, defines the transition relation of a Kripke structure in the following way:

$$R = \{ (\sigma, \sigma') \mid (\sigma, \sigma') \models R \}$$
Labelling

- The labelling of the states (assignment) can be directly derived from the assignment:

\[ AP = V \]

\[ L(\sigma) = \{ v \in V \mid \sigma(v) = 1 \} = \{ v \in V \mid \sigma \models v \} \]

i.e. each state (assignment) is labelled with those variables that are true in this assignment
Summary

- **Boolean variables:**
  \[ V = \{ a, c \} \]

- **Initial formula:**
  \[ S_0 \equiv \neg c \]

- **Transition formula:**
  \[ R \equiv a' = \neg a \land c' = c \lor a' = a \land c' = \neg c \]

- **Implicit labelling:**
  \[ AP = V \]

**Diagram:**

- States:
  - 0, 1
  - a, c

- Transitions:
  - \((0,0), (0,1), (1,0), (1,1)\)

- Initial states:
  - \((0,0), (1,0)\)

- Transition relation:
  - \(((0,0),(1,0)), ((1,0),(0,0))\)
  - \(((0,1),(1,1)), ((1,1),(0,1))\)
  - \(((0,0),(0,1)), ((0,1),(0,0))\)
  - \(((1,0),(1,1)), ((1,1),(1,0))\)
As a Transition System

- **Boolean variables:**
  \[ V = \{ a, c \} \]

- **Initial formula:**
  \[ S_0 \equiv \neg c \]

- **Transition formula:**
  \[ T \equiv \begin{cases} (a' = \neg a \land c' = c) , \\ (a' = a \land c' = \neg c) \end{cases} \]

**Implicit labelling:**
\[ AP = V \]

This equality is often implicit for variables that do not occur primed.

For example in MCiE (important for efficiency).
More Examples

In this section, we show by the help of two examples how to represent different kinds of systems as Kripke structures represented by formulas.

- Synchronous circuit (hardware)
- Concurrent processes
- Petri nets
Combinatorial Circuit

\[ b' = (a \land b \lor \neg c) \]
\[ c' = \neg c \]
Sequential Synchronous Circuit

\[ b' = (a \land b \lor \neg c) \land (c' = \neg c) \land (a' = 0 \lor a' = 1) \]
Concurrent Processes

\[ \text{loop forever} \]

\[
pca = 0 \quad x := 0; \]
\[
pca = 1 \quad y := 0; \]

\[
( pca = 0 \land pca' = 1 \land x' = 0 \land y' = y \land pcb' = pcb ) \lor \\
( pca = 1 \land pca' = 0 \land y' = 0 \land x' = x \land pcb' = pcb ) \lor \\
( pcb = 0 \land pcb' = 1 \land x' = 1 \land y' = y \land pca' = pca ) \lor \\
( pcb = 1 \land pcb' = 0 \land y' = 1 \land x' = x \land pca' = pca )
\]

\[ \text{loop forever} \]

\[
pcb = 0 \quad x := 1; \]
\[
pcb = 1 \quad y := 1; \]
Petri nets

request₁

critical₁

idle₁

semaphor

request₂

critical₂

idle₂

How do the formulas look for Petri nets as a transition system?
5.4 ROBDDs (details)

- Motivation
- Definition
- Operations on ROBDDs
- Quantified Boolean formulas (QBF)

Reduced Ordered Binary Decision Diagrams; for simplicity often just called Binary Decision Diagrams (BDDs).
Motivation

- The number of states of realistic systems is gigantic.

⇒ Representing sets of states by enumerating every state explicitly is a bad idea.

- Sets could be represented “symbolically”, e.g. by formulas (see next slide)
Sets as formulas

- $\neg C$
- $A$
- $A \leftrightarrow C$
- $\neg A \wedge C$

Boolean formulas representing sets of states
Formulas

Some operations on sets can be efficiently executed for sets that are represented as formulas:

- union: \( p \lor q \)
- disjunction: \( p \land q \)
- complement: \( \neg p \)
- set difference: \( p \land \neg q \)

Problem:

- the same set can have different representations
- it is extremely inefficient to find out whether two formulas represent the same set (NP-complete).
- therefore, formulas are not a good representation for sets of states.

