Related Courses:
• Programming Theory (Parosh Abdulla):
  – principles for verifying and analyzing sequential programs
• Formal Program Development (Lars-Henrik Eriksson):
  – Systematic Development of correct programs
• Software Engineering (Roland Bol):
  – Organizing the development of software systems
• Operating Systems, Real Time Systems, Computer Networks
  – Principles and algorithms for coordinating parallel and distributed systems
• Logic, Automata theory
  – We will use some of the theory from these courses.

Goal:
Modeling, Specifying, and Analyzing concurrent, parallel, and distributed algorithms, systems, and programs.

Contents:
• Modeling parallel systems (as transition systems)
• Specifying requirements and correctness properties
• Algorithms for automatically checking that a model satisfies a property
  – Model checking / state space exploration
• Application to algorithms encountered in operating systems/computer networks courses
• Analysis of concurrent software.
• Use of Software tool for all the above: SPIN

Administrative
Instructors:
• Bengt Jonsson, room 1435 bengt(at)it.uu.se
• Mayank Saksena, room 1137 mayanks(at)it.uu.se
Course page
http://www.it.uu.se/edu/course/homepage/verteknik/vt08/
Examination:
• 5 homework exercises (solved in pairs)
• Final exam on the topics covered in lectures.
SPIN
• You must use SPIN for the exercises.
  – You are encouraged to install SPIN on your own computer.
  – Installed at /stud/docs/kurs/ReactiveSystems/bin/spin
  – XSPIN at /stud/docs/kurs/ReactiveSystems/bin/xspin
  – Further material at http://spinroot.com/spin/

Course Material
You will need
• Lecture Handouts (slides)
• A few papers (will be distributed)
• SPIN documentation (on the WWW, and distributed)
Reference texts: You can choose one/several from
• SPIN MODEL CHECKER Primer and Reference Manual by Gerard Holzmann, good textbook for the course.
• Old notes prepared by me some years ago.

Structure of the Material
The course is a close interplay between
• Concepts and techniques for modeling, specification, and verification
• Implementation in the tool SPIN
  – almost an exact realization of the theory
• Application to examples
Examination

- Homeworks, to be solved in pairs. mandatory
- Final exam
  Each counts for half of your final grade.

HOW TO DO WELL:

- Do the homework seriously
- Make sure that you master the material so that you are able to solve all exercises with moderate effort
- Ask when things are not clear

COURSE OVERVIEW:

What problems can be solved?

Verification

Verification = "building the system right"

<table>
<thead>
<tr>
<th>System description</th>
<th>Correctness properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web server</td>
<td>Absence of</td>
</tr>
<tr>
<td>implementation</td>
<td>Run-time errors</td>
</tr>
<tr>
<td>Protocol standard</td>
<td>deadlocks</td>
</tr>
<tr>
<td>Functional spec.</td>
<td>Memory leaks</td>
</tr>
<tr>
<td></td>
<td>Protocol service</td>
</tr>
</tbody>
</table>

Verification

- Testing consumes ~half of software development effort
- Several "expensive" accidents caused by bugs
  - Ariane 5 crash 1996
  - Pentium division bug
  - Mars pathfinder ceased to work 1997
  - Viruses, ....

Some of the Improvements needed

Better Development tools
- Programming languages
- Development environments/Libraries
- Software architectures
Better Skilled People
- Better designers
- better programmers
Better Processes
- Better Collaboration between developers / with customers
- Better Documentation
Better Verification Techniques
- Testing and verification: This (and other) courses

Motivation: Idealized Design process

Requirements
- High level design

Detailed design
- coding
- testing
- deployment
Motivation: Idealized Design process

- Requirements
  - High level design
  - Detailed design

  Where most design errors are made

- Coding
  - Testing
  - Deployment

  Where most design errors are found

Introducing, Detecting and Correcting errors: cost

- Errors detected: the later the more expensive

Testing:
- By far the most used technique
- The most "practical" technique
- Can verify a wide range of properties
- Can only be used on implementation
- Difficult to make exhaustive
- Hard to make reproducible for concurrent/distributed programs
- Manual selection of test cases and input needs work.

