Formal methods
Tutorial I.

http://d3s.mff.cuni.cz

Jan Kofroň

Department of Distributed and Dependable Systems

FACULTY OF MATHEMATICS AND PHYSICS
Charles University
Model Checking

- Markov chains
- Timed automata
- Labelled transition system
- Kripke structure

Model

Property specification: $AG(\text{start} \rightarrow AF \text{heat})$

Model checker

Property satisfied

Error report

Property violated
What makes model-checking software difficult?

Problems using existing checkers:

- Model Construction
- Property specification
- State explosion
- Output interpretation

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
What makes model-checking software difficult?

Problems using existing checkers:
- Model Construction
- Property specification
- State explosion
- Output interpretation

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
What makes model-checking software difficult?

Problems using existing checkers:
- Model Construction
- Property specification
- State explosion
- Output interpretation
What makes model-checking software difficult?

Problems using existing checkers:
- Model Construction
- Property specification
- State explosion
- Output interpretation

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
\[ x := 0; \quad y := 0; \]

\textbf{for} \ i := 1 \ \textbf{to} \ 3 \ \textbf{do} \quad \begin{aligned} & (x := x + 1; \quad y := y + 1) \\ & x := x + 1 \\ & y := 0 \\ & x := 0 \\ & y := y + 1 \end{aligned}
LTS semantics

- Semantics \(\sim\) equivalence
  - Many ways to define equivalence
    - How alternatives (branches) in LTS are handled
  - Branching spectrum
    - Trace
    - Failures
    - . . .
    - Simulation
    - Strong bisimilarity
LTS: Linear time/Branching time equivalence spectrum

- strong bisimilarity
  - 2-nested simulation equivalence
    - ready simulation equivalence
      - possible-futures equivalence
      - ready trace equivalence
        - readiness equivalence
        - failure trace equivalence
          - failure equivalence
            - completed trace equivalence
              - trace equivalence
          - simulation equivalence
LTS specification – Process Algebras

- **Process Algebra**
  - (A) theory of concurrency
  - Way to capture LTS
  - Equational reasoning
  - Most known
    - CSP (Hoare), CCS (Milner), **ACP** (Bergstra, Klop),
    - $\pi$-Calculus (Milner, Parrow, Walker)
  - Application areas – LTS modeling:
    - concurrent systems
    - communication protocols
    - electronic circuits, production systems, biochemical processes
Modeling (program) behavior

- Markov chains
- Timed automata
- Labelled transition system
- Kripke structure

Open
Start
Empty
Close
Empty
Close
Start
Close
Start
Heat

Property specification: $\textit{AG(start \rightarrow AF heat)}$

Model
Model checker

Property satisfied
Error report

Property violated

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
Modeling (program) behavior

Markov chains
Timed automata
Labelled transition system
Kripke structure

Model

Property specification

\[ AG(\text{start} \rightarrow \text{AF heat}) \]

Model checker

Property satisfied

Property violated

Error report

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
Modeling (program) behavior

- Model (of a program) – various types
  - LTS
  - Kripke Structure
  - Markov chains
  - Timed automata
  - ...

- Property specification
- Model checker
- Model
- Property satisfied
- Property violated
- Error report

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
Program states

- Program state: program counter (location) + variable values + stack(s) + heap.
  - Stack, Heap: ignore for now
  - Program counter: low-level detail, language dependent
  - Graphical representation: different nodes are different states

- In previous example:
  - \( x:=0; y:=0; \)
  - for \( i:=1 \) to \( 3 \) do
    - \( (x:=x+1; y:=y+1) \)
  - If “steps” are assignments, STS has 8 steps, and 9 states: the initial state and the state after each step
Kripke structure

- Basics: A type of STS
  - Infinite paths (endless loops)
  - States labeled by atomic propositions
- It is usually easy to transform other representations into KS
  - STS with first order logic prepositions in particular
- We will see how to construct KS from a program
Kripke structure – definition

- Let \( \mathcal{AP} \) be a set of *atomic propositions* (boolean expressions over variables, constants and predicate symbols)