Checking for equality of sets is a very crucial operation in model checking! (BTW: why?) \( \rightarrow \) slide 19/77
Goal

- Representation of sets such that
  - set operations and
  - check for equality

can be computed efficiently

The answer will be Reduced Ordered Binary Decision Diagrams (ROBDDs)!
Binary Decision Trees

\[ a = b \land c = d \]

![Binary Decision Tree Diagram]

Problem: These will also be very big!
Identify same sub-trees

\[ a=b \land c=d \]
Identify same sub-trees

\[ a = b \land c = d \]
Identify same sub-trees

\[ a = b \land c = d \]
Identify same sub-trees

\[ a=b \land c=d \]
Delete redundant nodes

\[ a = b \land c = d \]
Delete redundant nodes

\[ a = b \land c = d \]
Identify same sub-trees

\[ a=b \land c=d \]
Identify same sub-trees

\[ a = b \land c = d \]
Result

\[ a=b \land c=d \]
"Prettified result": ROBDD

\[ a = b \land c = d \]

- All variables on the paths occur in the same order (we had that from the start)
- No identical subgraphs anymore
- No redundant nodes anymore

\[ \Rightarrow \text{ Reduced Ordered Binary Decision Diagram} \]
Observations

- For every set (and a fixed variable order) there exists exactly one ROBDD representing it!

- For many practically relevant sets, the ROBDDs representing them are small.

- The size of the ROBDDs depends on the chosen variable order (on the paths):
  
  For example, the ROBDD for the set characterized by $a=b \land c=d$ is small with variable order $a < b < c < d$; it is bigger with variable order $a < c < d < b$. 
Observations

- There are sets for which the ROBDD will be big for any variable order (multiplication)

- Finding good or even optimal variable orders is one of the challenges of symbolic model checking

- There is no efficient way to find an optimal variable order in general (results from complexity theory)

- But, there are heuristics:
  - Variables that are “somehow related“ should be close to each other
  - Local optimisations by switching two variables
Question

How do we generate an ROBDD?

Answer: Start with full tree and reduce it!
Question

- How do we generate an ROBDD?

Answer: Start with full tree and reduce it!

This is a very bad idea!

Rather, we build them bottom up from formulas with operations on ROBDDs.
Operationen in ROBBDs

- Boolean variable
- Negation
- Restriction and Shannon expansion
- Binary operations
- ROBDDs and Kripke structures
Boolean variable

The set represented by variable $a$ is represented by the ROBBD:
Negation

Complexity: $O(1)$

In practice, this is done a bit differently. Existing ROBDDs are never changed!
Restriction & Shannon expansion

- For a set (resp. Boolean function) $p$ over variables $v_1, \ldots, v_n$ and a Boolean value $t \in \mathbb{B}$, we define the Boolean function $p|_{v_i \leftarrow t}$ by

$$p|_{v_i \leftarrow t}(v_1, \ldots, v_n) = p(v_1, \ldots, v_{i-1}, t, v_{i+1}, \ldots, v_n)$$

- $p|_{v_i \leftarrow t}$ is called restriction of $p$.

- It holds (Shannon expansion of $p$):

$$p = (\neg v \land p|_{v \leftarrow 0}) \lor (v \land p|_{v \leftarrow 1})$$

This is like an “if-then-else” in logics.
For a ROBDD representing a Boolean function $p$, the ROBDD for the $p|_v \leftarrow t$ can be obtained as follows:

- For a ROBDD representing a Boolean function $p$, the ROBDD for the $p|_v \leftarrow t$ can be obtained as follows:

- Subsequently: systematic reduction of the resulting ROBDD.

Remember: Existing ROBDDs are never changed!

In practice, this is done a bit different.

