Model Checking
#342234
http://fmv.jku.at/mc

WS 2018
Johannes Kepler University
Linz, Austria

Prof. Dr. Armin Biere
Institute for Formal Models and Verification
http://fmv.jku.at
Formal Methods in Computer Science

Formal Specification

Formal Synthesis

Formal Verification
Examples of Formal Methods

- Synchronous Theorem Proving
- Compiler
- Languages
  - B−Method
  - SDL
- SAT
- Equivalence
- Model Checking
- VDM
- Formal Specification
- Formal Verification
- Checking
- UML
- Formal Synthesis
- ASM
- Z
- Model Checking
- Theorem Proving
- Formal Verification
- Equivalent Checking
- SAT
- Formal Synthesis
- Compiler
- Synchronous Languages
• Formal Models (Bachelor 4th Semester), resp. Formal Basics III
  – Formal Modelling of (Distributed) Computer Science Systems

• Model Checking (1st Semester Master)
  – Algorithmic Aspects of Explicit Model Checking

• Advanced Model Checking (Master)
  – Satisfiability Solving (SAT)
  – Algorithmic Aspects of Symbolic Model Checking with SAT and BDDs
Motivation of Formal Methods

• Formal Methods as Core Computer Science
  – abstraction is *the* tool of computer science
  – “abstraction” as the most important tool in computer science
  – Computer Science systems *are* mathematical objects
  – abstractions are formal models

• examples of formal technologies with increasing practical relevance:
  – simulation of abstract models for validation purposes
  – equivalence checking in circuit design
  – model checking, abstract interpretation
Origin of Computer Science

Natural Science

Computer Science

Mathematics

Engineering
The growth of a bacterial population can be modeled using differential equations. In reality, the change in population over a small time interval $\Delta t$ is given by:

$$\Delta P(t) = \alpha P(t)$$

This leads to a differential equation:

$$\frac{d P(t)}{d t} = \alpha P(t)$$

with the solution:

$$P(t) = C e^{\alpha t}$$
Computational Science

- modelling and simulation of computer models
  - of natural and artificial systems
  - allows projection into the future ...
    z.B. weather forecast
  - ... and optimization
    e.g. reversal of global warming through less CO$_2$ production

- representation as mathematical equations
  - in general there are **no closed solutions**
  - validation with numerical methods

- **Computational Science as part of Computer Science (?!)**
How much are the developments costs of a program with a certain size?

application programs: \[ PM = 2.4 \cdot (KDSI)^{1.05} \]
utility programs: \[ PM = 3.0 \cdot (KDSI)^{1.12} \]
system programs: \[ PM = 3.6 \cdot (KDSI)^{1.20} \]

\( PM \) = person months (costs) \quad \text{KDSI} = \text{Kilo Delivered Source Instructions (size)}

typical usage of empirical methods particularly in Software Engineering
• are there models in mathematics?
  
  – the natural numbers “are just there”
    Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk [Kronecker]
  
  – there is no need for interpretation: subject = reality

• model concept in mathematical logic
  
  – is it possible to model mathematics with mathematics?
    no, not in general [Gödel]
  
  – weaker statements are possible:
    many theorems can be derived formally
    (within a formal calculus)
• programs and other digital systems are formal objects
  – they have precise mathematical models (denotational/operational)
  – Reality = Model
    (modulo complex semantics, compiler bugs, hardware failure, . . .)
  – properties of the models also hold in reality

• proving properties of models is difficult
  – for Software in general undecidable
  – for Hardware in NP or PSPACE

• only valid for functional properties, not for quantitative aspects
  – availability, through put, latency, etc. are difficult to model precisely
modelling of a controller

- model also includes the artifact (controller)
- models are an approximation of reality
  - real system is different from the model
- goal is construction/synthesis and optimization of the controller
• goal is construction/synthesis and optimization of Computer Science systems

• models for quantitative analysis/optimization
  – Markov-chains etc.
  – quantitative/probabilistic simulation

• high-level models
  – stepwise refinement/synthesis (e.g. code generation, compiler)
  – example: Model Driven Architecture (MDA) = executable UML models
  – example: behavioral models and synthesis for digital systems
• models as in Natural Science
  – Computational Science, computer models

• further empirical models:
  – empirical methods in Software Engineering

• mathematical/formal modelling
  – logic as basis, SW/HW as formula, reality = model

• high-level models
  – qualitative (functional) or quantitative, refinement/synthesis
Refinement SW

for instance software design

- requirements
- high-level design
- low-level design
- implementation
- configuration
Refinement HW

for instance hardware design

architecture
RTL
gates
switch-level
layout
Specification vs Implementation

- verification
  - validation
  - verification
  - simulation
  - test
- implementation
- specification
- synthesis
  - translation
  - compilation
  - transformation
  - refinement

(formal or adhoc)
Finite Automata (FA)

compare with Formal Basics 3 resp. Formal Models

motivation: automata for modelling, specification and verification

**Definition** a *finite automaton* \( A = (S, I, \Sigma, T, F) \) consists of

- set of states \( S \) (usually finite)
- set of initial states \( I \subseteq S \)
- input alphabet \( \Sigma \) (usually finite)
- transition relation \( T \subseteq S \times \Sigma \times S \)
  - write \( s \xrightarrow{a} s' \) iff \((s, a, s') \in T\) iff \( T(s, a, s') \) “holds”
- set of final states \( F \subseteq S \)
**Definition** An FA $A$ accepts a word $w \in \Sigma^*$ iff there are $s_i$ and $a_i$ with 

$$
\begin{align*}
  s_0 & \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n,
\end{align*}
$$

where $n \geq 0$, $s_0 \in I$, $s_n \in F$ and $w = a_1 \cdots a_n \ (n = 0 \Rightarrow w = \varepsilon)$.

**Definition** The language $L(A)$ of $A$ is the set of its accepted words.

- use automata or regular languages to describe event streams
- “conformance” of implementation event streams to their specification
- conformance relates to sub set relation of languages
**Definition** The Product Automaton $A = A_1 \times A_2$ of two FA $A_1$ and $A_2$ with common input alphabet $\Sigma_1 = \Sigma_2$ has the following components:

\[
S = S_1 \times S_2, \quad I = I_1 \times I_2 \\
\Sigma = \Sigma_1 = \Sigma_2, \quad F = F_1 \times F_2 \\
T((s_1, s_2), a, (s'_1, s'_2)) \text{ iff } T_1(s_1, a, s'_1) \text{ and } T_2(s_2, a, s'_2)
\]

**Theorem** Let $A, A_1,$ and $A_2$ as above, then $L(A) = L(A_1) \cap L(A_2)$

**Example:** construction of automata which accepts words with prefix $ab$ and suffix $ba$

(as regular expression: $a \cdot b \cdot 1^* \cap 1^* \cdot b \cdot a$, where $1$ denotes the set of all letters)
**Definition** for $s \in S$, $a \in \Sigma$ define $s \xrightarrow{a}$ as the set of successors of $s$ with

$$s \xrightarrow{a} = \{ s' \in S \mid T(s, a, s') \}$$

**Definition** An FA is *complete* iff $|I| > 0$ and $|s \xrightarrow{a}| > 0$ for all $s \in S$ and $a \in \Sigma$.

**Definition** … *deterministic* iff $|I| \leq 1$ and $|s \xrightarrow{a}| \leq 1$ for all $s \in S$ and $a \in \Sigma$.

**Fact** … deterministic and complete iff $|I| = 1$ and $|s \xrightarrow{a}| = 1$ for all $s \in S$, $a \in \Sigma$. 
Definition The power automaton $A = \mathbb{P}(A_1)$ of an FA $A_1$ has the following components

$$
S = \mathbb{P}(S_1) \quad (\mathbb{P} = \text{power set}) \quad I = \{I_1\}
$$

$$
\Sigma = \Sigma_1 \quad F = \{F' \subseteq S \mid F' \cap F_1 \neq \emptyset\}
$$

$$
T(S', a, S'') \iff S'' = \{s'' \mid \exists s' \in S' \text{ with } T_1(s', a, s'')\}
$$

Theorem $A, A_1$ as above, then $L(A) = L(A_1)$ and $A$ is deterministic and complete.

Example: Spam filter based on the white list “abb”, “abba”, and “abacus”!

(regular expression: “abb” | “abba” | “abacus”)

Model Checking #342234 WS 2018 Armin Biere JKU Linz
Definition The complement automaton \( A = C(A_1) \) of an FA \( A_1 \) has the same components as \( A_1 \) except \( F = S \setminus F_1 \).

Theorem The complement automaton \( A = C(A_1) \) of a deterministic and complete automaton \( A_1 \) accepts the same language \( L(A) = \overline{L(A_1)} = \Sigma^* \setminus L(A_1) \).

