Behavior models and verification

Lecture 1

Jan Kofroň, František Plášil
Sylabus I.

• Mathematical structures for behavior modeling:
  - labeled transition systems, Kripke structures, Timed automata, Markov chains

• Specification of system properties
  - temporal logics: LTL, CTL, TCTL, PCTL,

• Basic verification tasks
  - equivalence checking and model checking
Decidability and complexity
- of equivalence checking and model checking in dependence of the type of the model

Software tools
- for model checking

Hard issues in formal verification
- infinite-state systems
- state explosion problem
  - Strategies to fight it
Grading

• Final grades will be determined by the quality of homework and the result of the final exam in the following ratio:
  - 55% Assignments (two homeworks)
  - 45% Final exam
    - >= 80 of 100 ~ 1
    - >= 71 ~ 2
    - >= 62 ~ 3

• Final exam
  - Written test
Available at the course web page:

http://d3s.mff.cuni.cz/teaching/nswi101/
Administrative Information

Time and Location: Winter Semester 2016/2017
Lectures: Wed 14:00 S1
Lab: Thu 15:40 S10

Guaranteed by: Department of Distributed and Dependable Systems

Winter Term: 2/2 Zk+Z

Lecturers:
- František Plášil
  e-mail: frantisek.plasil@d3s.mff.cuni.cz
- Jan Kofroň
  e-mail: jan.kofron@d3s.mff.cuni.cz

Lab:
- Jan Kofroň
  e-mail for HW: nswi101@d3s.mff.cuni.cz

Information in SIS: NSWI101

News

Slides

Lectures

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Slides</th>
</tr>
</thead>
</table>

Lab

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Slides</th>
</tr>
</thead>
</table>

Annotation

Basic concepts of behavior description of parallel and distributed systems. Equivalence checking and model checking — techniques and tools.
Need: Striking stories

- Ariane 5, 1996
  - False angle of attack
    - caused by incorrect altitude data following a software exception and the rocket self-destructed in 37 seconds after launch
  - The software exception
    - an overflow in the conversion of a 64-bit floating-point number to a 16-bit signed integer value
    - => an operand error.
  - Direct cost 500.000.000 EUR,
    - indirect cost 2.000.000.000 EUR
• The operand error occurred because of an unexpected high value of the horizontal velocity sensed by the platform.

• The value was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in higher horizontal velocity values.
Need: Striking stories

- Intel: Pentium FDIV bug, 1994
  - Cost $500 000 000
Need: Striking stories

- Mars Climate Orbiter
  - NASA: SEPTEMBER 30, 1999
    The peer review preliminary findings indicate that one team used English units while the other used metric units for a key spacecraft operation

- Mars Polar Lander Dec. 1999
  - The leading theory is that a surface contact detector located on the landing struts mistakenly interpreted the force of the landing strut's deployment as contact with the surface, causing the landing rockets to shut down prematurely and the probe to impact at too high a velocity.
Need: Striking stories

- Shutdown of USS Yorktown (CG-48), 1997
  - A sailor mistakenly typed 0 in a field of the kitchen inventory application
  - Subsequent division by this field caused an arithmetic exception, which propagated through the system, led to power shutdown for about 3 hours
Need: Striking stories

- Sensor failure caused not detecting a human in the seat. In turn, the airbag malfunction (failing in a car crash) appeared.

- In 2015, Nissan recalled 3.5 millions of cars to fix this

- Airbag problems reported by other car manufacturers in 2016
  - General Motors – GMC, Chevrolet, Buick, Cadillac
A nice collection available at

http://www5.in.tum.de/persons/huckle/bugse.html
Context: HW and SW reliability

- Complex (mostly reactive) systems
  - HW circuits (processors, ...)
  - Communication protocols
  - Traffic control (traffic lights on intersections)
  - Air-traffic control
  - Mission-critical
  - Cyber Physical Systems (IoT,...)

- SW code in general
  - Drivers, OS parts...
Achieving system reliability

- Experimental methods
  - Testing
    - (applied to the system itself)
  - Simulation
    - (experimenting with a model of a system)

- Formal methods
  - Deductive verification
    - Theorem proving
  - Equivalence checking
    - Comparing two specifications (models)
  - Model checking
    - Checking a particular property of a model (even code)
**Model Checking**

**Kripke structure**

Model

Property specification

$AG(start \rightarrow AF heat)$

Model checker

Property satisfied

Property violated

Error report

---

Jan Kofroň, František Plášil, Lecture 1
What makes model-checking software difficult?