Complexity: $O(|p|)$

Size of the ROBDDs for $p$

Complexity: $O(|p| \cdot \log(|p|))$
An important special case is the restriction to the first variable $v_1$ of the ROBDD:

\[ p|_{v_1} \leftarrow 0 \text{ bzw. } p|_{v_1} \leftarrow 1 \]

In practice, this special case is exploited.

**Complexity:**
\[ O(1) \]
The binary Boolean operations can be formulated recursively by the help of the Shannon expansion:

- $p \land q = \left( \neg v \land (p|_v \leftarrow 0 \land q|_v \leftarrow 0) \right) \lor \left( v \land (p|_v \leftarrow 1 \land q|_v \leftarrow 1) \right)$

- $p \lor q = \left( \neg v \land (p|_v \leftarrow 0 \lor q|_v \leftarrow 0) \right) \lor \left( v \land (p|_v \leftarrow 1 \lor q|_v \leftarrow 1) \right)$

- $p \star q = \left( \neg v \land (p|_v \leftarrow 0 \star q|_v \leftarrow 0) \right) \lor \left( v \land (p|_v \leftarrow 1 \star q|_v \leftarrow 1) \right)$

works for arbitrary binary operators!
Binary Boolean operations

ROBDD for $p \otimes q$ from ROBDDs for $p$ and $q$:

- Generate ROBDDs for $p|_v \leftarrow 0$, $q|_v \leftarrow 0$, $p|_v \leftarrow 1$, and $q|_v \leftarrow 1$
- Construct recursively $p|_v \leftarrow 0 \otimes q|_v \leftarrow 0$ and $p|_v \leftarrow 1 \otimes q|_v \leftarrow 1$
- The OBDD for $p \otimes q$ is:

\[ \begin{align*}
\text{Overall complexity (if cleverly implemented):} \\
O( |p| \cdot |q| )
\end{align*} \]

- Reduce the OBDD systematically to an ROBDD.
ROBDDs: Summary

- As long as all involved ROBDDs remain small, all operations on ROBDDs are efficient.

- There are many libraries implementing ROBDDs and the operations on them (often with clever algorithms for optimizing the variable order). MCiE is a very simple implementation.

- In practice, all ROBDDs in the same context are maintained in a single data structure (as a „forest“ of ROBDDs and hash tables for avoiding duplicate nodes). Then, equality of ROBDDs can be decided in constant time (same pointer).
Quantified Boolean formulas

- For model checking, we need **Boolean formulas** with quantification of Boolean variables $\nu$ (QBF):
  \[ \exists \nu . p \]

- $\exists \nu . p$ is just an abbreviation for $p\mid_{\nu} \leftarrow 0 \lor p\mid_{\nu} \leftarrow 1$

- $\exists \nu . p$ is an abbreviation for
  \[ \exists \nu_1 . ( \exists \nu_2 . ( \ldots ( \exists \nu_n . p ) \ldots ) ) \]

- Respectively, $\forall \nu . p$ stands for $p\mid_{\nu} \leftarrow 0 \land p\mid_{\nu} \leftarrow 1$

- And $\forall \nu . p$ stands for
  \[ \forall \nu_1 . ( \forall \nu_2 . ( \ldots ( \forall \nu_n . p ) \ldots ) ) \]
Relation product

- For a formula, $p(u, v)$ over variables $U$ and $V$ and a formula $q(v, w)$ over variables $V$ and $W$, we call

$$\exists v . \ p(u, v) \land q(v, w)$$

the relation product of $p(u, v)$ and $q(v, w)$.

- The ROBDD for the relation product can be realized with the above abbreviations by the Boolean operations. That, however, is a bit inefficient.

- In practice, the relation product is implemented directly. The worst case complexity is exponential; but, it works reasonably well in many practical setting.
5.5 Symbolic model checking

Represent everything, i.e. initial condition, transition relation as well as the result, as ROBDDs:

**Given:**
- $S_0$ and $\mathcal{R}$ as ROBDDs over $V$ resp. $V \cup V'$
- a CTL-Formula $p$.

**Wanted:**
- The ROBDD for the set of states $S_p$ (the set of states in which $p$ is true).
We assume that we have calculated the ROBDDs for the sets $S_p$ and $S_q$ already.

