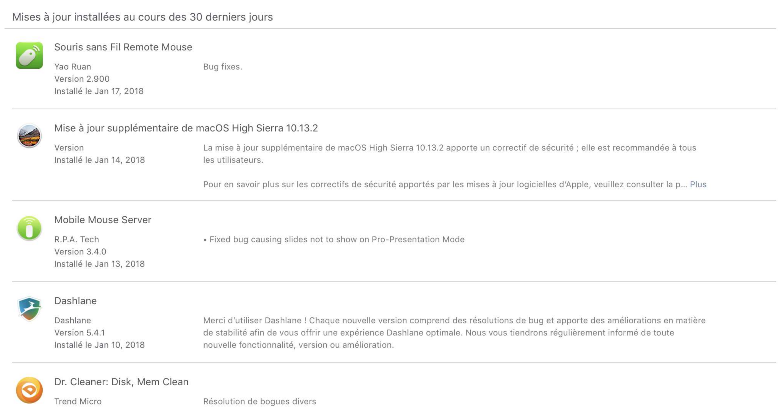
Introduction to Software Verification

Anca Muscholl, Marc Zeitoun, LaBRI, U. Bordeaux

January 2019

Hunting bugs: why?

- Bugs are an integral part of computer science.
- Are bugs really serious? No, just a bit annoying. . .



Hunting bugs: why?

- Bugs are an integral part of computer science.
- Are bugs really serious? Actually, bugs are often **CRITICAL**...

Mariner 1

Spirit

Ariane 5

Some software bugs with disastrous consequences (1)

Aircraft and aerospace industry

1962 **Loss** of proper route of **Mariner 1** (NASA), 5mn after launch.
Possible cause: transcription error in an equation.
The most expensive hyphen in history (A. C. Clarke)

1996 **Ariane 5 self-destruction** (maiden flight), 37 sec. after takeoff.
Cause. Conversion error 64 bits float to 16 bits integer.
US $370 million

2004 **Lock** of the **Mars Rover Spirit** robot.
Cause. Too many open files in flash memory.

Some software bugs with disastrous consequences (1)

Aircraft and aerospace industry

1962 **Loss** of proper route of **Mariner 1** (NASA), 5mn after launch.
Possible cause: transcription error in an equation.
The most expensive hyphen in history (A. C. Clarke)

1996 **Ariane 5 self-destruction** (maiden flight), 37 sec. after takeoff.
Cause. Conversion error 64 bits float to 16 bits integer.
US $370 million

2004 **Lock** of the **Mars Rover Spirit** robot.
Cause. Too many open files in flash memory.

Health

85–87 **5 deaths** by massive irradiation, dues to the **Therac-25** unit.
Cause. Conflicting software access to shared resources.

Some software bugs with disastrous consequences (2)

Communication

1990 Large scale **crash** of the AT&T network, domino effect.
Cause. Defecting units were alerting their neighbors…

Some software bugs with disastrous consequences (2)

Communication

1990 Large scale **crash** of the AT&T network, domino effect.
Cause. Defecting units were alerting their neighbors…
but receiving alert messages crashed receivers!

Some software bugs with disastrous consequences (2)

Communication

1990 Large scale **crash** of the **AT&T** network, domino effect.
Cause. Defecting units were alerting their neighbors…
but receiving alert messages crashed receivers!

Energy

2003 **General Electric** Northeast **blackout**, USA & Canada.
Cause. Again: bad handling of concurrent access to shared resources in a monitoring program.

Finance

2/2012 **Tokyo Stock Market** stuck because of a bug.

Some software bugs with disastrous consequences (3)

Computer Science

1994 **Pentium FDIV Intel** bug on floating point operations.
Cause. Flawed division algorithm (found by T. Nicely).

06–08 **OpenSSL** generated keys and insecure encrypted data
$\Rightarrow$ all software using OpenSSL (such as `ssh`) affected.
Cause. Broken random number generator in OpenSSL.