- **Kripke structure** over \( \mathcal{AP} \) is a 4-tuple \( M = (S, I, R, L) \):
  - \( S \) is a finite set of states
  - \( I \subseteq S \) is a set of initial states
  - \( R \subseteq S \times S \) is a transition relation such that \( R \) is left-total, i.e., \( \forall s \in S \exists s' \in S \) such that \( (s, s') \in R \)
  - \( L : S \to 2^{\mathcal{AP}} \) is a labeling (or *interpretation*) function
Creating Kripke structure

\[ P = \text{cobegin} \{ P_0 \mid | \mid P_1 \} \]

\[ \text{Var}_0 = \text{Var}_1 = \{\text{turn}\} \]

\[ P_0 = \]

\begin{verbatim}
start;
while true do
  wait(turn = 0);
  turn := 1;
end;
\end{verbatim}

\[ P_1 = \]

\begin{verbatim}
start;
while true do
  wait(turn);
  turn := 0;
end;
\end{verbatim}
Creating Kripke structure

\[ P = \text{cobegin} \{ P_0 \mid \mid P_1 \} \]

\[ \text{Var}_0 = \text{Var}_1 = \{\text{turn}\} \]

\[ P_0 = \]
\[ \text{start}; \]
\[ m_0: \text{while true do} \]
\[ nc_0: \text{wait}(\text{turn} = 0); \]
\[ cr_0: \text{turn} := 1; \]
\[ \text{od}; \]
\[ n_0: \text{end}; \]

\[ P_1 = \]
\[ \text{start}; \]
\[ m_1: \text{while true do} \]
\[ nc_1: \text{wait}(\text{turn}); \]
\[ cr_1: \text{turn} := 0; \]
\[ \text{od}; \]
\[ n_1: \text{end}; \]
Creating Kripke structure for $P_0$

\[
P_0 =
\]
\begin{align*}
\text{start;} \\
m_0 &: \text{while true do} \\
\text{nc}_0 &: \text{wait}(\text{turn} = 0); \\
\text{cr}_0 &: \text{turn} := 1; \\
\text{od}; \\
n_0 &: \text{end};
\end{align*}

\textit{transition labels} are not part of KS!
$P_0 =\$

\begin{align*}
\text{start;} \\
\text{m}_0: \text{while true do} \\
\text{nc}_0: \text{wait}(\text{turn} = 0); \\
\text{cr}_0: \text{turn} := 1; \\
\text{od;} \\
\text{n}_0: \text{end};
\end{align*}

This is KS!
(propositions all atomic)
Generating state space

Model

Property specification

$G(t_0 \rightarrow F t_1)$

Property satisfied

Property violated

Error report
Generating state space

Model

Markov chains
Timed automata
Labelled transition system
Kripke structure

Model checker

Property specification

$G(t_0 \rightarrow F t_1)$

Property satisfied
Error report

Property violated
Generating state space

- State space of single KS is the KS itself
- State space of parallel composition of more KSs is Cartesian product of particular KSs
- State space is generated inside model checker
  - avoiding mistakes in manual composition
  - simplifying maintenance of input models

```
Property specification
PCTL
TCTL
CTL
Markov chains
Timed automata
Labelled transition system
Kripke structure
LTL
Model
Property satisfied
Property violated
Error report
```

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
State space for \( P_0 \parallel P_1 \)
Property specification in LTL

Property specification: $G(t_0 \rightarrow F t_1)$

Model: $m_0 \rightarrow m_1$ turn, $n_0 \rightarrow n_1$, $c_0$ turn

Model checker: Property satisfied, Property violated

Gears: PCTL, TCTL, CTL, LTL

Kripke structure: Markov chains, Timed automata, Labelled transition system

Error report
Linear Temporal Logic

- PCTL
- TCTL
- CTL
- Markov chains
- Timed automata
- Labelled transition system
- Kripke structure

Model

Property specification

\[ G(t_0 \rightarrow F t_1) \]

Model checker

- Property satisfied
- Property violated
- Error report

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
Linear Temporal Logic

- Also called **Propositional Temporal Logic**
- Able to capture many important properties
- Efficient model checking algorithm exists
  - Linear in the size of model
  - Exponential in the size of property formula
LTL Mutex properties

- Mutex properties
  - Safety: \( G \neg (c_1 \land c_2) \)
  - Liveness: \( G(t_1 \rightarrow F c_1) \)
  - Non-blocking: \( G(n_1 \rightarrow X t_1) \)
  - No strict sequencing. The following should be false at \( s_0 \):

\[
G \left( c_1 \rightarrow \left( \neg c_2 \land \left( c_1 \land \neg c_2 \ U \left( \neg c_1 \land \left( \neg c_1 \land \neg c_2 \ U \ c_2 \right) \right) \right) \right) \\
\land \\
G \left( c_2 \rightarrow \left( \neg c_1 \land \left( c_2 \land \neg c_1 \ U \left( \neg c_2 \land \left( \neg c_2 \land \neg c_1 \ U \ c_1 \right) \right) \right) \right)
\]
Syntax of LTL