Prototyping and Simulation:
- Can be used on design level
- Difficult to make exhaustive
- Manual selection of test cases and input needs work

Code and Design Reviews:
- Good at finding (some classes of) problems
- Needs organization and people

Verification Techniques: short overview

Testing: By far the most used technique
- Completely automatic
- Can verify a limited set of properties (type-correctness, absence of some run-time errors)
- Tools available only for some languages and properties

Prototyping and Simulation:
- Can be used on design level
- Difficult to make exhaustive
- Manual selection of test cases and input needs work

Code and Design Reviews:
- Good at finding (some classes of) problems
- Needs organization and people

Verification Techniques: short overview

Static Program Analysis: Analyzing the source code by tools
- Can verify a limited set of properties (type-correctness, absence of some run-time errors)
- Tools available only for some languages and properties

Model Checking: Analyzing a prototype/model by tools
- Can be done early in the design cycle, e.g., on design level.
- Automated (provided tools available)
- Can check many kinds of properties
- A model must be constructed (at a suitable level of abstraction)
- Model must be maintained when system evolves.
- Does not scale to very large models

Motivation: Purpose of Model Verification

- Build model of the design. Analyze it thoroughly
- Implement your model, and spend effort to check conformance to the model

Verification Techniques: short overview

- Checking correctness of
  - Communication protocols
  - Distributed Algorithms
  - Controllers
  - Hardware circuits
  - Embedded and real-time systems and software
  - e.g., Absence of race conditions, deadlocks, livelocks, priority inversions, proper synchronization, ...

Model checking is the appropriate technique when there are many many different scenarios of interaction between components in a system

Problems that can be addresed by Model Checking
**Merits of this simpler approach**

- Checking simple properties (e.g. deadlock freeness) is already extremely useful!
- The goal is no longer seen as proving that a system is completely correct (bug-free)
- The objective is to have tools that can help a developer find errors and gain confidence in her/his design. That is achievable
- Now widely used in hardware design, protocol design, embedded systems, ...

---

**CONTRAST: Assertional verification for “data-centric” programs**

What does this program do?

```
start
y1,y2=x1,x2
print(y1)
stop
```

It computes the Greatest Common Divisor (gcd) of x1 and x2

---

How can a program check this fact?

```
start
y1,y2=x1,x2
print(y1)
stop
```

Can this be checked by a computer?
Static Analysis: Example (input)

\[ \begin{align*}
  n &:= n_0; \\
  i &:= 0; \\
  \text{while} &\{ i < 0 \} \text{ do} \\
  \quad j &:= 0; \\
  \text{while} &\{ j < 0 \} \text{ do} \\
  \quad j &:= j + 1; \\
  \quad i &:= i - 1; \\
  \text{od}; \\
\end{align*} \]

Lecture 1

---

Static Analysis: Example (output)

\[ \begin{align*}
  n &:= n_0; \\
  i &:= 0; \\
  \text{while} &\{ i < 0 \} \text{ do} \\
  \quad j &:= 0; \\
  \text{while} &\{ j < 0 \} \text{ do} \\
  \quad j &:= j + 1; \\
  \quad i &:= i - 1; \\
  \text{od}; \\
\end{align*} \]

Lecture 1

---

Overview of Model Checking

Model: M

Promela

Property: \( \phi \)

Model Checker

Yes!

No!

