Formal Verification, Model Checking

Radek Pelánek
Formal Methods: Motivation

- examples of what can go wrong – first lecture
- non-intuitiveness of concurrency (particularly with shared resources)
  - mutual exclusion
  - adding puzzle
Formal Methods

‘Formal Methods’ refers to mathematically **rigorous** techniques and tools for

- specification
- design
- verification

of software and hardware systems.
Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property.
## Formal Verification vs Testing

<table>
<thead>
<tr>
<th></th>
<th>Formal Verification</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>finding bugs</td>
<td>medium</td>
<td>good</td>
</tr>
<tr>
<td>proving correctness</td>
<td>good</td>
<td>-</td>
</tr>
<tr>
<td>cost</td>
<td>high</td>
<td>small</td>
</tr>
</tbody>
</table>
# Types of Bugs

<table>
<thead>
<tr>
<th></th>
<th>likely</th>
<th>rare</th>
</tr>
</thead>
<tbody>
<tr>
<td>harmless</td>
<td>testing</td>
<td>not important</td>
</tr>
<tr>
<td>catastrophic</td>
<td>testing, FV</td>
<td>FV</td>
</tr>
</tbody>
</table>
Formal Verification Techniques

- **manual** human tries to produce a proof of correctness
- **semi-automatic** theorem proving
- **automatic** algorithm takes a model (program) and a property; decides whether the model satisfies the property

We focus on automatic techniques.
Motivation

Application Domains of FV

- generally **safety-critical systems**: a system whose failure can cause death, injury, or big financial loses (e.g., aircraft, nuclear station)
- particularly **embedded systems**
  - often safety critical
  - reasonably small and thus amenable to formal verification
Well Known Bugs

**Ariane 5** explosion on its first flight; caused by reuse of some parts of a code from its predecessor without proper verification

**Therac-25** radiation therapy machine; due to a software error, six people are believed to die because of overdoses

**Pentium FDIV** design error in a floating point division unit; Intel was forced to offer replacement of all flawed processors
Outlook

- this lecture (foundations):
  - basics of a model checking technique
  - overview of modeling formalisms, logics
  - basic algorithms

- next lectures (real-time, applications):
  - theory: timed automata
  - extensions for practical modeling
  - verification tool Uppaal
  - case studies, realistic examples
Goal of the Lecture

- goal: to understand the basic principles of model checking technique
- important for efficient use of a model checking tool
Overlap with Other Courses

- IV113 Introduction to Validation and Verification
- IA159 Formal Verification Methods
- IA040 Modal and Temporal Logics for Processes
- IA006 Selected topics on automata theory

Verification in this course:
- foundations only briefly
- real-time aspects
Contents

2 Modeling
   - Guarded Command Language
   - Finite State Machines
   - Other Modeling Formalisms

3 Specification
   - Types of Properties
   - Temporal Logics
   - Timed Logics

4 Algorithms
   - State Space Search
   - Logic Verification
   - State Space Explosion
Model Checking

- **automatic** verification technique
- user produces:
  - a model of a system
  - a logical formula which describes the desired properties
- model checking algorithm:
  - checks if the model satisfies the formula
  - if the property is not satisfied, a counterexample is produced
Model Checking (cont.)

- specification
- temporal logic
- model checking
- system
- formal model
model checking algorithms are based on state space exploration, i.e., “brute force”

state space describes all possible behaviours of the model

state space $\sim$ graph:
- nodes $=$ states of the system
- edges $=$ transitions of the system

in order to construct state space, the model must be closed, i.e., we need to model environment of the system
Example: Model and State Space

Model Checking

NC
Wait
CS

f:=0
f=0
f:=1

N,N,0
W,N,0
W,W,0
N,W,0
N,C,1
C,N,1
C,W,1
W,C,1
Model Checking: Steps

1. modeling: system → model
2. specification: natural language specification → property in formal logic
3. verification: algorithm for checking whether a model satisfies a property
Modeling Formalisms

- **guarded command language**: simple low level modeling language
- **finite state machines**: usually extended with variables, communication
- **Petri Nets**: graphical modeling language
- **process algebra**: infinite state systems
- **timed automata**: focus of the next lecture
the simplest modeling language
not useful for actual modeling
simple to formalize
- we discuss formal syntax and semantics
- foundation for later discussion of timed automata
Guarded Command Language