- LTL formula has one of the following forms (φ and ψ are formulae):
  - 0, 1, p, ¬φ, φ ∧ ψ, φ ∨ ψ, φ ⇒ ψ
  - (for any variable p in AP)
  - Xφ neXt
  - Gφ Globally
  - Fφ Future
  - φ U ψ Until
LTL model checking – basic idea

- Given: $M$ (and a state in $M$) and $f$
  1. create $A_M$
  2. negate the property $f$
  3. create a Büchi automaton $A_{\neg f}$.
    - This automaton accepts all violations of the property $f$
  4. it holds that $M \models f$ iff $L(A_M \times A_{\neg f}) = \emptyset$
Property specification in CTL

Markov chains
Timed automata
Labelled transition system
Kripke structure

Model

Property specification in CTL

Property satisfied

Property violated

AG(t0 → EF t1)

Model checker

Error report
Computational Tree Logic

- Considers computational tree (branching)
  - Unlike LTL, which evaluates traces in isolation
- Also able to capture many important properties
- Efficient model checking algorithm exists
  - Linear in the size of model
  - Linear in the size of property formula (number of sub-formulae)
CTL syntax

A CTL formula has one of the following forms:

- 0, 1, p, \( \neg \varphi \), \( \varphi \land \psi \), \( \varphi \rightarrow \psi \), \( \varphi \lor \psi \)
  - p is an atomic formula, \( p \in AP \)
- AX \( \varphi \), EX \( \varphi \)
- AG \( \varphi \), EG \( \varphi \)
- AF \( \varphi \), EF \( \varphi \)
- A[\( \varphi \lor \psi \)], E[\( \varphi \lor \psi \)]
  - where \( \varphi, \psi \) are CTL formulae
Explicit CTL model checking algorithm

For every state $s$ in $S$, the algorithm labels $s$ with all subformulae of $\varphi$ which are true in $s$

- $\text{label}(s)$ – the set of labels associated with $s$
- initially, $\text{label}(s) = L(s)$

then, the algorithm goes through a series of stages

- during the $i$-th stage, the subformulae with $i-1$ nested operators are processed
- when a subformula is processed, it is added to the labeling of each state $s$ in which it is true (i.e. $\text{label}(s)$ is updated)

Once the algorithm terminates, we will have that

$M, s \models \varphi$ iff $\varphi \in \text{label}(s)$
CTL formula parse tree

\[(\text{EG } E[p \lor q]) \land \text{EX } r\]

\[\text{EG } E[p \lor q]\]

\[E[p \lor q]\]

\[p\]

\[q\]

\[r\]

\[\text{EX } r\]
LTL versus CTL example

\[ M, s \models_{\text{LTL}} \text{FG } p \]

not

\[ \neg M, s \models_{\text{CTL}} \text{AF (AG } p) \]

- \( \bigcirc \) = \( p \) holds
- \( \bigcirc \) = \( p \) does not hold
Explicit vs. symbolic model checking

- Explicit model checking
  - Each state of $M$ is **explicitly** represented in memory as a labeled, directed graph, and checked

- Symbolic model checking
  - Based on manipulation with **Boolean formulae**
  - The algorithm operates on entire sets of states rather than on individual states
  - Reduction of time and memory consumption
Did you know...?

Explicit model checking
- Each state of $M$ is explicitly represented as a labeled, directed graph, and checked

Symbolic model checking
- Based on manipulation with Boolean functions
- The algorithm operates on entire states rather than individual states
- Reduction of time and memory consumption

George Boole (1815 – 1864)
English mathematician, philosopher and logician
Ordered Binary Decision Diagrams

- Canonical form representation for Boolean formulae
  - Often substantially more compact than traditional normal forms (conjunctive NF, disjunctive NF)
  - Variety of applications
    - symbolic simulation
    - verification of combinational logic
    - verification of finite-state concurrent systems

- We first introduce binary decision trees
  - ... and then generalize binary decision trees to obtain (ordered) binary decision diagrams
Binary Decision Diagrams (BDDs)

- Rooted, directed acyclic graphs
- Two types of vertices
  - Nonterminal
    - Each nonterminal vertex \( v \)
      - is labeled by a variable \( \text{var}(v) \)
      - has two successors:
        - \( \text{low}(v) \) ... variable \( v \) is assigned 0
        - \( \text{high}(v) \) ... variable \( v \) is assigned 1
  - Terminal
    - Each terminal vertex \( v \) is labeled by \( \text{value}(v) \) which is either 0 or 1
OBDD Example

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
The size of an OBDD can depend critically on the variable ordering.

- For the ordering \( a1 < b1 < a2 < b2 \):
  - The OBDD has a smaller size.