Error trace

---

Unveiling bad mutual exclusion algorithm

```c
/* Bad Mutual Exclusion */
n int a, p, e;
void Symmetry(Iod)

begin
  if 0 < e && e < p
  go begin;
  x = 16;
  y = 1;
  go begin;
  if e < 8
  go begin;
  x = 16;
  y = 1;
  go begin;
  e = 0;
  x = 0;
  y = 0;
```
Example with Concurrency

```c
int main(void)
{
    thread_t thread_id, main_id;
    main_id = thr_self();
    thr_setconcurrency(2);
    thr_create(NULL, 0, thread_sub, (void *)main_id, THR_SUSPENDED, &thread_id);
    while(1) {
        printf("MAIN: continuing subroutine thread
"), fflush(stdout);
        thr_continue(thread_id);
        printf("MAIN: suspending self
"), fflush(stdout);
        thr_suspend(main_id);
    }
    return(0);
}

void *thread_sub(void *arg) {
    thread_t thread_id;
    thread_t main_id = (thread_t) arg;
    thread_id = thr_self();
    while(1) {
        printf("THREAD: continuing main thread
"), fflush(stdout);
        thr_continue(main_id);
        printf("THREAD: suspending self
"), fflush(stdout);
        thr_suspend(thread_id);
    }
    return((void *)0);
}
```

Model the example for the SPIN tool

```c
bool Suspend_main, Suspend_thread, arg;
active proctype main() provided (!Suspend_main) {
    L_0:
    do ::
        Printf("MAIN: continuing subroutine thread
"),
        fflush(stdout);
        Suspend_thread = 0;
        Printf("MAIN: suspending self
"),
        fflush(stdout);
        Suspend_main = 1;
    od;
    goto Return;
}

active proctype thread() provided (!Suspend_thread) {
    L_1:
    do ::
        Printf("THREAD: continuing main thread
"),
        fflush(stdout);
        Suspend_main = 0;
        Printf("THREAD: suspending self
"),
        fflush(stdout);
        Suspend_thread = 1;
    od;
    goto Return;
}

Desiderata for good a model

- Captures essentials of behavior of system/program/algorith
- Should be simple to understand, and well-structured
  - To validate that you model the correct thing
- Can be thoroughly analyzed (e.g., by SPIN)
  - Avoid unnecessary complications
  - Try to abstract/simplify necessary complicated aspects
  - Not "too big"

Output from analysis by SPIN

```
THREAD: continuing main thread
THREAD: suspending self
MAIN: continuing subroutine thread
THREAD: continuing main thread
THREAD: suspending self
MAIN: suspending self
18: main():[Suspend_main = 1]
spin: trail ends after 18 steps
5:proc 0 (main) line 5 (state 7) (invalid and state)
Print("MAIN: continuing subroutine thread
"),
18: proc 1 (thread) line 20 (state 7) (invalid and state)
Print("THREAD: continuing main thread
"),
global vars:
bit Suspend_main: 1
bit Suspend_thread: 1
bit arg: 0
...```

Hippies problem

```
Hippies must get across bridge. There is only one torch. At most two people can cross together. Can all cross in at most 60 minutes?
```

How to make models

- By hand from a description of algorithm/system
- As specification during system design
- When analyzing existing programs
- By hand from code
- By automated extraction tools from program code
  - ModEx adapts ANSI-C code to SPIN
  - Hard problem: automated simplification
- Automatically from test suites
**An 'abstract' version of a field bus protocol**

**Remaining Problems**

Constructing a Model
- not so easy, this course will make you experts

Making absolutely sure that the actual system/software conforms to the model
- hard problem: there are several techniques:
  - Conformance testing
  - Static program analysis
  - Automated code generation

**Small Idealized Example from Reality**

Small Example: Mars Pathfinder 1997

Typical properties of synchronization in real-time systems
- Mutual exclusion
  - A process cannot access the data-bus unless it owns a mutex-lock
- Scheduling priority
  - Saving data to memory has higher priority than processing data
  - Low priority process cannot execute when high priority process is ready to execute or executes

**Idealized model of processes**

Variables: \( l \) : integer
Initially \( l = 1 \)

High priority process

Low priority process

Initial states:
- High priority: \( l = 1 \)
- Low priority: \( l = 0 \)

Transitions:
- \( l = 1 \rightarrow l = 0 \)
- \( \text{High@idle} \rightarrow l = 1 \)
- \( \text{High@idle} \land l = 1 \rightarrow l = 0 \)