- integer variables
- rules:
  - if condition then update
- conditions: boolean expressions over variables
- updates: sequences of assignments to variables
Example

\[ a : \text{if} \ x = 0 \ \text{then} \ x := 1 \]
\[ b : \text{if} \ y < 2 \ \text{then} \ y := y + 1 \]
\[ c : \text{if} \ x = 1 \land y \geq 1 \ \text{then} \ x := 0, \ z := 1 \]

Notes:
- this is an artificial example (does not model anything meaningful)
- \(a, b, c\) are names of actions
- no control flow
- rules executed repeatedly
- initial state: \(x = 0, y = 0, z = 0\)
Guarded Command Language

Syntax

- let $V$ be a finite set of integer variables
- expressions over $V$ are defined using standard boolean ($=, <$) and binary ($+, -, \cdot, ...$) operations
- model is a tuple $M = (V, E)$
- $E = \{t_1, \ldots, t_n\}$ is a finite set of transitions, where $t_i = (g_i, u_i)$:
  - predicate $g_i$ (a boolean expression over $V$)
  - update $u_i(\bar{x})$ (a sequence of assignments over $V$)
The semantics of model $M$ is a state space (formally called *Kripke structure*) $\semantics{M} = (S, \rightarrow, s_0, L)$ where

- states $S$ are valuations of variables, i.e., $V \rightarrow \mathbb{Z}$
- $s \rightarrow s'$ iff there exists $(g_i, u_i) \in T$ such that $s \in \semantics{g_i}, s' = u_i(s)$
  - semantics $\semantics{g_i}$ of guards and $u_i(s)$ is the natural one
- $s_0$ is the zero valuation ($\forall v \in V : s_0(v) = 0$)
Example

\[
\begin{align*}
a & : \text{if } x = 0 \text{ then } x := 1 \\
b & : \text{if } y < 2 \text{ then } y := y + 1 \\
c & : \text{if } x = 1 \land y \geq 1 \text{ then } x := 0, z := 1
\end{align*}
\]

Construct the state space.
Example

$a : \text{if } x = 0 \text{ then } x := 1$

$b : \text{if } y < 2 \text{ then } y := y + 1$

$c : \text{if } x = 1 \land y \geq 1 \text{ then } x := 0, z := 1$
Application

- simple to formalize, powerful (Turing power)
- not suitable for “human” use
- some simple protocols can be modeled
- control flow – variable pc (program counter)
Example: Ticket Protocol

The system with $n$ processes

Program

\[
\begin{align*}
\text{global var } & s, t : \text{integer;} \\
\text{begin} & \\
& t := 0; \\
& s := 0; \\
& P_1 \mid \cdots \mid P_n; \\
\text{end.}
\end{align*}
\]

The $i$-th component

Process $P_i$ ::= 

\[
\begin{align*}
\text{local var } & a : \text{integer;} \\
\text{repeat forever} & \\
& \begin{cases}
\text{think \quad \langle \ a := t; \\ t := t + 1; \rangle } \\
\text{wait \quad \text{when} \ \langle \ a = s \rangle \ \text{do} \\
\text{use \quad \text{critical section} \\
\text{\qquad \langle \ s := s + 1; \rangle} \\
\text{end.}
\end{cases}
\end{align*}
\]
Guarded Command Language

Example: Ticket Protocol

\[
\begin{align*}
\text{pc1} & := 0; \quad \text{pc2} := 0; \\
t & := 0; \quad s := 0; \quad a1 := 0; \quad a2 := 0; \\
\text{pc1} = 0 & \rightarrow \text{pc1} := 1, \quad a1 := t, \quad t := t + 1; \\
\text{pc1} = 1 & \quad \text{&&} \quad a1 <= s \rightarrow \text{pc1} := 2; \\
\text{pc1} = 2 & \rightarrow \text{pc1} := 0, \quad s := s + 1; \\
\text{pc2} = 0 & \rightarrow \text{pc2} := 1, \quad a2 := t, \quad t := t + 1; \\
\text{pc2} = 1 & \quad \text{&&} \quad a2 <= s \rightarrow \text{pc2} := 2; \\
\text{pc2} = 2 & \rightarrow \text{pc2} := 0, \quad s := s + 1;
\end{align*}
\]
Extended Finite State Machines