Example: Spam filter based on black list “abb”, “abba”, and “abacus”!
(regular expression: “abb” | “abba” | “abacus”)

Model Checking #342234 WS 2018 Armin Biere JKU Linz
Conformance of Event Streams

- modelling and specification with automata:
  - event streams of an implementation represented by FA $A_1$
  - partial specification of event streams as FA $A_2$

- conformance test:
  - $L(A_1) \subseteq L(A_2)$
  - iff $L(A_1) \cap \overline{L(A_2)} = \emptyset$
  - iff $A_1 \times C(\mathbb{P}(A_2))$ contains no reachable final state

- **Example:** specification $S = (cs | sc | ss)^*$, implementation $I = (s | c)^2)^*$
Further Conformance Examples

- temporal properties: \((1\text{ denotes an arbitrary letter)}\)
  - every third step \(a\) holds: \((1 \cdot 1 \cdot a)^*\)
  - exactly every third step \(a\) holds: \((\bar{a} \cdot \bar{a} \cdot a)^*\)
  - \(a\) (acknowledge) has to be preceded by \(r\) (request): \((\bar{r})^* \cdot a\)
  - each \(a\) has to be preceded by an \(r\): \((1^* \cdot a)^* \cdot (\bar{r})^* \cdot a\)

- refinement: (scheduling of three processes \(a, b\) and \(c\))
  - abstract round robin scheduler: \((abc \mid acb \mid bac \mid bca \mid cab \mid cba)^*\)
  - round robin scheduler, \(a\) higher priority than \(b\): \((abc \mid acb \mid cab)^*\)
  - round robin scheduler, \(a\) before \(b, c\) before \(b\): \((acb \mid cab)^*\)
  - deterministic round robin scheduler of implementation: \((cab)^*\)
• similar approach:
  – given a propositional formula \( f \) over boolean variables \( V = \{x_1, \ldots, x_n\} \)
  – the expansion \( E(f) \subseteq 2^n \) is the set of satisfying assignments of \( f \)
    \[
    (a_1, \ldots, a_n) \in E(f) \iff f[x_1 \rightarrow a_1, \ldots, x_n \rightarrow a_n] = 1
    \]
  – e.g. \( E(f) \neq \emptyset \) iff \( f \) satisfiable

• modelling and specification:
  – \( f_1 \) characterizes the implementation for all possible configurations
  – \( f_2 \) represents a partial specification of all valid configuration

• conformance test: \( f_1 \Rightarrow f_2 \) iff \( E(f_1) \subseteq E(f_2) \) iff \( E(f_1) \cap \overline{E(f_2)} = \emptyset \)
  (or in practice: \( f_1 \land \neg f_2 \) unsatisfiable)
Abstracting a C-Program to an FA

\begin{align*}
\text{while} \ (\ldots) \ {}\&{}
\text{lock} (){}\&{}
\ldots
\text{while} \ (\ldots) \ {}\&{}
\text{if} \ (\ldots) \ {}\&{}
\text{lock} (){}\&{}
\ldots
\text{else} \ {}\&{}
\text{unlock} (){}\&{}
\ldots
\text{lock} (){}\&{}
\ldots
\text{unlock} (){}\&{}
\ldots
\end{align*}

\[(l \cdot (l \mid u \cdot l)^* \cdot u)^*\]
Spurious Counter Examples

```
assert (i < n);
lock ();
do {
  ...
  i++;
  if (i >= n)
    unlock ();
  ...
} while (i < n);
```

\[ l \cdot (\varepsilon|u)^* \text{ violates partial specification } 1^* \cdot l \cdot \overline{u}^* \]

(“…” neither leads to a lock nor unlock and leaves i and n untouched)
refinement of abstraction by introduction of a predicate variable:

\[
\begin{align*}
\text{assert } (i < n); \\
\text{lock } (); \\
\text{do } \\
\text{...} \\
i++; \\
\text{if } (i \geq n) \\
\text{unlock } (); \\
\text{...} \\
\text{while } (i < n);
\end{align*}
\]

assert \( (i < n) \);
lock ()
do {
  ...
i++;
  if \ (i \geq n)
    unlock ();
  ...
} while \ (i < n);

is abstracted to

\[
\begin{align*}
\text{assert } (b); \\
\text{lock } (); \\
\text{do } \\
\text{...} \\
i++; \\
\text{if } (b) 
\quad b = *; \\
\text{if } (!b)
\quad \text{unlock } ();
\text{...} \\
\text{while } (b);
\end{align*}
\]

assert \( b \);
lock ()
do {
  ...
i++;
  if \ (b)
    b = *;
  if \ (!b)
    unlock ();
  ...
} while \ (b);

\( l \cdot \varepsilon^* \cdot u \) satisfies partial specification \( 1^* \cdot l \cdot \overline{u}^* \)

(“...” neither leads to a lock nor unlock and leaves \( i \) and \( n \) untouched)
```c
int bsearch (int * a, int n, int e) {
    int l = 0, r = n;
    if (!n) return 0;
    while (l + 1 < r) {
        int m = (l + r) / 2;
        printf("l=%d r=%d\n", l, r);
        if (e < a[m]) r = m;
        else l = m;
    }
    return a[l] == e;
}
```

```c
int main (void) {
    int n = INT_MAX;
    int * a = calloc (n, 4);
    (void) bsearch (a, n, 1);
}
```

Running this allocates 8GB and actually took more than 3 seconds.

```bash
$ ./bsearch
l=0 r=2147483647
l=1073741823 r=2147483647
Segmentation fault
```

**fix:**
```
int min = l + (r - l)/2;
```
Labelled Transition Systems (LTS)

- general semantical model for **process algebra**
  - focus on **reactive** or **open** systems
  - concept of **environment** with **external** events
  - implementation (on one abstraction layer) determines **internal** events

- an LTS \( A = (S, I, \Sigma, T) \) essentially is an FA:
  - only behavior, e.g. potential of transition, is important
  - no final states: no “explicit” language
  - “implicit” language \( L(A) \) defined by \( F = S \)
Vending Machine of Milner

\[ G \]

- \( p \) = pay
- \( d \) = dark chocolate
- \( m \) = milk chocolate

\[ B \]
• semantics of the two LTS should be “different”
  
  – $G$ allows to select the type of chocolate after paying
  
  – $B$ non deterministically determines the chocolate type while paying

• but $B$ and $G$ are **language equivalent**:

  
  – $L(B) = p \cdot (d \mid m) = L(G)$

• same problem with conformance test:

  
  – language based conformance test identifies $B$ and $G$
  
  – language conformance ignores “branching behavior”
What is Simulation?

- behavior of implementation $A_1$ should be consistent with specification $A_2$
  - each transition in $A_1$ has a counterpart in $A_2$
  - $A_2$ simulates $A_1$
  - $A_2$ may have more behavior

- to simplify exposition merge $A_1$ and $A_2$ into one LTS $A$
  - common alphabet $\Sigma$
  - (disjoint) union of the other components:
    \[ S = S_1 \cup S_2, \quad I = I_1 \cup I_2, \quad T = T_1 \cup T_2 \]
  - … written as \[ A = A_1 \cup A_2 \]
**Definition** a relation $\preceq \subseteq S \times S$ over an LTS $A$ is a **simulation** iff
(read $s \preceq t$ as $t$ simulates $s$)

$$s \preceq t \text{ then } \forall a \in \Sigma, s' \in S \left[ s \xrightarrow{a} s' \Rightarrow \exists t' \in S \left[ t \xrightarrow{a} t' \land s' \preceq t' \right] \right]$$

**Fact** there is exactly one maximal simulation over an LTS $A$

**Proof** (sketch) $S$ finite

- union of simulations is again a simulation

- the set of simulations over $A$ is non empty (it contains the identity)
Construction of the Maximum Simulation Relation

• starting point: $\preceq_0 = S \times S$ (usually not a simulation)

• refine $\preceq_i$ to $\preceq_{i+1}$ as follows

  \[ s \preceq_{i+1} t \iff s \preceq_i t \quad \text{and} \quad \forall a \in \Sigma, s' \in S \left[ s \xrightarrow{a} s' \Rightarrow \exists t' \in S \left[ t \xrightarrow{a} t' \land s' \preceq_i t' \right] \right] \]

• for finite $S$ there is an $n$ with $\preceq_n = \preceq_{n+1}$
  
  – $\preceq_n$ obviously a simulation
  
  – maximality less obvious

• can be interpreted as fixpoint computation
Let $\preceq$ be a simulation.

Show $\preceq \subseteq \preceq_i$ with induction over $i$.

Base case is trivial, induction step follows.

Assume (indirect proof): $\preceq \not\subseteq \preceq_{i+1}$.

Then there is $s$ and $t$ with $s \preceq t$ but $s \not\sim_{i+1} t$.

Therefore there has to be $s'$ and $a$ with $s \xrightarrow{a} s'$, but $t \not\xrightarrow{a} t'$ or $t \not\gamma_i t'$ for all $t'$.

With the induction hypothesis $\preceq \subseteq \preceq_i$ we get: $t \not\xrightarrow{a} t'$ or $t \not\gamma_i t'$ for all $t'$.

Contradiction to the assumption that $\preceq$ is a simulation.
More on Simulations

**Fact** maximal simulations are transitive and reflexive

**Proof** (sketch)

- maximum simulation is reflexive because identity is a simulation
- transitivity by the following lemma

**Lemma** transitive hull of a simulation is again a simulation

**Proof** the following operator produces simulations from simulations

$$\Psi : \mathcal{P}(S \times S) \rightarrow \mathcal{P}(S \times S) \quad \Psi(\preccurlyeq)(r,t) \iff r \preccurlyeq t \quad \text{or} \quad \exists s [r \preccurlyeq s \land s \preccurlyeq t]$$
Definition

LTS $A_2$ simulates LTS $A_1$ iff there is a simulation $\sqsubseteq$ over $A_1 \cup A_2$ such that for all initial states $s_1 \in S_1$ of $A_1$ there is an initial state $s_2 \in S_2$ of $A_2$ with $s_1 \sqsubseteq s_2$. Also written as $A_1 \sqsubseteq A_2$.

Fact  simulation on LTS are transitive and reflexive

Proof  (sketch)

- construct maximal simulation over all three LTS
- show existence of simulating initial states
- project to the two outer LTS
Definition A trace of an LTS $A$ is a word $w = a_1 \cdots a_n \in \Sigma^*$ with

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_{n-1}} s_{n-1} \xrightarrow{a_n} s_n,$$

where $s_0 \in I$ and $n \geq 0$.

