Introduction & Formal Methods

Jan Kofroň

http://d3s.mff.cuni.cz
Introduction to dependable systems – NSWE 002

What you learn:

- Dependable systems
- Formal methods
- Model checking, code analysis
- Middleware/Cloud computing
- Realtime scheduling & design
- Modelling and performance measurements
- Metamodeling and model-driven development
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
  - 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

- Therac-25
  - Computer controller radiation therapy machine
  - 6 accidents 1985-1987
    - three people died as the direct consequence of radiation burns
  - Race condition as the primary cause
  - Other causes included
    - Poor design, no review of the software
    - Bad man-machine interface
    - Overconfidence in the software
    - Not understanding safety
    - The software was in use previously, but different hardware design covered its flaws
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
Means to make systems secure and dependable

• Design, e.g., design by models
  ▪ Model -> Code
  ▪ Can be (partially) automatized

• Performance monitoring and prediction

• System abstractions
  ▪ System-resource level
  ▪ Application level

• Formal methods
  ▪ Validation and verification of
    • Models – Model checking
    • Code – Static analysis, code (model) checking
Motivation for using formal methods

- Sometimes (software) system needs to be correct
  - avionics
  - medicine
  - weapons

- If its complexity is high, direct code analysis is not possible
  - techniques and approaches do not scale limitlessly
For complex system, create its model and analyze it, instead of system itself.
• What is model of system and how it should look like?
  ▪ structure resembling original system
  ▪ captures important aspects of system
  ▪ is well-formalized
  ▪ uses few abstractions
  ▪ is simpler than original system
  ▪ can be analyzed by tools
Model of system II.

- Usually takes the form of **finite** graph
  - vertices with transitions
  - Kripke structure
  - labeled transition system
  - (finite) (deterministic) automaton
  - ...

Jan Kofroň: Introduction & Formal Methods
x:=0; y:=0;
for i:=1 to 3 do
  (x:=x+1; y:=y+1)
Model analysis

- Model is analyzed
  - it is **verified** that it satisfies some properties
  - this process is called **Model Checking**

- Model checking: “Given a model of a system, test automatically whether this model meets a given specification”
Properties can be either
- implicit, e.g., absence of deadlock, termination
- explicit, e.g., eventual response to an action, reaching a specific state

Usually expressed in **temporal logic**
- Linear time logic (LTL)
- Branching temporal logic (CTL)

“For all executions it holds that there is a state when the systems is ready to accept requests”

“For all executions it holds that from each state a terminal state can be reached”
Model Checking

Kripke structure

Model

Property specification

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

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

\( \text{CTL} \)

AG(start \( \rightarrow \) AF heat)

Property satisfied

Property violated

Error report

Jan Kofroň, František Plášil, Lecture 1
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];
}
```

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
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...

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

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

\[
\text{Heap.b.head} \equiv \text{Heap.b.tail}
\]

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

\[
((\_\text{collect}(\text{heap}_b) == 1) \\&\& \text{BoundedBuffer\_col.}\text{instance[\_index(\text{heap}_b)].head} == \text{BoundedBuffer\_col.}\text{instance[\_index(\text{heap}_b)].tail}) \\| \ ((\_\text{collect}(\text{heap}_b) == 3) \\&\& \text{BoundedBuffer\_col\_0.}\text{instance[\_index(\text{heap}_b)].head} == \text{BoundedBuffer\_col\_0.}\text{instance[\_index(\text{heap}_b)].tail}) \\| \ ((\_\text{collect}(\text{heap}_b) == 0) \&\& \text{TRAP}))
\]
What makes model-checking software difficult?

Problems using existing checkers:

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

CTL

Property specification

\[ AG(start \to AF heat) \]
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?)
• Check whether a system (e.g. program) satisfies a given property
  - The property specified as a formula in a predicate logic
  - The behavior specified in a functional programming language or as a logic formula

\[ \forall x: [a] \ (\text{length}(x) = \text{length}(\text{reverse}(x))) \]

length [] = 0
length x::xs = 1 + length(xs)
reverse [] = []
reverse x::xs = conc(reverse(xs), [x])
...

...
Theorem Proving: Characteristics

- Exhaustive
  - All errors are found and no false negatives are generated
- It is undecidable in general
  - Human assistance needed to finish a proof
- Not very suitable for proving properties of imperative programs
  - Hard to capture global state and side effects
Proof-carrying code

- For a given program, a formula describing its correctness is generated
  - with respect to a given set of security policies
- The formula is proven
  - either automatically or with human assistance
- The proof is distributed along with the code
  - as a security certificate
- The receiver of the code
  - Generates the formula and checks whether the proof is proving the formula
Modeling System

- Model can be
  - constructed by hand
  - if available “reversed-engineered” from code
    - source code, byte code, machine code
    - not always possible depending on properties we want to check
Ways to represent the model
- drawing LTS, Kripke structure or automaton not feasible for real systems

Specialized formalisms
- textual – process algebras, behavior specification languages, ...
- graphical – UML, timed automata, ...
Means to fight state space explosion
- partition system into pieces verified separately
- provide overview of the system
- help express the communication and properties more precisely
Study programs

- Bakalářský obor
  - Softwarové systémy

- Navazující magisterský plán Softwarové systémy
  - Spolehlivé systémy (Dependable systems)

- Doktorský program
  - 4I2 – Softwarové systémy