- each process (thread) is modelled as one finite state machine (machine state = process program counter)
- machines are extended with variables:
  - local computation: guards, updates
  - shared memory communication
- automata can communicate via channels (with value passing):
  - handshake (rendezvous, synchronous communication)
  - asynchronous communication via buffers
Example: Peterson’s Algorithm

- `flag[0], flag[1]` (initialized to `false`) — meaning / *I want to access CS*
- `turn` (initialized to `0`) — used to resolve conflicts

**Process 0:**

```plaintext```
while (true) {
    <noncritical section>;
    flag[0] := true;
    turn := 1;
    while flag[1] and turn = 1 do { };
    <critical section>;
    flag[0] := false;
}
```

**Process 1:**

```plaintext```
while (true) {
    <noncritical section>;
    flag[1] := true;
    turn := 0;
    while flag[0] and turn = 0 do { };
    <critical section>;
    flag[1] := false;
}
```
Example: Peterson’s Algorithm

Exercise: create a model of Peterson’s Algorithm using extended finite state machines, i.e., of the following type:
Example: Peterson’s Algorithm

Finite State Machines

1. NCS
   - flag[0] := 1
   - flag[0] := 0
   - flag[1] == 0 or turn == 2

2. W1
   - turn := 1

3. CS

4. W2
Art of Modeling

- choosing the right level of abstraction
- depends on purpose of the model, assumption about the system, ...
- example: if \( x == 0 \) then \( x := x + 1 \)
  - one atomic transition
  - two transitions: test, update (allows interleaving)
  - multiple “assembler level” transitions: if, load, add, store
EFSM: Semantics

- formal syntax and semantics defined in similar way as for guarded command language
- just more technical, basic idea is the same
- note: state space can be used to reason about the model – e.g., to prove mutual exclusion requirements (cf. Assignment 1)
Example: Peterson’s Algorithm
Finite State Machines

Example: Communication Protocol
Example: Elevator

**Fig. 1.13.** The cabin

**Fig. 1.14.** The $i^{th}$ door
Example: Elevator
Application: Verification of Link Layer Protocol
Layer Link Protocol of the IEEE-1394

- model of the “FireWire” high performance serial bus
- $n$ nodes connected by a serial line
- protocol consists of three stack layers:
  - the transaction layer
  - the link layer
  - the physical layer
- link layer protocol – transmits data packets over an unreliable medium to a specific node or to all nodes (broadcast)
- transmission can be performed synchronously or asynchronously
Finite State Machines

Notes

- link layer
  - main focus of verification
  - modeled in high detail
- transportation layer, physical layer (bus)
  - “environment” of link layer
  - modeled only abstractly
Finite State Machines
Finite State Machines
Timed Automata

- extension of finite state machines with clocks (continuous time)
- next lecture
Petri Nets: Small Example

graphical formalism (place, transitions, tokens)
Petri Nets: Realistic Model
Other Modeling Formalisms

**Process Algebra**

- $A \xrightarrow{a} XX$
- $X \xrightarrow{b} A \parallel B$

- basic process algebra (BPA), basic parallel processes (BPP)
- infinite state system modeling (e.g., recursion)
- mainly theoretical research
Specification of Properties

- properties the verified system should satisfy
- expressed in a formal logic
## Safety and Liveness

<table>
<thead>
<tr>
<th>Safety</th>
<th>Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>“nothing bad ever hap-</td>
<td>“something good eventually happens”</td>
</tr>
<tr>
<td>pens”</td>
<td></td>
</tr>
<tr>
<td>example: error state is</td>
<td>example: when a request is issued,</td>
</tr>
<tr>
<td>never reached</td>
<td>eventually a response is generated</td>
</tr>
<tr>
<td>verification = reachability problem, find a run which violates the property</td>
<td>verification = cycle detection, find a run in which the ‘good thing’ is postponed indefinitely</td>
</tr>
</tbody>
</table>
Examples of Safety Properties

- no deadlock
- mutual exclusion is satisfied
- a corrupted message is never marked as a good one
- the wheels are in a ready position during the landing
Examples of Liveness Properties

- each process can eventually access critical section
- each request will be satisfied
- a message is eventually transmitted
- there will be always another sunrise
Temporal Logic