Model

Kripke structure

Model checker

Property specification

CTL

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

Property satisfied

Property violated

Error report
What makes model-checking software difficult?

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

```java
void add(Object o) {
    buffer[head] = o;
    head = (head+1)%size;
}

Object take() {
    ...
    tail=(tail+1)%size;
    return buffer[tail];
}
```

Program

Model

Semantic gap:

Programming Languages

*methods, inheritance, dynamic creation, exceptions, etc.*

Model Description Languages

*States and transitions*
What makes model-checking software difficult?

Problems using existing checkers:

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

Difficult to formalize a requirement in temporal logic

“Between the window open and the window close, button X can be pushed at most twice.”

...is rendered in LTL as...

\[
[]((\text{open} \land \lnot\text{close}) \rightarrow \\
((\lnot\text{pushX} \land \lnot\text{close}) \lor \\
(\text{close} \lor ((\text{pushX} \land \lnot\text{close}) \lor \\
(\text{close} \lor ((\lnot\text{pushX} \land \lnot\text{close}) \lor \\
(\text{close} \lor ((\lnot\text{pushX} \lor \text{close})))))))))
\]
Forced to state property in terms of *model* rather than *source*

We want to write *source level* specifications...

**Heap.b.head == Heap.b.tail**

We are forced to write *model level* specifications...

```plaintext
(((_collect(heap_b) == 1)\
  && (BoundedBuffer_col_1.instance[_index(heap_b)].head ==
       BoundedBuffer_col_1.instance[_index(heap_b)].tail)) )\n|| ((_collect(heap_b) == 3)\
  && (BoundedBuffer_col_0.instance[_index(heap_b)].head ==
       BoundedBuffer_col_0.instance[_index(heap_b)].tail)) )\n|| ((_collect(heap_b) == 0) && TRAP))
```
What makes model-checking software difficult?

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

AG(start → AF heat)

CTL
State space

- Model induces **state space**
  - combination of states of particular parts (behavior of involved processes)
  - potentially exponential in size of model

- **State space explosion** = Problem of too many states induced by the model

- Moore’s law and algorithm advances can help

- **BUT**: Explosive state growth in software inherently limits scalability
What makes model-checking software difficult?

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

Raw error trace may be 1000’s of steps long

Must map lines of source program onto model description

Mapping to source is made difficult by

Semantic gap & clever encodings of complex features
multiple optimizations and transformations

Over-approximations in abstractions may yield infeasible error traces (how to decide if feasible or not?)
\begin{align*}
x &:= 0; \\
y &:= 0; \\
\text{for } i &:= 1 \text{ to } 3 \text{ do} \\
(x &:= x + 1; \ y := y + 1) \\
(\quad &x := x + 1 \\
\quad &y := 0 \\
\quad &x := 0 \\
\quad &y := y + 1)
\end{align*}
- **Labeled Transition System**
  - Is a triple \((S, \text{Act}, \rightarrow)\)
  - \(S\) set of *states* (domain)
  - \(\text{Act}\) set of *labels* (actions)
  - \(\rightarrow\) *transition relation* \(\rightarrow \subseteq S \times \text{Act} \times S\)
LTS Reminds Finite Automaton

- **LTS**
  - Is a triple \((S, \text{Act}, \rightarrow)\)
  - **Trace**
    - string(sequence) of labels following a path in an LTS
    - \text{Act} = L, typically enhanced by \(\tau\) (internal action)

- **Finite Automaton**
  - \((S, \text{Act}, \rightarrow, \text{start\_state}, \text{Accepting\_States})\)
  - word
Sequences of observable actions:

$$\text{traces}(s) = \{ \sigma \in L^* \mid s \xrightarrow{\sigma} \}$$

$$\text{traces}(s_0) = \{ \varepsilon, \text{coin}, \text{coin.coffee}, \text{coin.tea} \}$$
Trace preorder