Fact $L(A) = \{ w \mid w \text{ trace of } A \}$

Theorem (simulating LTS are conservative abstractions)

If LTS $A_2$ simulates $A_1$ ($A_1 \preceq A_2$), then $L(A_1) \subseteq L(A_2)$.

Application $P \preceq A \preceq S \implies L(P) \subseteq L(S)$

($P = \text{ program}, A \text{ abstraction, } S \text{ specification}$)
• \( \tau \in \Sigma \) represents a non observable *internal event*

• previous definition of simulation becomes **strong simulation**

\[
\text{\( s \preceq t \) then } \forall a \in \Sigma, \ s' \in S [s \xrightarrow{a} s' \Rightarrow \exists t' \in S [t \xrightarrow{a} t' \land s' \preceq t']] \]

• write \( s \xrightarrow{\tau^* a} t \) if there is \( s_0, \ldots, s_n \) with

\[
s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_{n-1} \xrightarrow{a} s_n = t \]

• a relation \( \preceq \) is a **weak simulation** iff

\[
\text{\( s \preceq t \) then } \forall a \in \Sigma \setminus \{\tau\}, \ s' \in S [s \xrightarrow{\tau^* a} s' \Rightarrow \exists t' \in S [t \xrightarrow{\tau^* a} t' \land s' \preceq t']] \]
• use $\tau$ to *abstract* events
  
  – for instance computations / data flow irrelevant for synchronization

• $\tau$-cleared LTS $A$ of an LTS $A_1$ with $\tau$: $\Sigma = \Sigma_1 \setminus \{\tau\}$, $T(s, a, t)$ iff $s \xrightarrow{\tau^a} t$ in $A_1$.

  – $\tau$ removal produces a strong simulation from a weak one

  – previous algorithms can be adapted to work here as well

• transitivity and applications as with strong simulation

• divergence $s \xrightarrow{\tau^+} s$ is not handled sufficiently

  – $A_1 \preceq A_2$ allows $A_1$ to diverge and $A_2$ not
Idea: implementation is \textit{exactly} the specified behavior and not more!

**Definition** a relation \( \approx \) is a \textbf{strong bisimulation} iff

\[
s \approx t \quad \text{then} \quad \forall a \in \Sigma, \ s' \in S \left[ s \xrightarrow{a} s' \Rightarrow \exists t' \in S \left[ t \xrightarrow{a} t' \land s' \approx t' \right] \right] \quad \text{and}
\]

\[
\forall a \in \Sigma, \ t' \in S \left[ t \xrightarrow{a} t' \Rightarrow \exists s' \in S \left[ s \xrightarrow{a} s' \land s' \approx t' \right] \right]
\]

**Definition** a relation \( \approx \) is a \textbf{weak bisimulation} iff

\[
s \approx t \quad \text{then} \quad \forall a \in \Sigma \setminus \{\tau\}, \ s' \in S \left[ s \xrightarrow{\tau^* a} s' \Rightarrow \exists t' \in S \left[ t \xrightarrow{\tau^* a} t' \land s' \approx t' \right] \right] \quad \text{and}
\]

\[
\forall a \in \Sigma \setminus \{\tau\}, \ t' \in S \left[ t \xrightarrow{\tau^* a} t' \Rightarrow \exists s' \in S \left[ s \xrightarrow{\tau^* a} s' \land s' \approx t' \right] \right]
\]

weak bisimulation is useful when abstracting internal events of the implementation with \( \tau \)

theoretical application: bisimulation equivalent LTS “have the same properties”
Minimization of Deterministic FA

Given a deterministic and complete FA \( A = (S, I, \Sigma, T, F) \)

- starting point: \( \sim_0 = (F \times F) \cup (F \times F) \)
  - partition with respect to “final state flag”
  - equivalence relation

- refine \( \sim_i \) by \( \sim_{i+1} \)

  \[
  s \sim_{i+1} t \quad \text{iff} \quad s \sim_i t \quad \text{and}
  \forall a \in \Sigma, s' \in S \left[ s \xrightarrow{a} s' \Rightarrow \exists t' \in S \left[ t \xrightarrow{a} t' \land s' \sim_i t' \right] \right] \quad \text{and}
  \forall a \in \Sigma, t' \in S \left[ t \xrightarrow{a} t' \Rightarrow \exists s' \in S \left[ s \xrightarrow{a} s' \land s' \sim_i t' \right] \right]
  \]

- termination \( \sim_{n+1} = \sim_n \) guaranteed at \( n = |S| \)

- equivalence relation \( \sim = \sim_n \) produces minimal automaton \( A/\sim \)
Reachability Analysis

**Definition**  
*Reachability Analysis* is …

- computations of all reachable states
  - starting from initial states
  - result is represented either “explicitly” or “symbolically”
  - can be used for optimizing systems, e.g. remove dead code

- check reachability of certain states
  - corresponds to model checking of safety properties
Explicit vs Symbolic

**explicit** = each state/transition is represented separately

**symbolic** = sets of states/transitions represented as formulas

<table>
<thead>
<tr>
<th></th>
<th>explicit model</th>
<th>symbolic model</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit analysis</td>
<td>graph search</td>
<td>explicit model checking</td>
</tr>
<tr>
<td>symbolic analysis</td>
<td>—</td>
<td>symbolic model checking</td>
</tr>
</tbody>
</table>

originally: symbolic MC for HW (SMV), explicit MC for SW (SPIN)
today: symbolic and explicit MC for both SW+HW
 Explicit vs Symbolic Model

• in theory explicit matrix representation of $T$ is enough

<table>
<thead>
<tr>
<th>transition matrix</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

initial state: 0

• in practice $T$ is given in a modelling or programming language

```python
initial_state = 0;
next_state = (current_state + 2 * (bit + 1)) % 8;
```

• symbolic representation can be exponentially more succinct
Symbolic Reachability Analysis

- symbolic traversal of state space
  - determine set of states reachable in one step from a given set of states
  - successor states are computed and represented symbolically
  - has some flavor of breadth first search
  - for (infinite state) programs undecidable

- result is symbolic representation of all reachable states
  - for instance 0 <= state && state < 8 && even (state)
• symbolic representation usually has “parallel composition operator”
  – for instance product of automata, each automaton is represented explicitly
    \[ G = C_1 \times C_2 \]
  – components are programmed separately
    \[
    \text{process A begin P1 end || process B begin P2 end;}
    \]

• program size of whole system:
  – \(|C_1| + |C_2|\), resp. \(|P1| + |P2|\)
  – sum of the sizes of the components

• also applies to HW
• size of state space:
  \[ |S_G| = |S_{C_1}| \cdot |S_{C_2}| \]
  – product of the number of states of the components

• sequential circuits:
  – \( n \)-bit counter can be implemented with \( O(n) \) gates, but has \( 2^n \) states

• hierarchical descriptions can lead to another exponential factor

• for (infinite state) SW even more complex:
  – in theory not even double exponential is enough
  – in practice heap is the problem
Explicit vs Symbolic Modelling

• explicit representation reduces to graph search:
  – search for reachable states starting from the initial states
  – linear in the size of the number of all states

• symbolic representation:
  – computation of the successor states is the main problem
  – on-the-fly expansion of $T$ for a concrete state (simulate or interpret $T$)
  – avoid computation of explicit transitions of $T$ for unreachable states
  – alternatively compute symbolic representation of successor states
• mark all visited states
  – implementation depends on, whether states are generated on-the-fly
• unmarked successors of visited states pushed on search stack
• next working state popped from top of search stack
• if “error” or “target” state reached then
  – path to error resp. target state is on the search stack
  – to simplify the algorithms abort search
recursive_dfs_aux (Stack stack, State current) 
{
    if (marked (current))
        return;

    mark (current);
    stack.push (current);

    if (is_target (current))
        stack.dump_and_exit ();   /* target reachable */

    forall successors next of current
        recursive_dfs_aux (stack, next);

    stack.pop ();
}
Reachability Analysis with DFS

recursive_dfs ()
{
    Stack stack;

    forall initial states state
        recursive_dfs_aux (stack, state);

    /* target not reachable */
}

• “abort/exit” needs to be handled more gracefully of course

• recursive version may run out of stack memory

• support on-the-fly generation of reachable states by hashing
this C program crashes after a “few” recursions
non_recursive_dfs_aux (Stack stack) {
  
  while (!stack.empty ()) {
    current = stack.pop ();
    
    if (is_target (current))
      dump_family_line_and_exit (current);
    forall successors next of current
      {
        if (cached (next)) continue;
        cache (next);
        stack.push (next);
        next.set_parent (current);
      }
  }
}
non_recursive_dfs ()
{
  Stack stack;

  forall initial states state
  {
    if (!cached (state))
      cache (state);
    stack.push (state);
  }
  non_recursive_dfs_aux (stack);
  /* target not reachable */
}
non_recursive_buggy_dfs_aux (Stack stack)
{
    while (!stack.empty ())
    {
        current = stack.pop ();
        if (is_target (current))
            dump_family_line_and_exit (current);
        if (cached (current)) continue;
        cache (current);
        forall successors next of current
        {
            stack.push (next);
            next.set_parent (current);
        }
    }
}
non_recursive_but_also_buggy_aux (Stack stack)
{
    while (!stack.empty ())
    {
        current = stack.pop ();
        forall successors next of current
        {
            if (cached (next)) continue;
            cache (next);
            stack.push (next);
            next.set_parent (current);
            if (is_target (next))
                dump_family_line_and_exit (next);
        }
    }
}
bfs_aux (Queue queue)
{
    while (!queue.empty())
    {
        current = queue.dequeue();
        if (is_target (current))
            dump_family_line_and_exit (current);
        forall successors next of current
        {
            if (cached (next)) continue;
            cache (next);
            queue.enqueue (next);
            next.set_parent (current);
        }
    }
}
bfs ()
{
    Queue queue;

    forall initial states state
    {
        if (!cached (state))
            cache (state);
        queue.enqueue (state);
    }

    bfs_aux (queue);