78–95 Flaw in the **Needham-Schroeder** authentication protocol.
Cause. Possible Man in the middle attack, discovered by G. Lowe.
Design bug!

Are bugs common?

http://en.wikipedia.org/wiki/List_of_software_bugs

- Upgrade your smartphone apps to check.

Are bugs common?

http://en.wikipedia.org/wiki/List_of_software_bugs

- Upgrade your smartphone apps to check.
- Some examples:
  - **Carte Vitale** bug, January 2013.
  - **SFR** bug, affecting unemployment statistics, October 2013.
  - **Heartbleed** SSL bug, April 2014.
  - **Meltdown** and **Spectre**, affecting INTEL processors, 2018.

Baisse du chômage. Un bug chez SFR a faussé les chiffres

France - 01 Octobre

Le ministère du Travail a reconnu que la forte baisse du chômage au mois d'août (-50 000 demandeurs d'emploi) résultait en partie d'une panne chez l'opérateur SFR.

L'annonce, mercredi dernier, d'une forte baisse des inscrits à Pôle emploi, au mois d'août, après 27 mois consécutifs de hausse, n'était pas tout à fait le résultat du hasard.

Michel Sapin, qui avait réfuté toute **« aberration statistique »**, tout en restant prudent, a dû se résoudre, lundi, à minorer les chiffres avancés dans un premier temps.

Intel : ses processeurs récents eux-aussi touchés par des problèmes de patche…

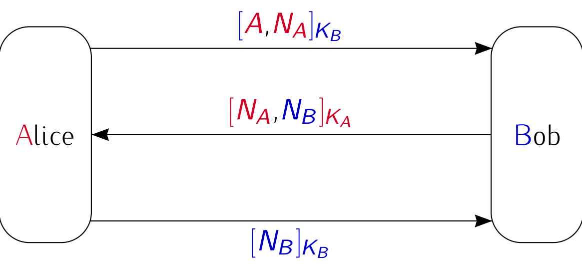
01Net

Il y a 20 heures

A concrete example: Needham-Schroeder protocol

- Goal of the protocol: authentication on a network.
- Means: each agent A has a pair of "keys":
  - K_A^{-1}: private key, known only by A. Plays the role of a **key**.
  - K_A, public key, known by everyone. Plays the role of a **lock**.
    Used to encrypt the messages sent to A.
    $[m]_{K_A}$ means "message m encrypted with K_A".
- Each agent tries to ascertain the identity of the other agent
  - by sending a random encrypted number, called cryptographic nonce,
  - and by requesting the nonce to be decoded and sent back.

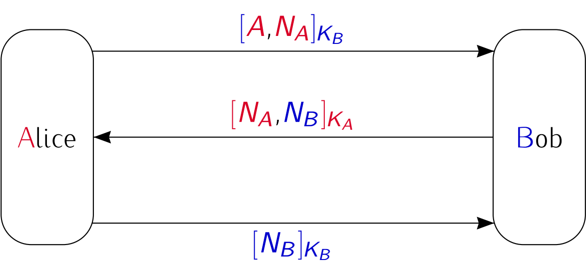
Needham–Schroeder protocol



Hypotheses

- a message can be intercepted, but only the owner of the corresponding private key can decode it.
- Keys cannot be forged.

Needham-Schroeder protocol



Hypotheses

- a message can be intercepted, but only the owner of the corresponding private key can decode it.
- Keys cannot be forged.

(Wrong) intuition

- Alice's belief: the only one who can discover N_A is the owner of K_B private key, so that's Bob.
- Symmetric reasoning for Bob.

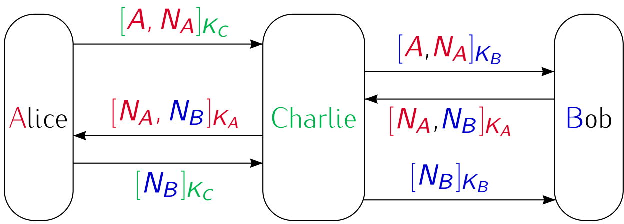
Needham-Schroeder protocol: attack

If Alice initiates the protocol to communicate with Charlie (the bad guy)...