- Preorder: a relation
  - reflexive
  - transitive
  - *(not required!)*: antisymmetric, i.e. \( a \leq b \land \neg(a=b) \Rightarrow \neg(b \leq a) \)
Trace preorder

\[ i \leq_{tr} s = \text{traces}(i) \subseteq \text{traces}(s) \]
Trace preorder

\[ i \leq_{tr} s = \text{traces}(i) \subseteq \text{traces}(s) \]
Trace preorder

\[ i \leq_{str} s = \text{traces}(i) \subseteq \text{traces}(s) \]
Trace preorder

\[ i \leq_{tr} s = \text{traces}(i) \subseteq \text{traces}(s) \]
Trace preorder

\[ i \leq_{tr} s = \text{traces}(i) \subseteq \text{traces}(s) \]
**Trace preorder**

\[ i \leq_{\text{str}} s = \text{traces}(i) \subseteq \text{traces}(s) \]
LTS semantics

- Semantics ~ 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: Behavior equivalence – trace eq.

- Trace based (behavior) equivalence
  - trace preorder determines trace equivalence
    - traces(s) = \{w \in \text{Act}^* \mid s \xrightarrow{w} s' \text{ for some } s' \in S\}
    - $s \leq_{t} t$ iff $\text{traces}(s) \subseteq \text{traces}(t)$
  - Trace equivalence $=_{t}$
    - $s =_{t} t$ iff $s \leq_{t} t$ & $t \leq_{t} s$
**LTS: Behavior equivalence – trace eq.**

- **Trace equivalence** \( =_t \)
  - \( s =_t t \) iff \( s \leq_t t \) & \( t \leq_t s \)
  - Notice nondeterminism
  - \( s, t, u \in S \) of an \((S, \text{Act}, \rightarrow)\)

- Informally: \( =_t \) does not reflect the structural differences in the LTS graphs

---

![Diagram showing trace equivalence]

The diagram illustrates the concept of trace equivalence using three LTS graphs. Each graph represents a configuration with labeled states and transitions. The traces from the initial state \( s \) to the final state \( t \) are shown, with \( =t \) indicating the equivalence of traces. The example demonstrates how \( =_t \) captures the structural equivalence of processes, despite the presence of nondeterminism, which \( =_t \) does not reflect.
LTS: Behavior equivalence – trace eq.

- Trace equivalence ver. Language equivalence
  - Finite Automaton
    - $(S, \text{Act}, \rightarrow, \text{init}_\text{state}, \text{Accepting}_\text{States})$ \textit{Word}
  - LTS
    - $(S, \text{Act}, \rightarrow)$ \textit{Trace}
  - \textit{What makes a difference}
    - A prefix of a trace $w$ is also a valid trace
    - $w$ can be infinite!
  - LTS $\rightarrow$ Automaton
    - (i) Make all the states accepting, choose an $\text{init}_\text{state}$
    - (ii) Büchi automaton ($\omega$ regular language)
      - Coping with infinite traces
  - Automaton $\rightarrow$ LTS
    - Introduce a ‘sentinel’ (“end of trace” label)
    - Consider only the traces ending with sentinel
Comments on LTS traces and languages

- LTS with finite # of states
  - plus init state and final states → finite automaton, depending upon the semantics of accepting a trace:
    - regular language
      - finite traces
    - omega regular language
      - infinite traces

- LTS with infinite # of states
  - type 0 language in general
    - Turing machine acceptance
Simulation preorder and equivalence

- **simulation determines simulation equivalence**
  - A relation \( R \subseteq S \times S \) is a *simulation* iff whenever \( (s, t) \in R \)
    then for each \( s \xrightarrow{a} s' \) there is \( t \xrightarrow{a} t' \) such that \( (s', t') \in R \)
  - Simulation preorder \( s \leq_S t \) iff
    there is (exists) a simulation \( R \) such that \( (s, t) \in R \)
  - **Simulation equivalence** \( s =_S t \) iff \( s \leq_S t \) \& \( t \leq_S s \)
LTS: Behavior equivalence – bisimilarity

• **Strong Bisimilarity**
  
  • A relation $R \subseteq S \times S$ is a *bisimulation* iff whenever $(s,t) \in R$
  
  then
  
  - for each $s \xrightarrow{a} s'$ there is $t \xrightarrow{a} t'$ such that $(s',t') \in R$
  - for each $t \xrightarrow{a} t'$ there is $s \xrightarrow{a} s'$ such that $(s',t') \in R$

  • *(Strong) bisimilarity equivalence* $s \sim t$ iff there is (exists) a bisimulation $R$ such that $(s,t) \in R$
• Strong Bisimilarity
  - \( \neg (s \sim u) \)
  - \( \neg (u \sim t) \)
  - \( \neg (t \sim s) \)
LTS: Behavior equivalence – bisimilarity