Next we give the algorithms for calculating the ROBDDs for the sets:

- $S_p \lor q$, $S_p \land q$ and $S_{\neg p}$,
- $S_{\text{EX}p}$,
- $S_{\text{EG}p}$ and
- $S_{\text{E}[p \cup q]}$.

These are the Boolean operations. Algorithms on the following slides!
Algorithm for $\mathbf{EX} p$

Observation:

- $\mathbf{EX} p \equiv \exists \nu'. \mathcal{R}(\nu, \nu') \land p(\nu')$

Given ans ROBDD

Relation product on ROBDDs

$p(\nu)$ given as ROBDD
The only thing left to do is to produce an ROBDD for $p(v')$ from an ROBDD for $p(v)$:

In practice, this renaming is done on the fly (and only temporarily) when the relation product is calculated.

### Algorithm for $\text{EX} p$

1. Rename

#### Complexity:

$O(|p|)$
Reminder: $E[p \cup q ]$

Given: $S_p$ and $S_q$

Wanted: $S_{E[p \cup q ]}$

$$S_0 = S_q$$
$$S_1 = S_q \cup ( S_p \cap EX(S_0))$$
$$S_2 = S_q \cup ( S_p \cap EX(S_1))$$
$$\ldots$$
$$S_{i+1} = S_q \cup ( S_p \cap EX(S_i))$$

until $S_{i+1} = S_i = S_{E[p \cup q ]}$
Algorithm for $E[p \cup q]$

- In this algorithm, the following operations on sets (ROBDDs) occur:
  - test for equality
  - union
  - intersection
  - $EX(S)$

- For all these operations, we have algorithms already (more or less efficient)

- If the iteration does not change anything (check for equality), this is the ROBDD for $S_E[p \cup q]$. 
Procedure \texttt{checkEU}(S_p, S_q)

\[ S := S_p \; ; \] // represented as ROBDD

\begin{align*}
\text{repeat} & \\
S' & := S; \\
S & := S_q \lor (S_p \land \text{checkEX}(S)); \\
\text{until} & \quad S = S';
\end{align*}

\text{return} \; S;

Check for equality! (answers question on \(\rightarrow\) slide 10)
Reminder \( \text{EG}^p \)

This is the inefficient algorithm from the introduction.

With the help of ROBDDs it becomes reasonably efficient.

Given: \( S_p \)

Wanted: \( S_{\text{EG}^p} \)

\[
\begin{align*}
S_0 &= S_p \\
S_1 &= S_p \cap \text{EX}(S_0) \\
S_2 &= S_p \cap \text{EX}(S_1) \\
&\vdots \\
S_{i+1} &= S_p \cap \text{EX}(S_i)
\end{align*}
\]

until \( S_{i+1} = S_i = S_{\text{EG}^p} \)
Procedure \textbf{checkEX(}\(S_p\)\textbf{)}

\[ S := S_p; \quad \text{// represented as ROBDD} \]

\textbf{repeat}

\[ S' := S; \]

\[ S := S_p \land \text{checkEX}(S); \]

\textbf{until} \( S = S' \);

\textbf{return} \( S \);

ROBDD operation

procedure for \( \text{EX}(S) \)

Check for equality
Symbolic model checking

- The use of ROBDDs for the representation of sets of states is called **symbolic model checking** (as in contrast to explicit model checking).

- Symbolic model checking contributed to the initial success of model checking (SMV and today NuSMV)!

- Though it uses more inefficient algorithms as one would use with explicit sets, symbolic model checking is sometimes more efficient (but that depends!).

- It does not work always (for bigger examples).

- There are many other techniques for model checking!

- To date, applying model checking for realistic systems requires much experience.