Verification Techniques, Winter/Spring 2009

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

Instructors:
- Bengt Jonsson, room 1435   bengt(at)it.uu.se
- TBD

Course page
http://www.it.uu.se/edu/course/homepage/verteknik/vt09/

Examination:
- 3 homework exercises (solved individually or in pairs)
- "mini-project": model, specify, and analyze a case study.
- 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
- jSPIN at http://stwww.weizmann.ac.il/~g-cs/benari/jspin

You will need
- Lecture Handouts (slides)
- A few papers (will be distributed)
- SPIN documentation (on the WWW, and distributed)

Reference texts: Recommend to choose one/several from
- Mordechai Ben-Ari, Principles of the Spin Model Checker, Springer Verlag, 2008, pedagogic textbook covering Promela
- 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 individually or in pairs. **mandatory**
- "mini-project": model, specify, and analyze a case study. **Most important part of course.**
- Final exam, covering lectures
  Each counts for a third of your final grade.

**HOW TO DO WELL:**

- Do the homework seriously
- Make sure that you master the material to make a good mini-project
- Ask when things are not clear

---

COURSE OVERVIEW: What problems can be solved?

---

Verification

- **Verification** = "building the system right"

  - **System description**
    - Web server implementation
    - Protocol standard
    - Functional spec.
  - **Correctness properties**
    - Absence of:
      - Run-time errors
      - Deadlocks
      - Memory leaks
    - Protocol service

---

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 testers and verifiers
- 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
- **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.

Verification Techniques: short overview

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

Static Program Analysis:
- 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

Model Checking:
- 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

- Requirements
  - High level design
  - Detailed design

Build model of the design. Analyze it thoroughly

- coding
- testing
- deployment

Implement your model, and spend effort to check conformance to the model

Problems that can be addressed by Model Checking

- Checking correctness of
  - Communication protocols
  - Distributed Algorithms
  - Controllers
  - Hardware circuits
  - Embedded and real-time systems and software
  - 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
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?
(Floyd 1967, Hoare 1969)

\begin{itemize}
  \item \texttt{y1, y2 := x1, x2}
  \item \texttt{y1 = y2}
  \item \texttt{y1 > y2}
  \item \texttt{y1 == y2}
\end{itemize}

Can this be checked by a computer?
Slide 22

WY3  Wang Yi; 2006-03-29

Slide 23

WY4  Wang Yi; 2006-03-29

Slide 24

WY5  Wang Yi; 2006-03-29
programming code and comments
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\n");
        fflush(stdout);
        thr_continue(thread_id);
        printf("MAIN: suspending self\n");
        thr_suspend(main_id);
    }
    return(0);
}
```

**Model the example for the SPIN tool**

```c
void **tread_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\n");
        fflush(stdout);
        thr_continue(main_id);
        printf("THREAD: suspending self\n");
        fflush(stdout);
        thr_suspend(thread_id);
    }
    return((void **)0);
}
```

**Output from analysis by SPIN**

```
THREAD: continuing main thread
THREAD: suspending self
MAIN: continuing subroutine thread
THREAD: continuing main thread
THREAD: suspending self
18: main(0):[Suspend_main = 1]
spin: trail ends after 18 steps
```

**Hippies problem**

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

**Desiderata for good a model**

- Captures essentials of behavior of system/program/algorithm
- 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"

**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
      - ModX adapts ANSI-C code to SPIN
    - Hard problem: automated simplification
      - Automatically from test suites
Example of Model

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

High priority process
Low priority process

Idealized model of processes

High@idle \rightarrow l := 1

l := 1 \rightarrow l := 0

High@idle \wedge l = 1 \rightarrow l := 0

Idealized model of processes

High@idle \rightarrow l := 1

l := 1 \rightarrow l := 0

High@idle \wedge l = 1 \rightarrow l := 0