- For the ordering \( a1 < a2 < b1 < b2 \):
  - The OBDD has a larger size.
Ordered Binary Decision Diagrams (OBDDs)

- For n-bit comparator
  - $a_1 < b_1 < ... < a_n < b_n$
    - 3n + 2 vertices in the OBDD
  - $a_1 < ... < a_n < b_1 < ... < b_n$
    - $3 \times 2^n - 1$ vertices in the OBDD

- In general
  - Finding an optimal ordering for variables is infeasible
    - Even checking that a particular ordering is optimal is NP-complete
  - There are many functions that have exponential size OBDDs for any variable ordering

- However: In practice, using OBDDs to encode Boolean functions, sets, Kripke structures, etc. in many cases saves time and memory
If $Q$ is an $n$-ary relation over \{0,1\}
- $Q$ can be represented by the OBDD for its characteristic function:
  $f_Q(x_1, \ldots, x_n) = 1$ iff $Q(x_1, \ldots, x_n)$

Let $Q$ be an $n$-ary relation over a finite domain $D$
- Without loss of generality we assume $D$ has $2^m$ elements for some $m > 0$
- We encode elements of $D$ using a bijection
  $\phi: \{0,1\}^m \rightarrow D$
- We construct a Boolean relation $Q_b$ of arity $m \times n$:
  $Q_b(< x_1 >, \ldots, < x_n >) = Q(\phi(< x_1 >), \ldots, \phi(< x_n >))$
  - $< x_i >$ is a vector of $m$ Boolean variables that encodes the variable $x_i$, which takes values in $D$
- $Q$ can now be represented as the OBDD determined by the characteristic function $f_{Q_b}$ of $Q_b$
Representing Kripke structures using OBDDs

- \( M = (S, R, L) \)
- Encoding \( S \)
  - We assume there are exactly \( 2^m \) states
  - \( \phi: \{0,1\}^m \rightarrow S \)
- Encoding \( R \)
  - The OBDD for characteristic function \( f_{R_b} \) of \( R_b(\langle x \rangle, \langle x' \rangle) \)
- Encoding \( L \)
  - Typically, \( L \) is defined as mapping from states to subsets of atomic propositions
  - It is more convenient to consider it as mapping from atomic propositions to subsets of states
  - An atomic proposition \( p \) is mapped to the set of states that satisfy it:
    \[ L_p = \{ s \mid p \in L(s) \} \]
  - \( L_p \) is represented using the encoding \( \phi \)
Representing Kripke structures using OBDDs

\[ R: (\neg x \land x') \lor (x \land x') \lor (x \land \neg x') \]

\[ L: a \rightarrow \{s_1, s_2\}, b \rightarrow \{s_1\} \]