... Charlie can fool Bob and pretend to be Alice.

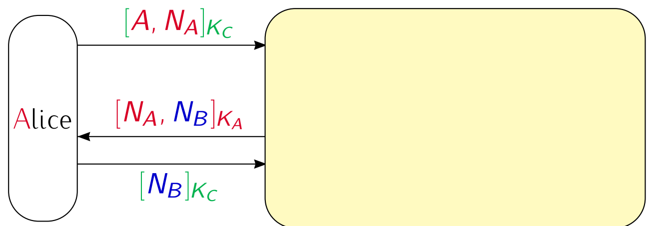
Needham-Schroeder protocol: attack

If Alice initiates the protocol to communicate with Charlie (the bad guy)...
... Charlie can fool Bob and pretend to be Alice.



Needham-Schroeder protocol: attack

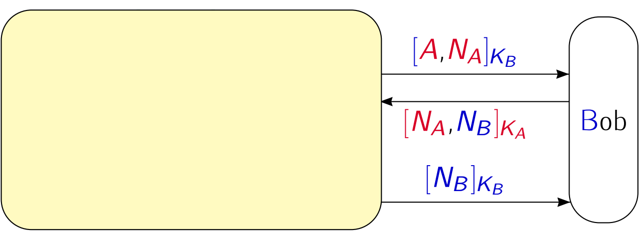
If Alice initiates the protocol to communicate with Charlie (the bad guy)...
... Charlie can fool Bob and pretend to be Alice.



- Alice and Bob honestly follow the protocol (not Charlie).
- Alice talks to Charlie: ok.

Needham-Schroeder protocol: attack

If Alice initiates the protocol to communicate with Charlie (the bad guy)...
... Charlie can fool Bob and pretend to be Alice.

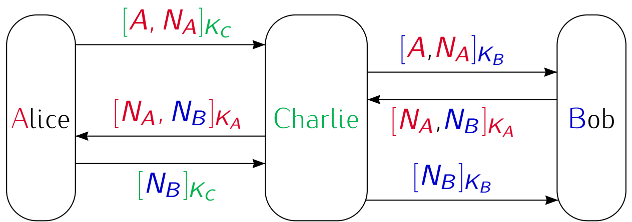


- Alice and Bob honestly follow the protocol (not Charlie).
- Alice talks to Charlie: ok.

Needham–Schroeder protocol: attack

If Alice initiates the protocol to communicate with Charlie (the bad guy)...

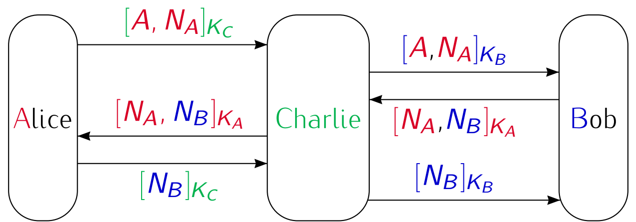
... Charlie can fool Bob and pretend to be Alice.



- Alice and Bob honestly follow the protocol (not Charlie).
- Alice talks to Charlie: ok.
- But Bob talks to Charlie, while thinking talking to Alice (N_B has kindly been revealed to Charlie by Alice).

Needham-Schroeder protocol: attack

If Alice initiates the protocol to communicate with Charlie (the bad guy)...
... Charlie can fool Bob and pretend to be Alice.



- Alice and Bob honestly follow the protocol (not Charlie).
- Alice talks to Charlie: ok.
- But Bob talks to Charlie, while thinking talking to Alice (N_B has kindly been revealed to Charlie by Alice).
- Question: Propose a simple fix.

History of the protocol

1978 Published by Needham and Schroeder.

1989 “Shown” correct by Burrows, Abadi, and Needham.