    /* target not reachable */
}
DFS vs BFS

- DFS
  - easy to implement recursively (which you should not do anyhow)
  - can be extended to detect cycles $\Rightarrow$ liveness
  - faster if there are many but deep targets

- BFS
  - requires non recursive formulation from the beginning
  - generates shortest paths to error/target states
  - no cycle detection (at least not easy to implement)
  - faster if there are few but shallow targets
Backward vs Forward Traversal

• forward:
  – successors of working states are next states to work with
  – all analyzed states are reachable

• backward:
  – predecessors of working states are next states to work with
  – analyzed states can be unreachable
     – most useful in symbolic analysis
     – in some applications backward analysis terminates fast
• global analysis “adds one bit” for each possibly reachable state

    bool cached (State state) { return state.mark; }
    void cache (State state) { state.mark = true; }

• global analysis without on-the-fly state generation

  – example: struct State { unsigned components[2]; bool mark; };
  – \(2^{64} = 2^{32} \cdot 2^{32}\) possible states can not be allocated \(9 \cdot 2^{24}\) TB \(\approx 151\) million TB

• local analysis with on-the-fly state generation

  – \(2^{20} = 1048576\) reachable states if always say \(\text{components}[i] < 1024\)
  – need 9 times more bytes, so only 9 MB
    plus overhead to access them
• requirements:
  – visited states have to be marked and saved \((\text{cache})\)
    \[\Rightarrow\] insertion operation for each new visited state
  – successors have to be checked for already “being visited” \((\text{cached})\)
    \[\Rightarrow\] contains check for each successor

• alternatives:
  – bit set: for each \textit{possible} state one bit (as in global analysis)
  – search trees: operations logarithmic in number of reachable states
  – \textbf{hash table}: operations constant time in number of reachable states
• look up and insertion time can be assumed to be constant
  – in theory much more complex analysis required

• “good” hash functions are important
  – hash index computed by randomization resp. distribution of input bits

• adaptation of hash table size:
  – either enlarge dynamically by constant factor!
  – or use all available memory
unsigned hash (unsigned data) { return data; }  

struct Bucket *  
find (unsigned data)  
{  
  unsigned h = hash (data);  
  h &= (size_table - 1);  
  h &= (size_table - 1);  
  ...  
}

\(\text{Table with collision chains:}\)
\[
\begin{array}{cccc}
0 & 8 & 0 & 17 \\
1 & 9 & 7 & 3 \\
2 & 3 & & \\
3 & & & \\
\end{array}
\]

\[\text{count_table} == 5\]

\[\text{size_table} == (1 \ll 2)\]
unsigned
bad_string_hash (const char * str)
{
    const char * p;
    unsigned  res;

    res = 0;

    for (p = str; *p; p++)
        res += *p;

    return  res;
}
unsigned
very_bad_string_hash (const char * str)
{
    const char * p;
    unsigned res;

    res = 0;

    for (p = str; *p; p++)
        res = (res << 4) + *p;

    return res;
}
unsigned
classic_string_hash (const char *str)
{
    unsigned res, tmp;
    const char *p;

    res = 0;

    for (p = str; *p; p++)
    {
        tmp = res & 0xf0000000; /* unsigned 32-bit */
        res <<= 4;
        res += *p;
        if (tmp)
            res ^= tmp >> 28;
    }

    return res;
}
• empirically good randomization for identifiers in programming languages
  – **average number collisions** as quality metric for good hash functions

• **fast**: max. 4 logical/arithmetical operations per character

• for strings longer than 8 characters good distribution of bits

• overlapping of 8-bit encodings of individual characters
  (but beware of ASCII encoding)

• clustering effects for many short strings (e.g. in automatically generated code)
  \[ n_1, \ldots, n_{99}, n_{100}, \ldots, n_{1000} \]
static unsigned primes [] = { 2000000011, 2000000033, ... };  
#define NUM_PRIMES (sizeof (primes) / sizeof (primes[0]))

unsigned

primes_string_hash (const char * str)
{
    unsigned res, i;
    const char * p;

    i = 0;
    res = 0;
    for (p = str; *p; p++)
    {
        res += *p * primes[i++];
        if (i >= NUM_PRIMES)
            i = 0;
    }

    return res;
}
static unsigned primes[] = { 2000000011, 2000000033, ... };  
#define NUM_PRIMES (sizeof(primes)/sizeof(primes[0]))

unsigned hash_state (unsigned * state, unsigned words_per_state)
{
    unsigned res, i, j;

    res = 0;
    i = 0;
    for (j = 0; j < words_per_state; j++)
    {
        res += state[j] * primes[i++];
        if (i >= NUM_PRIMES)
            i = 0;
    }

    return res;
}
#define CRC_POLYNOMIAL 0x82F63B78
static unsigned
crc_hash_bit_by_bit (const char * str)
{
    unsigned  const char * p;
    unsigned  res = 0;
    int       i, bit;
    for (p = str; *p; p++) {
        res ^= *p;
        for (i = 0; i < 8; i++) {
            bit = res & 1;
            res >>= 1;
            if (bit)
                res ^= CRC_POLYNOMIAL;
        }
    }
    return res;
}
• easy to parameterize
  – just use different prime numbers or starting positions (salt)

• integer multiplication is quite fast on modern processors
  – more space (transistors) available for multipliers
  – super scalar processors: multiple integer functional units

• how to adjust hash index to table size
  – power of two sizes: bit masking
  – otherwise calculate remainder
• problem with explicit **complete** state space exploration:

1. too many states due to state space explosion

2. single state requires already quite some space to be saved (10 - 1000 bytes?)

• simple idea of super-trace algorithm:

1. treat states with the same hash value as identical states

2. do not save states

3. save only their hash value

• super-trace algorithm = **bit-state hashing**
if a state with the same hash value as the hash value of the current state was visited before then the current state is considered to have been visited as well!
Analysis of Super-Trace Algorithm

• Advantages:
  – drastic reduction of space usage (one bit per state)
  – reduced by at least 8 times the size of a single state in bytes
  – fit size of hash table to the amount of available main memory
  – produces a lower bound on the number of visited and thus reachable states
  – hash function parameterization and/or search order randomization …
  – … thus different parts of the search space are explored

• Disadvantages:
  – incompleteness due to non collision free hashing
  – coverage?
Using Two Hash Functions for Super-Trace Algorithm

- calculate two hash values with different hash functions
  - save state by storing in both hash tables one bit
  - now state is assumed to have been visited iff both bits are already set
  - z.B. \( h_1, h_2 : \text{Keys} \rightarrow \{0, \ldots, 2^{32} - 1\} \)
  - two hash tables with \(2^{32}\) bits = \(2^{29}\) bytes = 512 MB

- easy to extend to \(n\) hash functions with \(n > 2\)
  - \(n = 4\) with 2 GB main memory for hash tables is realistic
  - cf. parameterization of hash function using multiplication with primes
• instead of storing $n$ bits it is also possible to store the $n$ bit hash value in the hash table

- hash table using 256 MB = $2^{28}$ bytes can store $2^{26}$ hash values / states
- 32 bit hash function $h: keys \rightarrow \{0, \ldots, 2^{32} - 1\}$
- 4 bytes hash value per state

(after [Holzmann 1998], 427567 reachable states, 1376 bits)
• best case assumptions:
  – hash function is collision free
  – hash table is filled as long as possible without collisions

• $m$ memory usage in bits, $s$ state size in bits, $r$ reachable states

  $\text{coverage}_{n=2} (m) = \frac{m}{r \cdot 2}$

  $\text{coverage}_{\text{hash compact}} (m) = \frac{m}{r \cdot w}$ \text{ w = word size in bits, e.g. 32 bits, $w \leq \lceil \log_2 r \rceil$}

  $\text{coverage}_{\text{complete}} (m) = \frac{m}{r \cdot s}$

• in practice collisions do occur

• complete methods actually need more memory to achieve the same coverage

• these techniques are actually so called “Bloom filters”
Asynchronous versus Synchronous Composition

• synchronous composition

  – lock step of all components, global clock

  – corresponds to product automaton construction

  – typical model for sequential **hardware**
    (even though on switch/transistor level we need asynchronous models)

• asynchronous composition

  – components run independently, local clock

  – typical model for communication protocols and distributed **software**

  – synchronization: system calls, interrupts, signals, messages, channels, RPC

  – appropriate simplification: **interleaving**
- alternating execution of each process for local actions

- synchronization on global actions by rendezvous

- standard parallel composition in process algebra
Asynchronous Composition of LTS through Interleaving

**Definition** Let $A_1$ and $A_2$ be LTS. Their parallel composition $A = A_1 || A_2$ consists of the following components:

- $S = S_1 \times S_2$,
- $\Sigma = \Sigma_1 \cup \Sigma_2$,
- $I = I_1 \times I_2$,
- $T$ is defined as follows:

$$
(s, t) \xrightarrow{a} (s', t') \text{ in } A \text{ iff } t \xrightarrow{a} t' \text{ in } A_2 \text{ and } s' = s \text{ if } a \in \Sigma_1 \cap \Sigma_2
$$

**interleaving** with synchronization on common actions
Local and Global Symbols/Events/Actions

**Definition** In $A_1 \parallel A_2$ a symbol $a$ is **local** for $A_i$ iff $a \in \Sigma_i$ and $a \notin \Sigma_j$ for all $i \neq j$.