\{((0,0), (0,1), (1,0))\}
Example – Symbolic CTL Model Checking

```
Example
```

```
Symbolic CTL Model Checking
```

```
Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
```

```
47
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
Symbolic Model Checking Example

- *Check the whiteboard 😊*
### Optimizations

- Markov chains
- Timed automata
- Labelled transition system
- Kripke structure

#### Model

- $m_0$
- $m_0$ turn
- $n_0$
- $n_0$ turn
- $c_0$
- $c_0$ turn

#### Property specification

$LTL\ G(t0 \rightarrow F t1)$

- Model checker
  - Property satisfied
  - Property violated

- Error report
Parallel composition of processes

- 3 processes
  - 1\textsuperscript{st} process – action \( \alpha_1 \)
  - 2\textsuperscript{nd} process – action \( \alpha_2 \)
  - 3\textsuperscript{rd} process – action \( \alpha_3 \)

- State explosion
Parallel composition of processes

- Assume the actions $\alpha_1$, $\alpha_2$, $\alpha_3$ are independent
  - e.g., updates of different variables
- Assume the “intermediate states” are unimportant with respect to the property being checked
  - i.e., all the states on the picture are labeled by the same sets of atomic propositions
Partial Order Reduction (for model checking)

- Is it really necessary to search all those paths, differing only in the order of the actions $\alpha_1$, $\alpha_2$, $\alpha_3$? **It is not!**
Partial Order Reduction

- **Idea:**
  - Before the model checking is done, *reduced state graph* is constructed
    - The full state graph is never constructed
  - The method exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders
  - Formulated by **Doron Peled** in 1993

- **The name – Partial Order Reduction**
  - Early versions of the algorithms were based on the partial order model of the program execution
  - Better name: *model checking using representatives*
Implementation in JPF

- **Java PathFinder**
  - Explicit code model checker for Java
    - Symbolic execution in latest version
  - Special virtual machine performing the verification
    - All executions with respect to thread interleaving and non-deterministic choices (random())
Implementation in JPF

- Too many possible interleavings
  - Exponential in number of threads

- Idea: switch context only when it makes sense
  - Sequential update of local variables does not affect other threads’ state
  - JPF executes instructions of current thread until the next instruction is either
    - “Scheduling relevant” instruction or
    - “Nondeterministic” instruction (random() )
Implementation in JPF

- Scheduling relevant instructions:
  - Only about 10% of instructions
  - Synchronization (monitorEnter, monitorExit, invokeX on synchronized methods)
  - Field access (putX, getX)
  - Array element access (Xaload, Xastore)
  - Thread instructions (start, sleep, yield, join)
  - Object methods (wait, notify)

- Rescheduling only if there is another runnable thread
Abstractions

- Ways to reduce (simplify) state space
  - optimization

- Eliminates some details of model
  - Cone of influence reduction
  - Data abstraction
Cone of influence reduction (CoIR)

- Focus on variables related to specification, i.e., formula to be model checked
- Variables not influencing values of variables in specification can be removed
  - As they cannot affect whether the spec is valid or not
\[ v'_0 = \neg v_0 \]
\[ v'_1 = v_0 \oplus v_1 \]
\[ v'_2 = (v_0 \land v_1) \oplus v_2 \]
Original state space

\[ v_2, v_1, v_0 \]

\[ 0,0,0 \]

\[ 0,0,1 \]

\[ 0,1,0 \]

\[ 0,1,1 \]

\[ 1,0,0 \]

\[ 1,0,1 \]

\[ 1,1,0 \]

\[ 1,1,1 \]
Theorem: Let $f$ be a CTL formula and $M$ a Kripke structure. Let $M'$ be a Kripke structure after CoIR of $M$ w.r.t. $f$. Then $M \models f \iff M' \models f$. 
Reduced state space

- $V' = \{v_0\} \rightarrow C = \{v_0\}$:

  \[ 0 \leftrightarrow 1 \]

- $V' = \{v_1\} \rightarrow C = \{v_0, v_1\}$:

  \[ 0, 0 \leftrightarrow 0, 1 \leftrightarrow 1, 0 \leftrightarrow 1, 1 \]

- $V' = \{v_2\} \rightarrow C = \{v_0, v_1, v_2\}$: original states
Data abstraction

- Number of combinations of possible values of (user) input can be enormous
- Results in very large or sometimes even infinite (floating point numbers) state space
- Model checking not possible in principle
- Solution: **Data abstraction**
Abstraction application

1. Define abstract domain(s) and map concrete values to abstract ones
2. Create reduced Kripke structure