1995 Shown incorrect by Lowe (after 17 years of use!).

1996 Shown incorrect by Lowe again with an automatic approach, by

1. **modeling** the protocol in CSP and
2. using the FDR **model-checker**.

How to detect and correct bugs?

Let's write a **super-compiler**, which would compile code **and** detect bugs.

How to detect and correct bugs?

Let's write a **super-compiler**, which would compile code and detect bugs.

2 issues

How to detect and correct bugs?

Let's write a **super-compiler**, which would compile code and detect bugs.

2 issues

- How to describe the expected behavior of programs under verification?
- Can we really write down such a super-compiler?

Programs cannot compute everything...

Rice's Theorem ruins our hopes

Every nontrivial property of recursively enumerable languages is

undecidable 😕

Programs cannot compute everything...

Rice's Theorem ruins our hopes

Every nontrivial property of recursively enumerable languages is

undecidable ☹ this raises challenges in CS ☺

Programs cannot compute everything...

Rice's Theorem ruins our hopes

Every nontrivial property of recursively enumerable languages is

undecidable ☹ this raises challenges in CS

In short: One cannot even write a **"super-compiler"** that would detect:

- unreachable code in input programs,
- violated assertions in input programs.

Programs cannot compute everything...

Rice's Theorem ruins our hopes

Every nontrivial property of recursively enumerable languages is

undecidable ☹ this raises challenges in CS

In short: One cannot even write a **“super-compiler”** that would detect:

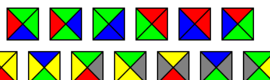
- unreachable code in input programs,
- violated assertions in input programs.

Even for programs using only 2 integer variables, no function call and only 2 instruction types:

- `x++`
- `if (x==0) goto p else x-- goto q`

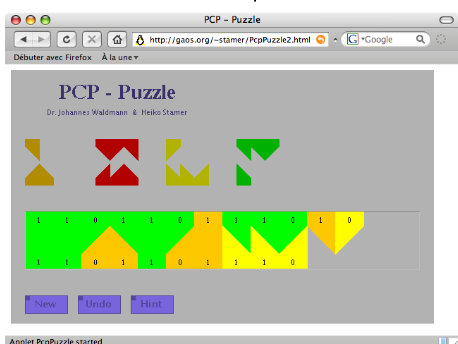
Programs cannot even tile the plane

- From a finite set of input tiles:



Can one tile a rectangular tiling respecting colors?

- ...another undecidable problem: Emil Post's puzzle



Hunting bugs is not easy

- One cannot automatically detect bugs.
- ... yet this is mandatory.

Software verification

Statement. Software bugs are very expensive.

To (try to) detect them, several **complementary, imperfect** approaches

- Simulation/test.
  - 🙂 Can find bugs,
  - 🙁 Cannot guarantee correctness.

Software verification

Statement. Software bugs are very expensive.

To (try to) detect them, several **complementary, imperfect** approaches

- Simulation/test.
  - 🙂 Can find bugs,
  - 🙁 Cannot guarantee correctness.
- Theorem proving.
  - 🙂 Provides guarantees on correctness.
  - 🙁 Not fully automatic, requires skills.

Software verification

Statement. Software bugs are very expensive.

To (try to) detect them, several **complementary, imperfect** approaches

- Simulation/test.
  - 🙂 Can find bugs,
  - 🙁 Cannot guarantee correctness.
- Theorem proving.
  - 🙂 Provides guarantees on correctness.
  - 🙁 Not fully automatic, requires skills.
- Model-checking.
  - 🙂 Provides guarantees on correctness.
  - 🙂 Fully automatic.
  - 🙁 Only works on more or less realistic system abstractions.
- ...

Software Model-checking. Clarke/Emerson & Queille/Sifakis, 1981

- Goal: to detect automatically bugs in circuits, protocols, etc.
- Effective during design stage, or else requires a modeling phase.
- Works on system model to check properties.