The set of local symbols for $A_i$ is denoted as $\Lambda_i$.

**Definition** A symbol is called **local** if it is local for one $A_i$.

The set of all local symbols is denoted as $\Lambda = \bigcup \Lambda_i$.

**Definition** Transition $(s_1, s_2) \xrightarrow{a} (s'_1, s'_2)$ in $A_1 \parallel A_2$ is local (for $A_i$), iff $a$ is local (for $A_i$).

**Definition** Symbols resp. transitions are **global**, iff they are not local.

The set of global symbols for $A_i$ is denoted $\Gamma_i$ and for all components denoted as $\Gamma = \bigcup \Gamma_i$.

If $i = 1$ let $\sigma(i) = 2$ and vice versa.

**Fact** $(s_1, s_2) \xrightarrow{a} (s'_1, s'_2)$ in $A$ iff $s_i \xrightarrow{a} s'_i$ in $A_i$ and $s'_j = s \sigma(i)$ if $a$ local for $A_i$

$s_j \xrightarrow{a} s'_j$ in $A_j$ for all $j = 1,2$ if $a$ global
Fact  Asynchronous parallel composition $||$ is associative.
(the notation $A_1 \ || \ A_2 \ || \ \ldots \ || \ A_n$ is therefore well defined)

Fact  \ldots and commutative modulo bisimulation:  $A_1 \ || \ A_2 \ \approx \ A_2 \ || \ A_1$

Fact  For the transition relation of $A_1 \ || \ A_2 \ || \ \ldots \ || \ A_n$ we have:

Let $\Psi(a) \subseteq \{1, \ldots, n\}$ be the set of indices $i$ with $a \in \Sigma_i$.

Let $\overline{\Psi(a)}$ be its complement.

\[
(s_1, \ldots, s_n) \xrightarrow{a} (s'_1, \ldots, s'_n) \text{ iff } s_i \xrightarrow{a} s'_i \text{ for all } i \in \Psi(a) \neq \emptyset \text{ and } s'_j = s_j \text{ for all } j \in \overline{\Psi(a)}
\]
Full Asynchronous Composition

**Definition**  For two LTS $A_1$ and $A_2$ the *full asynchronous* parallel composition $A = A_1 \parallel A_2$ consists of the following components:

\[
S = S_1 \times S_2, \quad \Sigma = \mathcal{P}(\Sigma_1 \cup \Sigma_2), \quad I = I_1 \times I_2, \quad T \text{ is defined as follows:}
\]

\[
s \xrightarrow{a} s' \text{ in } A_1 \text{ and } t' = t \quad \text{if } M = \{a\} \subseteq \Sigma_1 \setminus \Sigma_2
\]

\[
t \xrightarrow{b} t' \text{ in } A_2 \text{ and } s' = s \quad \text{if } M = \{b\} \subseteq \Sigma_2 \setminus \Sigma_1
\]

\[
(s, t) \xrightarrow{M} (s', t') \text{ in } A \quad \text{iff} \quad s \xrightarrow{a} s' \text{ in } A_1 \text{ and } t \xrightarrow{a} t' \text{ in } A_2 \quad \text{if } M = \{a\} \subseteq \Sigma_1 \cap \Sigma_2
\]

\[
s \xrightarrow{a} s' \text{ in } A_1 \text{ and } t \xrightarrow{b} t' \text{ in } A_2 \text{ if } a \in \Sigma_1 \setminus \Sigma_2
\]

and $b \in \Sigma_2 \setminus \Sigma_1$

and $M = \{a, b\}$
• extension of full asynchronous composition to arbitrary many components:

  \[ \Sigma = \mathcal{P}(\Sigma_1 \cup \cdots \cup \Sigma_n), \quad T = \ldots \]

  – synchronization on multiple global symbols in parallel is possible
  – exponential increase in alphabet size

• interleaving as simplification

  – \textbf{Fact} same set of reachable states
  – lengths of paths between states may differ but \ldots
  – \ldots interleaving model is not exact with respect to relative speed of components
State Explosion due to Interleaving

idea  follow only one out of 8 possible paths, e.g. just the red or green one
\[ \Sigma_i = \{a_i, s\}, \quad \Sigma = \{a_1, \ldots, a_n, s\}, \quad \Lambda_i = \{a_i\}, \quad \Gamma = \Gamma_i = \{s\} \]

- number states: \( |S| = |\{0, 1, 2\}^n| = 3^n \).
- number reachable states: \( |\{0, 1\}^n \cup \{2\}^n| = 2^n + 1 \)
- number of necessary states: \( |(1^*0^* \cap \{0, 1\}^n) \cup \{2\}^n| = (n + 1) + 1 \)
Synchronous Composition with Checker-Automaton

\[ C \times (A \parallel B) \]
• checker has to be “invariant” against omitted transitions
  – removing transitions may not alter reachability of final states
  – reduction is depends on checker

• achieving maximum reduction can not be the goal:
  – only possible if reachability of final states is known
  – then there is no gain

• goal is to use a simple criteria that allows to remove transitions
  – best case: these situations can be determined statically …
  – … or efficiently dynamically during search
**Definition**  
A State \( s = (s_1, \ldots, s_n) \) in \( A_1 \mid \cdots \mid A_n \) is called **local** to \( A_i \) iff all transitions in \( A_i \) with \( s_i \) as source are local to \( A_i \) and such a transition exists.

**Definition**  
A symbol \( a \) is commutative with a symbol \( b \) in state \( s \) iff for all \( s', s'' \) with \( s \xrightarrow{a} s' \) and \( s \xrightarrow{b} s'' \) there is a state \( t \) with \( s' \xrightarrow{b} t \) and \( s'' \xrightarrow{a} t \).

**Fact**  
Let \( s \) be local to \( A_i \). Then all \( \Lambda_i \) are commutative to all other \( \Sigma \setminus \Lambda_i \).
**Definition**  An *expansion* of a state is a subset of its successors.

BFS/DFS iterates only over expansions of `current` in inner loop

```c
partial_order_recursive_dfs_aux (Stack stack, State current)
{
  ...
  
forall  next in expansion (current)
    partial_order_recursive_dfs_aux (stack, next);
  ...
}
```

**Definition**  A *partial* expansion is a proper subset of the successors.
Definition  

*Local partial order reduction* uses the following expansion for a state $s$:

- exactly the local transitions of one $A_i$, if $s$ is local to $A_i$
  
  (this is a partial expansion in general)

- if there are multiple such $A_i$, then choose an arbitrary one

- if there is no such $A_i$, use all successors
  
  (this is a non partial or full expansion)
Locally Equivalent Traces
also called “stutter equivalent”

Definition two traces \( w \) and \( w' \) are locally equivalent, written \( w \approx_l w' \), iff they are identical after removing all local symbols.

Fact the local equivalence \( \approx_l \) is in deed an equivalence relation.

Proof

- reflexivity and symmetry are easy.

- write \( w|_{\Sigma'} \) for the trace \( w \) after removing symbols from \( \Sigma' \).

- transitivity: from \( w_1 \approx_l w_2, w_2 \approx_l w_3 \) follows \( w_1|_{\Gamma} = w_2|_{\Gamma} = w_3|_{\Gamma} \) and therefore \( w_1 \approx_l w_3 \).
**Definition**  
Checker $C$ ignores local symbols iff  
$s \xrightarrow{a} = \{s\}$ in $C$ for all local $a \in \Lambda$  
(also called “$a$ is invisible for $C$”)

**Fact**  
Let $C$ ignore local symbols and $w \approx_I w'$, then  
$w \in L(C) \iff w' \in L(C)$.

**Proof**  
For $w \in L(C)$, a state sequence that accepts $w$ also accepts $w'$ and vice versa.

Search with partial order reduction only needs to traverse one representative of each equivalence class of $\approx_I$. 
Problem

- partial expansion “delays” execution of transitions of other $A_j$
- those could be delayed forever even though they are executable

Solution

- cycles with only partial expansion need to be broken by fully expanding one state

- easy to implement in DFS:
  - each cycle is closed through a “back edge” to a state on the search stack
  - if a “back edge” is found the current state is fully expanded

- approximation in BFS: full expansion if an edge to an earlier generation is found.
• instead of synchronization through actions:
  – synchronization through global variables
  – and/or synchronization through monitors/semaphores
  – and/or synchronization through messages/channels

• partial order reduction can be applied using the following concepts:
  – independence resp. commutativity of statements
    * read/write and write/read dependent, read/read independent
    * similar approach for messages (read = receive, write = send)
  – invisibility of local statements
• Message Passing
  – communication with messages over channels/buffers
  – note: finite models always have channels with finite capacity

• independent or local operations (2 processes, 1 channel):
  – read from channel, which is **not full**
  – write into a channel, which is **not empty**

• dependent or global operations (2 processes, 1 channel):
  – read from a **full** channel
  – write into an **empty** channel
Liveness and Fairness

- **liveness**
  - opposite of safety (not “security”)
  - describes **unavailable** behavior
  - usually only makes sense, if concrete timing is abstracted away: concentrate on potential sequences of events

- **deadlock** is still a safety property
  - “no state without successor is reachable”

- **livelock** as a generic liveness property:
  - “system is locked up in an endless loop with *real* progress”
More Liveness Examples

• termination of programs/processes/protocols:
  – quicksort terminates
  – IEEE Firewire: initialization phase terminates with a proper topology

• expected events really happen:
  – operations in super scalar processors eventually “retire”
  – elevator eventually shows up, if called

• in these examples there are no time limits
Counterexamples for Liveness Properties

- **finite** state systems:
  
  ![Diagram of a lasso with stem and no progress]

- **infinite** state systems:

  - “divergence” possible: counter example may not have lasso shape
  - for instance an incrementing counter over natural numbers
Fairness Intuition

- abstraction of concrete scheduler:
  - order of process execution is arbitrary in the model
  - applies to interleaving and full asynchronous composition

- this abstraction may lead to spurious resp. artificial counter examples for liveness:
  - not executing a process even though executable may produce spurious live lock

- fairness:
  - scheduler does not ignore an (executable) process forever ...
  - … without specifying a concrete scheduler
    (which would be incorrect anyhow, because relative speed of processes unknown)
Definition a *fair LTS* $A = (S, I, \Sigma, T, F)$ is an ordinary LTS $(S, I, \Sigma, T)$, with $F \subseteq T$ a set of *fair transitions* of $A$.