   a) Replace concrete AP with abstract AP
   b) Merge states with same set of AP
3. Model checking
Example: traffic lights

- green
- red
- yellow
Example: traffic lights

- **AP** = \{red, yellow, green\}
  - in each state just one of them is true
  - in fact \{color = red, color = yellow, color = green\}

- **A** = \{stop, go\}

- \(h(red) = stop\)
- \(h(yellow) = stop\)
- \(h(green) = go\)
Example: reduced Kripke structure

\[ M \]

\[
\begin{array}{ccc}
\text{red} & \rightarrow & \text{green} \\
\text{yellow} & \rightarrow & \text{red}
\end{array}
\]

\[ M' \]

\[
\begin{array}{ccc}
\text{stop} & \rightarrow & \text{go} \\
\text{go} & \rightarrow & \text{stop}
\end{array}
\]

\[ M_r \]

\[
\begin{array}{ccc}
\text{go} & \rightarrow & \text{stop} \\
\text{stop} & \rightarrow & \text{go}
\end{array}
\]
Timed Automata

Model

Property specification

AG(start → AF heat)

Model checker

Property satisfied

Property violated

Error report

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
Recall: Büchi automaton and \( \omega \)-regular languages

Finite automaton accepting infinite words

A word is accepted if

- An accepting state is visited infinitely many times (standard case)
- A state from each accepting set is visited infinitely many times (generalized case)

\[
\begin{array}{c}
\text{Büchi automaton accepting } (a+b)^*a^\omega \\
\end{array}
\]
Timed languages

**Timed sequence** $t = t_1t_2t_3...$ is an infinite sequence of time values $t_i \in \mathbb{R}$, $t_i > 0$ satisfying:

1. Monotonicity, i.e., $\forall i \geq 1: t_i < t_{i+1}$
2. Progress, i.e., $\forall t \in \mathbb{R}, \exists i \geq 1: t_i > t$

**Timed word** is a tuple $(s,t)$, where

- $s$ is an infinite sequence of symbols
- $t$ is a timed sequence (above)
Timed automaton – example

In addition to Büchi, finite set of real variables representing clocks (below: \( x \))

- Initially set to 0, all incrementing at the same speed
- Can be reset to 0 at any transition
- Transition only allowed if the condition upon clocks holds
- Accepts timed words

Example of Timed automaton
For a set $X$ of clocks, the set $\Phi(X)$ of clock constraints $\delta$ is defined:

$$\delta := x \leq c \mid c \leq x \mid \neg \delta \mid \delta_1 \land \delta_2,$$

where $x$ is a clock in $X$ and $c$ is a constant in $\mathbb{Q}$.
The automaton below accepts the language:

\[ L = \{(abcd)^\omega, t) \mid \forall j.((t_{4j+3} < t_{4j+1}+1) \wedge (t_{4j+4} > t_{4j+2}+2))\} \]
Checking for emptiness

Important property

- Recall LTL model checking algorithm

**Idea:** Construct Büchi $B$ automaton such that $B$ accepts the same language (up to timing) as the timed automaton under consideration
• Induce infinite state space due to real clocks
• Need for a conversion before analysis (e.g., model checking)
  \[\rightarrow\] region automaton
Region automaton – example
Lemma: If \( r \) is a progressive run of \( R(A) \) over \( s \), then there exists a time sequence \( t \) and a run \( r' \) of \( A \) over \( (s,t) \) such that \( r \) equals \([r']\).

- Progressive means that for all clocks there is no bound
- We can consider just progressive runs
  - Proof skipped 😊
Theorem: Given Timed automaton $A = (\Sigma, S, S_0, \Delta, F)$, there exists Büchi automaton which accepts $Untime(L(A))$.

Idea:
1. Construct region automaton $R(A)$
2. Set of accepting states $F' = \{(s,a) \mid s \in F\}$
3. Omit time
Network of TA

The diagram illustrates a network of TA (Task-Activity) for controlling a lamp. The states and transitions are as follows:

- **Off**
  - Transition on press: *y := 0*
  - Transition on *y >= 5*: press

- **Low**
  - Transition on press: *y < 5*

- **Bright**
  - Transition on press: *y >= 5*

- **Idle**
  - Transition on press: *y := 0*

The user can press the lamp to change its state, and the lamp responds based on the current state and the user's action.
• A tool for verification of TA models
• Academic, but quite well established and used in industry nowadays
• Allows modeling, verification, simulation
• Successfully applied on communication protocols, multimedia applications, ...
• Available at http://www.uppaal.org/ and http://www.uppaal.com
Stochastic Model Checking

- Markov chains
- Timed automata
- Labelled transition system
- Kripke structure

Model

- open
- close
- heat
- start
- close
- heat
- empty

Property specification

\[ AG(\text{start} \rightarrow \text{AF heat}) \]

Model checker