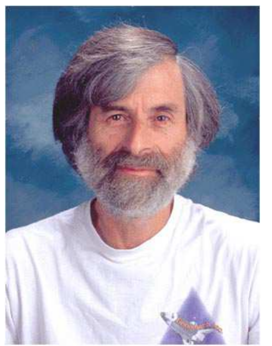
E.M. Clarke

E.A. Emerson

J. Sifakis

Turing Award 2007.

Another recent Turing Award in software verification



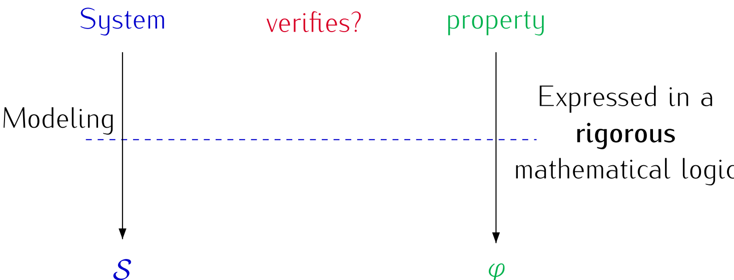
L. Lamport

Turing Award 2014.

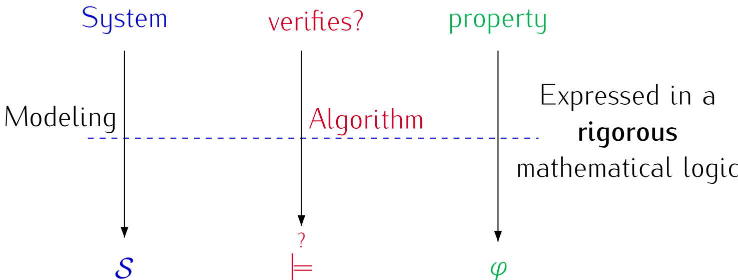
Model checking principle

System verifies? property

Model checking principle



Model checking principle



How to overcome Rice's theorem?

- Check less realistic models than Turing powerful ones, focusing on some aspects only.

  We will start with finite systems.

- Trade-off realistic models vs. expressiveness of logical language.

- Approximate model-checking, e.g.,
  - Check for a **bounded** number of computation steps.
  - Look for **semi-algorithms**, with no termination guarantee.
  - ...

Model checking finite systems: Kripke Structures

- Kripke structure $\mathcal{K}$: finite transition system whose states are labeled by atomic properties of a finite alphabet AP.
- Used to model finite systems.
  - State = snapshot of variable values & program counter.
  - Transitions = system's evolution.
- Each run generate a word on the alphabet $\Sigma = 2^{\mathsf{AP}} \rightsquigarrow L(\mathcal{K}) \subseteq \Sigma^{\omega}$.

Model checking finite systems: Kripke Structures

- Kripke structure $\mathcal{K}$: finite transition system whose states are labeled by atomic properties of a finite alphabet AP.
- Used to model finite systems.
  - State = snapshot of variable values & program counter.
  - Transitions = system's evolution.
- Each run generate a word on the alphabet $\Sigma = 2^{\mathsf{AP}} \rightsquigarrow L(\mathcal{K}) \subseteq \Sigma^{\omega}$.

- One uses a logical specification language to express properties
- A formula φ defines a language $L(\varphi)$ over $\Sigma = 2^{\mathsf{AP}}$.