(a transition is represented as $(s, a, s')$)

**Definition** an infinite path $\pi = s_0 \xrightarrow{a_0} s_1 \rightarrow \ldots$ in a fair LTS is *fair* iff

$\pi$ contains infinitely many fair transitions: $|\{i \mid (s_i, a_i, s_{i+1}) \in F\}| = \infty$.

**Example** choose $F$ as the set of transitions in which a deterministic component $A_j$ either does a local or global transition, or is “disabled” ($s \not\xrightarrow{a}$ for all $a \in \Sigma_j$). A fair path in $A_1$ according to $F$ in $A_1 \parallel \cdots \parallel A_n$ has to execute $A_j$ over and over again.
\( F = \text{all } \{b, r, s\}\)-transitions, for which the checker stays in the middle state

(counter example, that an \(a\) has to occur after \(r\) eventually)

there is a fair path with trace \(r(bs)^\omega = rbsbsbs\ldots\)

(in which \(A\) is executed once)
• brute force (results in a quadratic algorithm):
  – back edge: edge from current to state on stack during recursive DFS
  – cycle closed by back edge is fair iff the cycle contains a fair transition

• SCC = strongly connected component
  – max. set of nodes of a directed graph (= state space),
    in which every node is reachable from every other node in the set

• every cycle (incl. the cyclic part of any lasso) is contained in an SCC
  – SCCs can be found by DFS:
    linear algorithm by Tarjan for decomposing a directed graph into its SCCs
for each node/state calculate

1. *depth first search index* (DFSI): order nodes as they are discovered

2. min. reachable DFSI through back edges (MRDFSI), initialized by DFSI

determine DFSI in prefix phase of DFS  (before successors are visited)

push each newly reached node on auxiliary stack

minimize MRDFSI over the MRDFSI of each node and its immediate children
(during suffix or post-fix phase, but not over children already in an SCC)

after visiting children of current node with MRDFSI = DFSI:

- pop from aux. stack until current node is popped

- all popped nodes in this last step form an SCC
forall nodes N do dfsi[N] = 0, mrdfsi[N] = INF;
S = empty stack;
I = 0; forall nodes N do I = tarjan (N, I, S);

tarjan (N, I, S)
    if (dfsi[N] != 0) return I;
    mrdfsi[N] = dfsi[N] = ++I;
    S.push (N);
    forall successors M of N do I = tarjan (M, I, S);
    forall successors M of N do
        mrdfsi[N] = min (mrdfsi[N], mrdfsi[M]);
    if (dfsi[N] != mrdfsi[N]) return I;
    do M = S.pop (), scc[M] = N, mrdfsi[M] = INF; while (M != N);
    return I;
Generalized Fairness

**Problem** prev. example without *fair* scheduler, allows spurious counter example

**Definition** a general fair LTS \( A = (S, I, \Sigma, T, F_1, \ldots, F_n) \) is an LTS \((S, I, \Sigma, T)\), with \( F_i \subseteq T \) a family of fairness constraints.

**Definition** transition \((s, a, s')\) is *fair* for the fairness constraint \( F_i \) iff \((s, a, s')\) \(\in F_i\).

**Definition** an infinite path \( \pi = s_0 \xrightarrow{a_0} s_1 \rightarrow \ldots \) in a general fair LTS is *fair* iff \( \pi \) contains infinitely many fair transitions for each fairness constraint \( F_i \): \(|\{i \mid (s_j, a_i, s_{j+1}) \in F_i\}| = \infty\).

**Example Cont.** choose as second fairness constraint all transitions which are either local to \( A \) or global. Then no fair path exists and there is no counter example for the property that \( r \) has to be followed by an \( a \) eventually.
• similar algorithm as just for one fairness constraint:
  – while closing cycles (through back edges)
    check whether cycle contains all fairness constraints
  – or alternatively check if there is an SCC,
    which contains a fair transition for each fairness constraint

• reduction of general fairness to (single) fairness:
  “counter construction”
  – order fairness constraints, e.g. $F_1 < \ldots < F_n$
  – cross product with modulo $n$ counter which goes from $i$ to $(i + 1) \mod n$ iff
    the transition of the (original) LTS is fair with respect to $F_{i+1}$
  – new single fairness constraint consists of all transitions
    in which the counter goes from $n - 1$ to $0$
• abstract data type “boolean logic”:
  constructors: for boolean constants and variables
  operations: conjunction, disjunction, negation, …
  queries: test on satisfiability, tautology …

• fundamental data type in EDA (Electronic Design Automation) tools:
  simulators, synthesis, optimization, compilation, verification, …

• trade off between fast operations versus space usage
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
  f & = \quad & 0 & 0 & 0 & 0 \\
  g & = & 0 & 0 & 1 & 1 \\
  f \lor g & = & 0 & 0 & 1 & 1 \\
  \neg f & = & 0 & 0 & 1 & 0 \\
\end{align*}
\]

point wise operations
- function table always has $2^n$ rows for $n$ variables (grows exponential)

- operations are linear in the size of its operands:
  - e.g. conjunction generates function table of same size

- representation is **canonical**:
  - two semantically equivalent boolean formulas have identical function tables

- queries are linear:
  - tautology: check that there is no row with 0 as result
  - satisfiability: find at least one row with 1 as result
• potentially more compact than function table
  – only depends on number of prime implicants
  – minimizing is still NP hard (Quine-McCluskey, heuristics methods: Espresso)

• simple implementation as 2-level circuit (PLA)

• operation of disjunction is linear (without minimization)

• conjunction quadratic, negation exponential (even without minimization)

• satisfiability check has constant complexity:
  DNF satisfiable iff DNF has at least one cube
Exponential DNF for Parity

- no combination of blocks in KV diagram possible
- only \textit{full} cubes (= min terms) as prime implicants (with max. number of literals)
- DNF for parity of $n$ variables has $2^{n-1}$ cubes
Conjunction on DNF

\[(a \cdot \overline{b} \lor \overline{a} \cdot b \cdot \overline{c}) \land (a \cdot \overline{b} \lor \overline{b} \cdot \overline{c})\]

1st Operand

2nd Operand

multiply out

\[a \cdot \overline{b} \cdot a \cdot \overline{b} \lor a \cdot \overline{b} \cdot \overline{b} \cdot \overline{c} \lor \overline{a} \cdot b \cdot \overline{c} \cdot a \cdot \overline{b} \lor \overline{a} \cdot b \cdot \overline{c} \cdot \overline{b} \cdot \overline{c}\]

simplify

\[a \cdot \overline{b} \lor a \cdot \overline{b} \cdot \overline{c}\]

minimize, e.g. use Quine-McCluskey

\[a \cdot \overline{b}\]
Conjunction on DNF cont.

\[(a \lor b \lor c) \land (d \lor e \lor f)\]

1st operand \hspace{1cm} 2nd operand

distribution of “\(\land\)” over “\(\lor\)” results in

\[a \cdot d \lor a \cdot e \lor a \cdot f \lor b \cdot d \lor b \cdot e \lor b \cdot f \lor c \cdot d \lor c \cdot e \lor c \cdot f\]

no further simplifications!

example can be generalized:

the DNF for the conjunction of two DNFs with \(n\) resp. \(m\) cubes has \(O(n \cdot m)\) cubes
• net list for combinational circuit:
  hyper graph with *gates* as nodes and *signals* as hyper edges
  (hyper edge: set of nodes connected to this edge)

• parse tree of a boolean formula

• sharing of common sub formulas is more compact:
  carry out of a ripple-adders as tree is exponential in bit width
  even though it can be implemented with a linear sized circuit

• parse DAG (directed acyclic graph) for combinational logic
Parse DAG of Full Adder

VAR \( x \) VAR \( y \) VAR \( i \)

pointer to node representing the sum output

pointer to node representing the carry output

OR

XOR

AND

VAR \( x \)

VAR \( y \)

VAR \( i \)
enum Tag
{
  OR, AND, XOR, NOT, VAR, CONST
};

typedef struct Node Node;
typedef union NodeData NodeData;

union NodeData
{
  Node *child[2];
  int idx;
};

struct Node
{
  enum Tag tag;
  NodeData data;
  int mark;                     /* traversal */
};
- similar symbols in symbol table and expressions of compilers

- variable are encoded with integer indices

- boolean constants are represented with index 0 resp. 1

- operations nodes have pointers to their operands

- nodes can be shared
  
  (memory management: reference counting or garbage collection)