- Property satisfied
- Property violated
- Error report

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
Stochastic model checking – Motivation

- In some cases **absolute** absence of errors is infeasible
  - failures of particular parts of system
  - non-deterministic behavior of users
  - ...

- It might be useful to determine level of reliability in terms of probability
  - frequency of errors
  - time to recovery
  - throughput
  - mean waiting time
  - ...

Jan Kofroň: Tutorial on formal methods, D3S seminar, 2016/2017
The answer is:

Probabilistic (stochastic) model checking

*Lecture based on M. Kwiatkowska et al.: Stochastic Model Checking*

Stochastic model checking

• Not only validity of certain properties
  ▪ but also probability of reaching states/paths

• → Need for special language
  ▪ PCTL = Probabilistic Computational Tree Logic
  ▪ CSL = Continuous Stochastic Logic

• For discrete time analysis, Discrete-time Markov Chains (DTMC) are used as models

• For continuous time analysis, Continuous-time Markov Chains (CTMC) are used
**Definition:** A labeled DTMC $D$ is a tuple $(S, \bar{s}, P, L)$ where:

- $S$ is finite set of states
- $\bar{s} \in S$ is initial state
- $P: S \times S \rightarrow [0,1]$ is **transition probability matrix** where $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$
- $L: S \rightarrow 2^{AP}$ is labeling function assigning to each state set $L(s)$ of atomic propositions
Discrete-time Markov chains

- Sum of probabilities of transitions originating in each state must be 1!
- Terminating states can be modeled by self-loop with probability 1
Example

\[ P = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0.01 & 0.01 & 0.98 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
\end{pmatrix} \]
Probabilistic Computational Tree Logic (PCTL)

- Extension of CTL
- Syntax:
  \[
  \Phi ::= true \mid a \mid \neg \Phi \mid \Phi \land \Phi \mid P_{\sim p}[\phi] \quad \text{(state formula)}
  
  \phi ::= X\Phi \mid \Phi \mathrel{U}^{\leq k} \Phi \quad \text{, where} \quad \text{(path formula)}
  \]

  \(a\) is atomic proposition
  \(\sim \in \{<, \leq, \geq, >\}\)
  \(p \in [0,1]\)
  \(k \in \mathbb{N} \cup \infty\)

- ... plus common (derived) facts:
  - \(false \equiv \neg true\)
  - \(\Phi \lor \Psi \equiv \neg(\neg \Phi \land \neg \Psi)\)
Semantics of PCTL

\[ s \models true \quad \text{for all } s \in S \]
\[ s \models a \iff a \in L(s) \]
\[ s \models \neg \Phi \iff s \not\models \Phi \]
\[ s \models \Phi \land \Psi \iff s \models \Phi \land s \models \Psi \]
\[ s \models P_{\neg p} [\phi] \iff Prob^D (s, \phi) \sim p \]
\[ \omega \models X\Phi \iff \omega(1) \models \Phi \]
\[ \omega \models \phi \sqcap \leq k \psi \iff \exists i \in \mathbb{N} : (i \leq k \land \omega(i) \models \psi \land \forall j < i : (\omega(j) \models \phi)) \]

where \( Prob^D (s, \phi) \overset{\text{def}}{=} Pr_s \{ \omega \in Path^D (s) \mid \omega \models \phi \} \)
Common CTL operators

- CTL $F$ and $G$ operators:

\[
P_{\neg p}[F \Phi] \equiv P_{\neg p}[true \ U^{\leq \infty} \Phi]
\]

\[
P_{\neg p}[F^{\leq k} \Phi] \equiv P_{\neg p}[true \ U^{\leq k} \Phi]
\]

\[
G \Phi \equiv \neg F \neg \Phi
\]

\[
G^{\leq k} \Phi \equiv \neg F^{\leq k} \neg \Phi
\]
Negation

- Syntax does not allow for negation of path formulae
- However, it holds:

\[ P_{\sim p}[G \Phi] \equiv P_{\sim 1-p}[F \neg \Phi] \]
\[ P_{\sim p}[G \leq k \Phi] \equiv P_{\sim 1-p}[F \leq k \neg \Phi] \]

where \( \bar{<} \equiv >, \bar{\leq} \equiv \geq, \bar{\geq} \equiv \leq, \bar{>} \equiv < \)
Quantifiers

- $P_{\sim p} [\cdot]$ is probabilistic analogue to path quantifiers:
  - $EF \Phi \equiv P_{>0} [F \Phi]$
  - But: $AF \Phi$ is NOT the same as $P_{\geq 1} [F \Phi]$

\[
\begin{align*}
\text{c}_0 & \text{satisfies } P_{\geq 1} [F \text{ tails}] \\
\text{c}_0 & \text{does NOT satisfy } AF \text{ tails}
\end{align*}
\]
Examples of PCTL properties

- $P_{\geq 0.4}[X \text{ delivered}]$
  - probability that message gets delivered in next step is at least 0.4

- $init \rightarrow P_{\leq 0}[F \text{ error}]$
  - error state is not reachable from any init state

- $P_{\geq 0.9}[\neg \text{down } U \text{ served}]$
  - probability that server does not go down before request gets served is at least 0.9

- $P_{< 0.1}[\neg \text{done } U_{\leq 10} \text{ fault}]$
  - probability that error occurs before protocol is done and within 10 steps is less than 0.1