Model checking finite systems: Kripke Structures

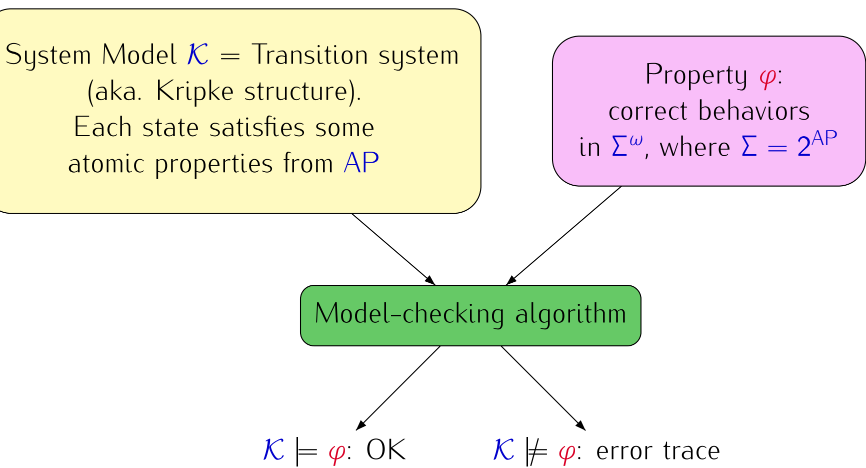
- Kripke structure $\mathcal{K}$: finite transition system whose states are labeled by atomic properties of a finite alphabet AP.
- Used to model finite systems.
  - State = snapshot of variable values & program counter.
  - Transitions = system's evolution.
- Each run generate a word on the alphabet $\Sigma = 2^{\mathsf{AP}} \rightsquigarrow L(\mathcal{K}) \subseteq \Sigma^\omega$.

- One uses a logical specification language to express properties
- A formula φ defines a language $L(\varphi)$ over $\Sigma = 2^{\mathsf{AP}}$.

- One can for instance check if **every** behavior of $\mathcal{K}$ satisfies φ:

$$L(\mathcal{K}) \subseteq L(\varphi)\ ?$$

Model-checking in a nutshell



Example: Peterson's algorithm

2 processes P_0, P_1. Shared variables `req[0]`, `req[1]` and `turn`.

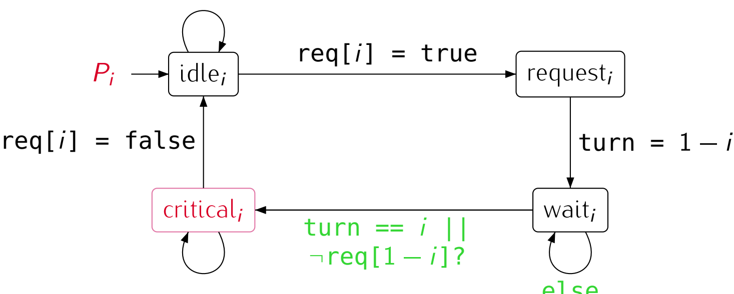
Code of P_i:

```
req[i] = true
turn = 1-i
while (req[1-i] && turn == 1-i)
      ;  // waiting in active loop
critical_section()
req[i] = false
```

- Guarantees several properties, in particular mutual exclusion.

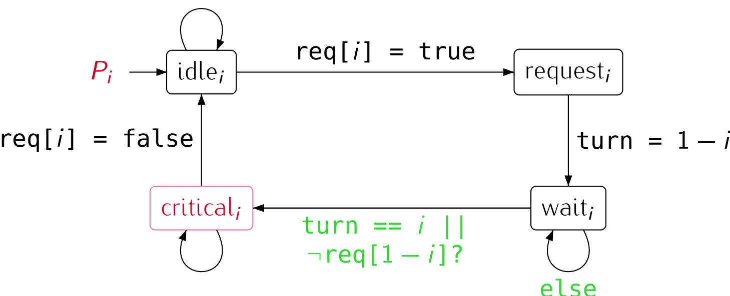
Modeling Peterson's algorithm

- 4 control states per process and 3 Boolean variables
  $\Rightarrow \leqslant 4^2 \times 2^3 = 128$ states for Kripke structure.
- Only ~ 30 reachable, but correctness wrt. MutEX not immediate.



Modeling Peterson's algorithm

- 4 control states per process and 3 Boolean variables
  $\Rightarrow \leqslant 4^2 \times 2^3 = 128$ states for Kripke structure.
- Only ~ 30 reachable, but correctness wrt. MutEX not immediate.



- Other MutEx algorithms: `http://en.wikipedia.org/wiki/Mutex`.
- Example: Dekker, $\sim$ 10 lines, $\sim$ 140 reachable states.