- no cycles (DAG)
Node *
new_node_val (int constant_value)
{
    Node *res;

    res = (Node *) malloc (sizeof (Node));
    memset (res, 0, sizeof (Node));
    res->tag = CONST;
    res->data.idx = constant_value;

    return res;
}

usually only 0 and 1 as values for constants
Node *
new_node_var (int variable_index)
{
    Node *res;

    res = (Node *) malloc (sizeof (Node));
    memset (res, 0, sizeof (Node));
    res->tag = VAR;
    res->data.idx = variable_index;

    return res;
}

variables are distinguished by their index
Node *
new_node_op (enum Tag tag, Node * child0, Node * child1)
{
    Node *res;

    res = (Node *) malloc (sizeof (Node));
    memset (res, 0, sizeof (Node));
    res->tag = tag;
    res->data.child[0] = child0;
    res->data.child[1] = child1;

    return res;
}

operator type as first argument

(assumption: child1 is 0 for negation operator)
Node *x, *y, *i, *o, *s, *t[3];

x = new_node_var (0);
y = new_node_var (1);
i = new_node_var (2);
t[0] = new_node_op (XOR, x, y);
t[1] = new_node_op (AND, x, y);
t[2] = new_node_op (AND, t[0], i);
s = new_node_op (XOR, t[0], i);
o = new_node_op (OR, t[1], t[2]);

explicit sharing through temporary pointers t[0], t[1] and t[2]
Example Algorithm: Input-Cone Variables

```c
void
input_cone_node (Node * node)
{
    if (node->mark)
        return;
    node->mark = 1;
    switch (node->tag)
    {
    case CONST:
        break;
    case VAR:
        printf("variable %d in input cone\n", node->data.idx);
        break;
    case NOT:
        input_cone_node (node->data.child[0]);
        break;
    default: /* assume binary operator */
        input_cone_node (node->data.child[0]);
        input_cone_node (node->data.child[1]);
        break;
    }
}
```
void
mark_node (Node * node, int new_value)
{
    if (node->mark == new_value)
        return;

    node->mark = new_value;
    switch (node->tag)
    {
        case VAR:
        case CONST:
            return;
        case NOT:
            mark_node (node->data.child[0], new_value);
            break;
        default:
            mark_node (node->data.child[0], new_value);
            mark_node (node->data.child[1], new_value);
            break;
    }
}
• algorithms are essentially a variant of DFS

• avoid multiple visits with mark flag

• usually two phases: traversal, reset of mark flags

• conjunction, negation are fast (simply use op)

• tautology and satisfiability checks are hard

• explicit sharing: not canonical representation
And-Inverter-Graphs (AIG)

- basic logical operators: conjunction and negation

- DAG representation:
  - operator nodes are all conjunctions
  - negation/sign as *edge attribute*
  - (bit stuffing: compactly stored as LSB in pointer)

- automatic sharing of isomorphic sub graphs

- simplification rules with constant time look ahead
negation/sign are edge attributes

(not part of node)
**typedef struct** AIG AIG;

**struct** AIG
{
  enum Tag tag;                 /* AND, VAR */
  void *data[2];
  int mark, level;              /* traversal */
  AIG *next;                    /* hash collision chain */
};

**#define** sign_aig(aig) (1 & (unsigned) aig)
**#define** not_aig(aig) ((AIG*)(1 ^ (unsigned) aig))
**#define** strip_aig(aig) ((AIG*)(~1 & (unsigned) aig))
**#define** false_aig ((AIG*) 0)
**#define** true_aig ((AIG*) 1)

**assumption for correctness:**

sizeof(unsigned) == sizeof(void*)
Advantages of Signs as Edge Attributes

- alignment of modern processors “wastes” several LSBs
  alignment is typically 4 or 8 bytes ⇒ 2 or 3 LSBs left
  `malloc` returns *aligned blocks*, e.g. 8 Byte aligned on Sparc

- negated and not negated formula represented by the same node
  (potentially reduces memory usage by half)

- maximal reduction to one operator (AND)

- negation extremely efficient (bit LSB of pointer)

- allows additional simplification rules
  constant time detection of arguments with opposite sign
int simp_aig (enum Tag tag, void *d0, void *d1, AIG ** res_ptr) {
  if (tag == AND) {
    if (d0 == false_aig || d1 == false_aig || d0 == not_aig (d1)) {
      *res_ptr = false_aig; return 1; }
    if (d0 == true_aig || d0 == d1) {
      *res_ptr = d1; return 1; }
    if (d1 == true_aig) {
      *res_ptr = d0; return 1; }
  }
  return 0;
}
merge nodes with same children
• is also called *algebraic reduction*

• main advantage: automatic sharing, thus less memory usage

• implementation:
  
  nodes stored in a hash table (unique table)

• on-the-fly reduction:

  invariant: two nodes always have different children

  before generating a new node search for already existing equivalent node

  if search is successful return equivalent node
Unique-Table

Collision-Chain

AND

VAR 1

Collision-Chain

next

next

next

next


```c
#define UNIQUE_SIZE (1 << 20)
AIG *unique[UNIQUE_SIZE];

AIG **
find_aig (enum Tag tag, void *d0, void *d1)
{
    AIG *r, **p;
    unsigned h = (tag + ((int) d0) * 65537 + 13 * (int) d1);
    h = h & (UNIQUE_SIZE - 1);    /* modulo UNIQUE_SIZE */
    for (p = unique + h; (r = *p); p = &r->next)
        if (r->tag == tag && r->data[0] == d0 && r->data[1] == d1)
            break;

    return p;
}
```

1. compute hash value as combination of tags and pointer values (resp. variable index or constant value)

2. normalize hash value to table size (modulo table size)

3. search through collision chain starting at the normalized hash value

4. compare nodes with node that is to be generated

5. if these are the same return pointer to the link pointing to the existing node

6. otherwise return pointer to last (empty) link field in collision chain
**void**

insert_aig (AIG * aig)
{
    AIG **p;
    int  l[2];

    p = find_aig (aig->tag, aig->data[0], aig->data[1]);
    assert (!*p);
    aig->next = *p;
    *p = aig;

    if (aig->tag == AND)
    {
        l[0] = strip_aig (aig->data[0])->level;
        l[1] = strip_aig (aig->data[1])->level;
        aig->level = 1 + ((l[0] < l[1]) ? l[1] : l[0]);
    }
    else
        aig->level = 0;
}

**find_aig** returns “new position” of hashed node
AIG *
new_aig (enum Tag tag, void *data0, void *data1)
{
  AIG *res;
  if (tag == AND && data0 > data1)
    SWAP (data0, data1);
  if (tag == AND && (simp_aig (tag, data0, data1, &res)))
    return res;
  if ((res = *find_aig (tag, data0, data1)))
    return res;
  res = (AIG *) malloc (sizeof (AIG));
  memset (res, 0, sizeof (AIG));
  res->tag = tag;
  res->data[0] = data0;
  res->data[1] = data1;
  insert_aig (res);
  return res;
}

AIG *
var_aig (int variable_index)
{
    return new_aig (VAR, (void *) variable_index, 0);
}

AIG *
and_aig (AIG * a, AIG * b)
{
    return new_aig (AND, a, b);
}

AIG *
or_aig (AIG * a, AIG * b)
{
    return not_aig (and_aig (not_aig (a), not_aig (b)));
}

AIG *
xor_aig (AIG * a, AIG * b)
{
    return or_aig (and_aig (a, not_aig (b)), and_aig (not_aig (a), b));
}
int
count_aig (AIG * aig)
{
    if (sign_aig (aig))
        aig = not_aig (aig);
    if (aig->mark)
        return 0;
    aig->mark = 1;
    if (aig->tag == AND)
        return count_aig (aig->data[0]) + count_aig (aig->data[1]) + 1;
    else
        return 1;
}

be carefull to handle signs
otherwise simple DFS as for DAG representation
void
mark_aig (AIG * aig, int new_value)
{
    if (sign_aig (aig))
        aig = not_aig (aig);
    if (aig->mark == new_value)
        return;
    aig->mark = new_value;
    if (aig->tag == AND)
    {
        mark_aig (aig->data[0], new_value);
        mark_aig (aig->data[1], new_value);
    }
}

less cases and less code as for DAG representation!
AIG *
node2aig (Node * node)
{
  
switch  (node->tag)
    {
    
case  VAR:
      return  new_aig (VAR, ( void  *) node->data.idx, 0);
    case  CONST:
      return  node->data.idx ? true_aig : false_aig;
    case  AND:
      return  and_aig (node2aig (node->data.child[0]),
                       node2aig (node->data.child[1]));
    case  OR:
      return  or_aig (node2aig (node->data.child[0]),
                      node2aig (node->data.child[1]));
    case  XOR:
      return  xor_aig (node2aig (node->data.child[0]),
                      node2aig (node->data.child[1]));
    default:
      assert (node->tag == NOT);
      return  not_aig (node2aig (node->data.child[0]));
    }
  }
}
• more robust C code
  (for instance 64 Bit conformance)

• memory management
  (either use reference counting or garbage collection)

• input format, parser

• more complex simplification rules for grand children
  (see our paper BrummayerBiere-MEMICS’06)
both formulas and AIGs represent XOR of $x$ and $y$

(multiply out right formula and simplify result to obtain left one)

in general there are multiple AIGs for the same boolean function
• based on ternary base operation ITE (if-then-else):
  condition is always a variable

• go back to Shannon also called Shannon graphs

• mainly used in the version of ROBDDs
  Reduced Ordered Binary Decision Diagrams

• [Bryant86] showed canonicity of ROBDDs:
  each boolean function has exactly one ROBDD modulo variable order
Graphical Notation

Nodes are marked with condition variable.