- Strong Bisimilarity

\[
x \sim y
\]
Influence of determinism

- If an $T = (S, \text{Act}, \rightarrow)$ is deterministic, then
  $\sim_s = \sim_t$ are equal
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
Process Algebras

- ACP style Process Algebra
  - Focus on algebraic approach
    - Organized in different equational theories
  - Process theory
    - Minimal
    - with **sequential** processes
    - with **recursive** processes
    - with **parallel** processes
    - with **abstraction**
    - with **probability**
  - Different semantical models
    - (there is a standard one: \( \sim \) i.e. bisimilarity)
Example

- $A = a.b.\varepsilon$ and $B = b.c.\delta$
  - $\varepsilon$ - successful termination
  - $\delta$ - deadlock – null process
  - $a$, $b$, $c$ - primitive actions (processes)
  - $.$ concatenation $\sim$ sequence of processes

- $A \parallel B$
  - Parallel execution
State space of $A \parallel B$

$A = a.b.\varepsilon$ and $B = b.c.\delta$
Features:

- **Parallelism**
  - $M \parallel (a.b.\varepsilon + b.c.\varepsilon)$
  - Actions are atomic (interleaving ... )!

- **Communication (synchronization)**
  - $\gamma(b, c) = d$
  - particular pair of actions becomes another action

- **Abstraction**
  - $\tau_{\{i_1, i_2\}}(a.i_{1}.i_{2}.b.\varepsilon) = a.\tau.b.\varepsilon$
  - listed actions become internal

The resulting theory

- **Algebra of Communicating Processes (ACP)**
ACP axioms

- Let $x$, $y$ be variables
  - $x + y = y + x$  \hspace{1cm} A1 \hspace{1cm} commut. +
  - $(x + y) + z = x + (y + z)$  \hspace{1cm} A2 \hspace{1cm} assoc. +
  - $x + x = x$  \hspace{1cm} A3 \hspace{1cm} idempot. +
  - $(x + y). z = x.z + y.z$  \hspace{1cm} A4 \hspace{1cm} right distr. + and .
  - $(x .y) . z = x .(y . z)$  \hspace{1cm} A5 \hspace{1cm} assoc. .
  - $x + \delta = x$  \hspace{1cm} A6 \hspace{1cm} neutral element
  - $\delta. x = \delta$  \hspace{1cm} A7 \hspace{1cm} null process

- Note
  - $z. (x + y) = z.x + z.y$ is not included (nondeterministic choice)!

Jan Kofroň, František Plášil, Lecture 1
• If A and B are processes which cannot communicate, then
  
  the parallel composition A || B executes A and B arbitrarily interleaved
  
  • For example:
    - A = a.b.ε and B = b.c.δ,
      then A || B can execute a, b, b, c, or b, a, c, b, or a, b, c, b, etc.

• The || operator can be eliminated:
  
  A || B = a.(b.b.c.δ + b.(b.c.δ + c.b.δ)) + b.(a.(b.c.δ + c.b.δ) + c.a.b.δ)
State space of $A \ || \ B$

\[ A = a.b.\varepsilon \text{ and } B = b.c.\delta \]
A and B as before, but now they communicate on action b

- The result of a communication is action d:
  - $\gamma(b, b) = d$
  - formally A and B together execute action d

Again, the $\mid \mid$ operator can be eliminated:

- $A \mid \mid B = a.(b.b.c.\delta + b.(b.c.\delta + c.b.\delta) + d.c.\delta) + b.(a.(b.c.\delta + c.b.\delta) + c.a.b.\delta)$
State space of $A \parallel B$ with communication

Assuming $\gamma(b, b) = d$
Enforcing communication

- By encapsulating (disabling) certain actions, communication can be enforced.
  - New process operator: $\partial_H(\_), \text{ with } H \subseteq A$.
  - Process $\partial_H(x)$ is like $x$, but cannot execute actions $a \in H$.

- For example,
  - $\partial_{\{b\}}(A \mid | B) = a.d.c.\delta$
  - The $b$ actions of $A | | B$ are encapsulated/disabled
Building concurrent systems

- Specify separate components
- Specify the communication actions between these components
- Construct parallel compositions of components
- Encapsulate certain actions to enforce communication
References