Model-checking in practice

1. Algorithm to compile a specification φ into an automaton $\mathcal{A}_\varphi$, which recognizes words satisfying φ models:

$$\forall t \in \Sigma^\omega : t \models \varphi \iff t \in L(\mathcal{A}_\varphi).$$

2. Check that all behaviors of $\mathcal{K}$ satisfy φ:

$$L(\mathcal{K}) \cap L(\mathcal{A}_{\neg\varphi}) = \varnothing \iff \mathcal{K} \models \varphi.$$

Model-checking in practice

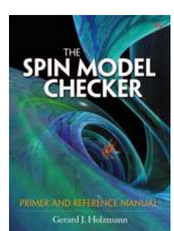
1. Algorithm to compile a specification φ into an automaton $\mathcal{A}_\varphi$, which recognizes words satisfying φ models:

$$\forall t \in \Sigma^\omega : t \models \varphi \iff t \in L(\mathcal{A}_\varphi).$$

2. Check that all behaviors of $\mathcal{K}$ satisfy φ:

$$L(\mathcal{K}) \cap L(\mathcal{A}_{\neg\varphi}) = \varnothing \iff \mathcal{K} \models \varphi.$$

3. For finite systems, this can be done automatically and quite efficiently, eg, with SPIN:



SPIN

Spin is a popular open-source software verification tool, used by thousands of people worldwide. The tool can be used for the formal verification of multi-threaded software applications. The tool was developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments. In April 2002 the tool was awarded the ACM System Software Award. [read more]

discover

- what is spin?
- success stories
- examples
- roots

learn

- tutorials
- books
- papers
- model extraction

use

- installation
- man pages
- options
- releases

community

- forum
- workshops
- support
- projects

The Spin 2014 workshop will be held at the Hilton San Jose, in Northern California, from 21-23 July 2014. The workshop organizers are Neha Rungta and Oksana Tkachuk.

- Abstract submission: 7 March 2014
- Papers are due: 14 March 2014
- Author notifications: 30 April 2014
- Camera ready papers: 9 May 2014

```
// a small example spin model
// Peterson's solution to the mutual exclusion problem (1981)

bool turn, flag[2];         // the shared variables, booleans
byte ncrit;                 // nr of procs in critical section

active [2] proctype user()  // two processes
{
    assert(_pid == 0 || _pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 || turn == 1 - _pid);

    ncrit++;
    assert(ncrit == 1);     // critical section
    ncrit--;

    flag[_pid] = 0;
    goto again
}
// analysis:
// $ spin -a peterson.pml
// $ cc -o pan pan.c
// $ ./pan
```

Download

Promela code for Peterson's algorithm

```
bool turn, flag[2]; // shared variables, booleans
byte ncrit;         // # procs in critical section

active [2] proctype user()      // two processes
{
  assert(_pid == 0 || _pid == 1);
  again:
  flag[_pid] = 1;
  turn = _pid;
  (flag[1 - _pid] == 0 || turn == 1 - _pid);

  ncrit++;
  assert(ncrit == 1);   // critical section
  ncrit--;

  flag[_pid] = 0;
  goto again
}
```

Short bibliography...

- Systems and Software Verification. Model-Checking Techniques and Tools. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, P. McKenzie. Springer, 2001.
- Principles of Model Checking. Ch. Baier, J.-P. Katoen. MIT Press, 2008.
- Model Checking. E.M. Clarke, O. Grumberg, D.A. Peled. MIT Press, 2000.
- 25 Years of Model Checking: History, Achievements, Perspectives. O. Grumberg, H. Veith (Eds). Springer, 2008.

What we will see in this course

- Systems: finite, timed, probabilistic, well-structured systems.
  - Modeling,
  - Verification algorithms.
- Some specification languages: LTL, CTL(*), TCTL, PCTL(*).
- Model-checking algorithms.
- Some model checkers like SPIN.