ELSE successor is reached through read dashed line.

THEN successor is reached through solid line.

Boolean constants: TRUE, FALSE.

Example:
- $x$: condition variable
- $y$: successor
- Nodes marked with: $0$, $1$

Model Checking #342234 WS 2018 Armin Biere JKU Linz
• inner nodes are ITE, leafs are boolean constants

• notation $\text{ite}(x, f_1, f_0)$ means $\text{if } x \text{ then } f_1 \text{ else } f_0$
  (note that ELSE argument $f_0$ follows $f_1$ despite reverse indices)

• semantic $\text{eval}$ produces boolean expressions out of a BDD

$$
\text{eval}(0) \equiv 0 \\
\text{eval}(1) \equiv 1 \\
\text{eval}(\text{ite}(x, f_1, f_0)) \equiv x \cdot \text{eval}(f_1) \lor \overline{x} \cdot \text{eval}(f_0)
$$

• BDDs are again algebraically reduced DAGs
  (max. sharing of isomorphic sub graphs as for AIGs)

• negated edges are also possible
From Function Table to BDD

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \oplus y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

decision tree

decision diagram (DAG)
Algebraic Reduction Rule

max. sharing of isomorphic sub graphs
Semantic Reduction Rule

elimination of redundant nodes
• apply both rules exhaustively
  (that gives reduced BDDs: the “R” in “ROBDDs”)

• variables on paths from the root to leaves are ordered always in the same way
  (that gives ordered BDDs: the “O” in “ROBDDs”)

• these assumptions make ROBDDs canonical modulo the variable order
  – different orders usually lead to different ROBDDs
  – variable order has great influence ROBDD size for a given function

• **in the following we always mean ROBDD, when we say BDD**
**boolean function/expression:**

\[ \bigwedge_{i=0}^{n-1} x_i = y_i \]

**interleaved variable order:**

\[ x_3 > y_3 > x_2 > y_2 > x_1 > y_1 > x_0 > y_0 \]

Comparison of two \( n \)-bit-vectors needs \( 3 \cdot n \) inner nodes for the interleaved variable order.
missing edges lead to 0
Variable Orders

- exponential difference between variable orders

- there also exist exponential functions:
  BDD is always exponential in size, e.g. middle output bit of multiplier circuits

- heuristics for static variable ordering:
  order the variables with DFS as they occur in the DAG/AIG
  (as for instance in input_cone_aig)

- dynamic reordering of variables
  based on in place exchange of neighboring variables
canonicity of BDDs results in:

- BDD is a tautology, iff it is only made of the 1 leaf
- BDD is satisfiable, iff it is not only made of the 0 leaf
- two BDDs are equivalent, iff they are isomorphic
  (constant time pointer comparison if unique table as in AIGs is used)

**Question:** where is the NP completeness of satisfiability?

**Answer:** hidden in the effort to construct the BDD.
Shannon's Expansion Theorem

\[ f(x) \equiv x \cdot f(1) \lor \overline{x} \cdot f(0) \]

let \( x \) the top most variable of two BDDs \( f \) and \( g \):

\[ f \equiv \text{ite}(x, f_1, f_0) \quad g \equiv \text{ite}(x, g_1, g_0) \]

with \( f_i \) resp. \( g_i \) the children of \( f \) and \( g \) for \( i = 0, 1 \).

\[ f(0) = f_0 \quad g(0) = g_0 \quad f(1) = f_1 \quad g(1) = g_1 \]

Because of the \( R \) in ROBDD \( x \) occurs only at the top of \( f \) and \( g \).

\[ (f@g)(x) \equiv x \cdot (f@g)(1) \lor \overline{x} \cdot (f@g)(0) \]
\[ \equiv x \cdot (f(1)@g(1)) \lor \overline{x} \cdot (f(0)@g(0)) \]
\[ \equiv x \cdot (f_1@g_1) \lor \overline{x} \cdot (f_0@g_0) \]

where \( @ \) is any binary boolean operation, such as \( \land, \lor, \oplus, \ldots \)

**recursive algorithm to compute operations on BDDs**
typedef struct BDD BDD;

struct BDD
{
  int idx, mark;
  BDD *child[2], *next;
};

#define sign_bdd(ptr) (1 & (unsigned) ptr)
#define strip_bdd(ptr) ((BDD*) (~1 & (unsigned) ptr))
#define not_bdd(ptr) ((BDD*) (1 ^ (unsigned) ptr))
#define true_bdd ((BDD*) 1)
#define false_bdd ((BDD*) 0)

#define is_constant_bdd(ptr) \
  ((ptr) == true_bdd || (ptr) == false_bdd)
#define UNIQUE_SIZE (1 << 20)
BDD *unique[UNIQUE_SIZE];

BDD **
find_bdd (int idx, BDD * c0, BDD * c1)
{
  BDD *r, **p;
  unsigned h = (idx + ((int) c0) * 65537 + 13 * (int) c1);
  h = h & (UNIQUE_SIZE - 1);
  for (p = unique + h; (r = *p); p = &r->next)
    if (r->idx == idx && r->child[0] == c0 && r->child[1] == c1)
      break;

  return p;
}
Normalisation of BDD Nodes with Edge Attributes

\[
\begin{align*}
\text{ite}(x, f_1, f_0) & \equiv x \cdot f_1 \lor \overline{x} \cdot f_0 \\
& \equiv (\overline{x} \lor f_1) \cdot (x \lor f_0) \\
& \equiv x \cdot f_1 \lor \overline{x} \cdot f_0 \lor f_1 \cdot f_0 \\
& \equiv x \cdot f_1 \lor \overline{x} \cdot f_0 \\
& \equiv \text{ite}(x, f_1, f_0)
\end{align*}
\]
BDD *
new_bdd_aux (int idx, BDD * c0, BDD * c1)
{
    BDD *res;

    assert (!sign_bdd (c0));

    if ((res = *find_bdd (idx, c0, c1)))
        return res;

    res = (BDD *) malloc (sizeof (BDD));
    memset (res, 0, sizeof (BDD));
    res->idx = idx;
    res->child[0] = c0;
    res->child[1] = c1;
    *find_bdd (idx, c0, c1) = res;

    return res;
}
BDD *
new_bdd (int idx, BDD * c0, BDD * c1)
{
  BDD *res;
  int  sign;

  if  (c0 == c1)
    return  c0;
  if  ((sign = sign_bdd (c0)))
  {
    c0 = not_bdd (c0);
    c1 = not_bdd (c1);
  }
  res = new_bdd_aux (idx, c0, c1);
  return  sign ? not_bdd (res) : res;
}
Different Top Indices

\[(f @ g)(x) \equiv x \cdot (f_1 @ g) \lor \overline{x} \cdot (f_0 @ g)\]

if top variables do not match only one argument is split
int
top_idx_bdd (BDD * a, BDD * b)
{
    int res[2];
    res[0] = (is_constant_bdd (a)) ? -1 : strip_bdd (a)->idx;
    res[1] = (is_constant_bdd (b)) ? -1 : strip_bdd (b)->idx;
    return res[res[0] < res[1]];
}
BDD *
cofactor (BDD * bdd, int pos, int idx)
{
    BDD *res;
    int sign;
    if (is_constant_bdd (bdd))
        return bdd;
    if ((sign = sign_bdd (bdd)))
        bdd = not_bdd (bdd);
    res = (bdd->idx == idx) ? bdd->child[pos] : bdd;
    return sign ? not_bdd (res) : res;
}
void
cofactor2 (BDD * a, BDD * b, BDD * c[2][2], int *idx_ptr)
{
    int idx = *idx_ptr = top_idx_bdd (a, b);
    c[0][0] = cofactor (a, 0, idx);
    c[0][1] = cofactor (a, 1, idx);
    c[1][0] = cofactor (b, 0, idx);
    c[1][1] = cofactor (b, 1, idx);
}
Functors for Apply

BDD *
basic_and (BDD * a, BDD * b)
{
    assert (is_constant_bdd (a) && is_constant_bdd (b));
    return (BDD *) (((unsigned) a) & (unsigned) b);
}

BDD *
basic_or (BDD * a, BDD * b)
{
    assert (is_constant_bdd (a) && is_constant_bdd (b));
    return (BDD *) (((unsigned) a) | (unsigned) b);
}

BDD *
basic_xor (BDD * a, BDD * b)
{
    assert (is_constant_bdd (a) && is_constant_bdd (b));
    return (BDD *) (((unsigned) a) ^ (unsigned) b);
}
typedef BDD *(*BasicFunctor) (BDD *, BDD *);

BDD *
apply (BasicFunctor op, BDD * a, BDD * b)
{
  BDD *tmp[2], *c[2][2];
  int idx;
  if (is_constant_bdd (a) && is_constant_bdd (b))
    return op (a, b);
  cofactor2 (a, b, c, &idx);
  tmp[0] = apply (op, c[0][0], c[1][0]);
  tmp[1] = apply (op, c[0][1], c[1][1]);
  return new_bdd (idx, tmp[0], tmp[1]);
}
Other Constructors

BDD *
and_bdd (BDD * a, BDD * b)
{
    return apply (basic_and, a, b);
}

BDD *
or_bdd (BDD * a, BDD * b)
{
    return apply (basic_or, a, b);
}

BDD *
xor_bdd (BDD * a, BDD * b)
{
    return apply (basic_xor, a, b);
}

BDD *
var_bdd (int idx)
{
    return new_bdd (idx, false_bdd, true_bdd);
}