\( P_{\geq 0.9}[X(\neg \text{try} \lor \text{succ})] \)
$\chi \Phi$ – Example

$Sat(\neg \text{try} \lor \text{succ}) = \{s_0, s_2, s_3\} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$

$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.01 & 0.01 & 0.98 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
\( \Xi \Phi - Example \)

\[ P = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0.01 & 0.01 & 0.98 & 0 \\
1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
\end{pmatrix} \]

\[
\begin{pmatrix}
0 \\
0 \\
1 \\
0 \\
0
\end{pmatrix}
\]

\[
\begin{pmatrix}
0 \\
0.99 \\
1 \\
1 \\
\end{pmatrix}
\]
\[ P_{>0.98}[F_{\leq 2} \text{succ}] = P_{>0.98}[\text{true } U_{\leq 2} \text{succ}] \]
\( \phi \cup^k \psi \) – for \( k \neq \infty \) – Example

\[ P = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0.01 & 0.01 & 0.98 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix} \]

\[
Sat(\text{true}) = \{s_0, s_1, s_2, s_3\}, \quad Sat(\text{succ}) = \{s_3\}
\]

\[
P[\neg \text{true} \lor \text{succ}] = P
\]
\[ \Phi \ U \leq_k \Psi \ - \ for \ k \neq \infty \ - \ Example \]

\[ \text{Prob}^D (\Phi \ U \leq 0 \Psi) = \text{succ} = [0,0,0,1] \]
\[ \text{Prob}^D (\Phi \ U \leq 1 \Psi) = P[\neg \text{true} \lor \text{succ}] \cdot \text{Prob}^D (\Phi \ U \leq 0 \Psi) = [0,0.98,0,1] \]
\[ \text{Prob}^D (\Phi \ U \leq 2 \Psi) = P[\neg \text{true} \lor \text{succ}] \cdot \text{Prob}^D (\Phi \ U \leq 1 \Psi) = [0.98,0.9898,0,1] \]

\[ P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.01 & 0.01 & 0.98 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]
\[ \Phi \leq^k \Psi - \text{for } k \neq \infty - \text{Example} \]

\[
\begin{align*}
\Pr_D^D (\Phi \leq^2 \Psi) &= [0.98, 0.9898, 0, 1] \\
\text{Hence } \text{Sat}(P_{>0.98}[F_{\leq^2} \text{succ}]) &= \{s_1, s_3\}
\end{align*}
\]
$\Phi \ U^{\leq k} \Psi$ – for $k = \infty$ – Example

$P_{>0.99}[\text{try U succ}]$

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.01 & 0.01 & 0.98 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$\Phi \cup \leq^k \Psi – \text{for } k = \infty – \text{Example}$

$Sat(\text{try}) = \{s_1\}, \quad Sat(\text{succ}) = \{s_3\}$
\( \Phi \cup U^{\leq k} \Psi - \text{for } k = \infty \) - Example

\[
P = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0.01 & 0.01 & 0.98 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
Sat(P_{\leq 0}[\text{try } U \text{ succ}]) = \{s_0, s_2\}, \quad Sat(P_{\geq 1}[\text{try } U \text{ succ}]) = \{s_3\}
\]
\( \Phi \ U^{\leq k} \ \Psi \ - \ for \ k = \infty - \ Example \)

\[
\begin{align*}
P &= \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0.01 & 0.01 & 0.98 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\end{align*}
\]

\[
\begin{align*}
Prob^D(s_0, \text{try } U \text{ succ}) &= 0 \\
Prob^D(s_1, \text{try } U \text{ succ}) &= 0.01 \cdot Prob^D(s_1, \text{try } U \text{ succ}) + 0.01 \cdot Prob^D(s_2, \text{try } U \text{ succ}) + 0.98 \cdot Prob^D(s_3, \text{try } U \text{ succ}) \\
Prob^D(s_2, \text{try } U \text{ succ}) &= 0 \\
Prob^D(s_3, \text{try } U \text{ succ}) &= 1
\end{align*}
\]
\( \Phi \ U \leq_k \Psi \ - \ for \ k = \infty - \ Example \)

\[ P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0.01 & 0.01 & 0.98 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

\[ \text{Prob}^D(\text{try } U \text{ succ}) = (0, \frac{98}{99}, 0, 1) \]
\( \Phi \ U \leq k \ \Psi - for \ k = \infty - Example \)

\[
P = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0.01 & 0.01 & 0.98 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
Prob^D(\text{try } U \text{ succ}) = (0, \frac{98}{99}, 0, 1)
\]

\( P_{>0.99}[\text{try } U \text{ succ}] \) is satisfied in \( s_3 \)
DTMC and PCTL can be extended by rewards (or costs)

- specification of cost for transition
- reasoning about cost of particular computation, e.g., satisfying PCTL property, restricting to computations with cost less than $k$, ...
PRISM – Probabilistic Model Checker

- Allows for checking DTMC, CTMC and other types of models
- Uses simple dedicated input language
- http://www.prismmodelchecker.org/

```plaintext
// two process mutual exclusion
mdp
module M1
  x : [0..5] init 0;
  [] x=0 -> 0.8:(x'=0) + 0.2:(x'=1);
  [] x=1 & y=2 -> (x'=2);
  [] x=0 & y=1 -> 0.5:(x'=2) + 0.5:(x'=0);
endmodule
module M2
  y : [0..5] init 0;
  [] y=0 -> 0.8:(y'=0) + 0.2:(y'=1);
  [] y=1 & x=2 -> (y'=2);
  [] y=2 & x=1 -> 0.5:(y'=2) + 0.5:(y'=0);
endmodule
```