- temporal logic is a formal logic used to reason about sequences of events
- there are many temporal logics (see the course IA040)
- the main classification: linear X branching
Linear Temporal Logics (LTL)

- \( X \phi \) neXt
- \( F \phi \) Future
- \( G \phi \) Globally
- \( \psi U \phi \) Until
LTL: Examples

- A message is eventually transmitted: $F \text{ transmit}$
- Each request will be satisfied: $G \ (\text{request} \Rightarrow F \ \text{response})$
- There will be always another sunrise: $G \ F \ \text{sunrise}$
- The road will be dry until it rains: $\text{dry} \ U \ \text{rains}$
- Process waits until it access CS: $\text{wait} \ U \ \text{CS}$
LTL: Examples

What is expressed by these formulas? For each formula draw a sequence of states such that the formula is a) satisfied, b) not satisfied.

- $GFa$
- $FGa$
- $G(a \Rightarrow Fa)$
- $aU(bUc)$
- $(aUb)Uc$
Timed Logics

- classical temporal logics
  - good for reasoning about sequences of states
  - may be insufficient for dealing with real time
- real time extensions
Metric Interval Temporal Logic (MITL)

- extension of LTL
- temporal operator can be restricted to certain interval
- examples:
  - $G\ (req \Rightarrow F_{\leq 3} \text{serv})$
    any request will be serviced within three time units
  - $dry \ U_{[12,14]} \ \text{rains}$
    after lunch it will rain, until that the road will be dry
Timed Logics

Specification in Practice

- timed logics – mainly theoretical research
- practical specification of properties:
  - classical temporal logics
  - often limited subset or only specific patterns
State Space Search

- construction of the whole state space
- verification of simple safety properties (e.g., mutual exclusion) = basically classical graph traversal (breadth-first or depth-first search)
- graph is represented implicitly = constructed on-demand from the model (description)
Logic Verification

- transformation to automata
- Buchi automaton: finite automaton over infinite words
- a word is accepted if the run of the automaton visits an accepting state infinitely often (compare with a final state for finite words)
Example

property: $G(req \Rightarrow F_{serv})$

negation: $F(req \land G\neg serv)$
Product Automaton

- property $\phi \rightarrow$ automaton for the negation of the property $A_{\neg \phi}$
- state space of the model $S +$ automaton $A_{\neg \phi} \rightarrow$ product automaton $S \times A_{\neg \phi}$
- product automaton represents erroneous runs
Product Automaton: Emptiness Check

model satisfies property $\iff$ the language of the product automaton is empty

- verification is reduced to non-emptiness check of product automaton
- Buchi automata: non-emptiness check is performed by (accepting) cycle detection
State Space Explosion

- size of the state space grows very quickly (with respect to size of the model)
- the worst case: exponential increase (next slide)
- theory: most interesting model checking problems are PSPACE-complete
- practice: the worst case does not occur, nevertheless memory/time requirements are very high
For \( n \) processes the number of states is \( 2^n + n \cdot 2^{n-1} \).
Dealing with State Space Explosion

- abstraction
- reduction techniques
- efficient implementations
Abstraction

- data abstraction (e.g., instead of \( \mathbb{N} \) use \( \{\text{blue, red}\} \))
- automated abstraction
- abstract - model check - refine
Reduction Techniques

- symmetry – consider only one of symmetric states
- partial order – consider only one of equivalent interleavings
- compositional construction – build the state space in steps
Efficient Implementations

- efficient representation of states, sets of states (symbolic methods — Binary Decision Diagrams)
- low level optimizations (e.g. memory management)
- distributed algorithms on networks of workstations
- randomization, heuristics – guiding toward errors
Model Checking: History

- 80’s: basic algorithms, automata theory, first simple tools, small examples
- early 90’s: reduction techniques, efficient versions of first tools, applications to protocol verification
- late 90’s: extensions (timed, probabilistic), first commercial applications for hardware verification
- state of the art: automatic abstraction, combination with other techniques, research tools for software verification, hardware verification widely adopted
Summary

- formal verification
- model checking: modeling, specification, verification
- modeling formalisms: guarded command language, finite state machines, Petri nets, ...
- formal property specification: temporal logics
- algorithms: state space search, Buchi automata, techniques for reducing state space explosion