Software Verification and Testing

Lecture Notes: Introduction to Formal Methods

Course Websites

official course description: (not up to date. . .)
www.dcs.shef.ac.uk/intranet/teaching/modules/msc/com6854.html

course material: (lectures, exercises, further information)
www.dcs.shef.ac.uk/~georg/COM6854.html

course description: (up to date)
www.dcs.shef.ac.uk/~georg/COM6854_description.html

SSIT website:
www.shef.ac.uk/dcs/postgrad/taught/sst.html
www.dcs.shef.ac.uk/intranet/teaching/modules/msc/ssit.html

my website: www.dcs.shef.ac.uk/~georg/

Organisation

lectures: monday 14–16

exercises: friday 11–12

exam: date to be announced.

questions: g.struth@dcs.shef.ac.uk

Course Outline

We explore selected formal methods for the description, construction and analysis of software systems.

Their objective is software reliability in applications where failure is unacceptable, e.g., e-commerce, telecommunication, transport, energy.

We focus on the development of formal specifications, on the verification of their behavioural properties, and on testing methods.

Course Outline

content: (roughly)

- introduction to formal methods [1 lecture]
- mathematical tools and models [2 lectures]
- formal specification languages [5 lectures]
- temporal logics and related formalisms [3 lectures]
- model checking [4 lectures]
- other verification techniques [1 lecture]
- testing [4 lectures]

Course Outline

what will we learn?

Seek simplicity and distrust it.
(A. N. Whitehead)

A Provocation

simple questions:

- why should you trust a bridge?
- why should you trust a bank transaction system?

observations:

- software successively replaces hardware
- traditional io-programs complemented by reactive and concurrent systems

myth: software engineers know what they are doing!

distrust: how much engineering is in software engineering?

Engineering

canon of standardised tools, methods and techniques:

- first empirical (cathedrals)
- then based on sophisticated mathematical models (statics, mechanics, electrodynamics, hydrodynamics,. . .)
- using elaborate simulation and testing procedures
- strict laws and standards for quality insurance

The Sorry State of Software Quality

some quotes:

- Cyber-Security Chief Voices Concerns About Software Quality (eweek.com, 2004)
- Who, if anyone is making bug-free software these days? (Information Week, 1999)
- Despite 50 years of progress, the software industry remains years-perhaps decades-short of the mature engineering discipline needed to meet the demands of an information-age society. (Scientific American, 1994)

diagnosis: since 1968, software crisis persists

The Sorry State of Software Quality

some numbers:

- software bugs cost US economy about $\$59.5$ billion dollars per year (NIST, 2002)
- best software companies remove $\sim 95\%$ of bugs; US average is $< 85\%$ (IEEE Software Magazine)
- experienced software engineers inject ~ 100 defects per kloc
- $< 1/3$ industrial software projects proceed as planned (Standish Group Report, 1995)
- $> 1/5$ projects completely fail (Standish Group Report, 1995)
- failure rate increasing. . . (Standish Group Report, 1995)
- look at http://spinroot.com/spin/Doc/course/Standish_Survey.htm

conclusion: never work as an industrial software engineer!

Where do Projects Fail?

Standish Group Report:

- high impact: incomplete or changing requirements and specifications, lack of user involvement
- low impact: hardworking staff, technology illiteracy

observation: projects fail in design phase, errors are discovered only later

task: catch errors early, save money!

Impact of Software Errors

observation: pc users are much more tolerant than car users

safety critical software: unreliability can lead to dramatic losses in

- e-commerce
- transport
- energy
- telecommunication

problem:

- software verification and testing can be very expensive!
- companies balance this with losses. . .

How to Avoid Software Errors?

in design phase: formal methods

in coding phase: testing

later: bug fixing with clients

warning: rare, but catastrophic errors are often not found by testing!

What are Formal Methods?

working definition: formal methods are techniques and tools that are based on models of mathematics and mathematical logic and support the description, construction and analysis of hardware and software systems.

aspects of formal methods:

- specification: build formal system models
- verification: prove that models behave as intended
- construction: derive correct executable code from formal specifications

What are Formal Methods Based On?

specification: formal languages, formal syntax, formal notation

verification: formal calculus, notion of proof, consistency, entailment

construction: notion of derivation, refinement, formal calculus

. . . in my opinion, the task of programming [...] can be accomplished by returning to mathematics.

(J.-R. Abrial)

Are Formal Methods Feasible?

Moore's law: cpu speed doubles every 18 months

problem solving: clever algorithms require less and less time and space

consequence: speed of automated formal methods currently increases faster than software complexity

trade-off:

- push-bottom methods (model checking) have limited applicability
- more powerful methods (theorem proving) require mathematical skills

theoretical limitations: decidability, complexity, halting problem, incompleteness

in practice: speed, simplicity, robustness, money. . .

Are Formal Methods Feasible?

truisms:

- quality of analysis depends on quality of model
- who verifies the verifier?

problem: models are often built by abstraction

- ok can be too optimistic
- counterexample can be artefact

consequence: verification should often be seen as debugging

Formal Methods

conclusion: formal methods

- are interesting, necessary and (quite) feasible
- could revolutionise our understanding of software development

question: should one work as a formal methodist?

answer: no! the success of a formal methodist is in the absence of failure.
managers don't see this and he will never get promotion. . .

Why Software Development Can be so Difficult

examples:

1. the Clayton tunnel accident
2. mutual exclusion
3. the Needham-Schröder authentication protocol

remark:

- (1) and (3) are protocols
- (2) is a distributed algorithm

Protocols

protocols: these are sets of rules that govern the interaction of concurrent processes in distributed systems

properties:

- protocol specifications can be very simple
- behaviour of protocols can be extremely difficult to analyse (combinatorial explosion)
- the problem of developing protocols dates back to the Greeks (and beyond)
- protocol errors belong to the most fascinating but tragical events in the history of engineering

. . . it is chiefly unexpeced occurances which require instant consideration and help.

(Polybios)

The Clayton Tunnel Accident

setting: 19th century

- 1.5 mile railway tunnel in Scotland with needle telegraph, signal man and semaphore at each end
- block-interval system: (only one train allowed in tunnel)
  - incoming train sets semaphore to red
  - signal man resets semaphore manually to green after train leaves tunnel
- telegraph messages:
  - `train in tunnel.`
  - `tunnel is free.`
  - `has train left tunnel?` (to make it fool-proof. . .)
- semaphore dysfunction: bell warns signal man to manage trains with red and green flag

The Clayton Tunnel Accident

the accident: (august 1861)

1. incoming train t_1 fails to set semaphore to red. Signalman m_1 hears bell, sends `train in tunnel.`, raises red flag
2. train t_2 sees red flag, but can stop only in tunnel
3. train t_3 stops before tunnel
4. m_1 sends again `train in tunnel.` to inform about second train (violation of protocol!)
5. m_1 sends `has train left tunnel?`
6. signal man m_2 sends `tunnel is free.`
7. m_1 believes that t_1 and t_2 have left tunnel and waves green flag
8. t_3 enters tunnel

t_2 has meanwhile decided to back out: 21 dead, 176 injured

The Clayton Tunnel Accident

One can almost hear the same comment being made time after time. "I could not imagine that this could ever happen." Yet bitter experience showed that it could, and gradually the regulations of railway engineering practice were elaborated.

(Nock 1976, railway historian)

The Needham-Schröder Protocol

task: authentication protocol

- agents (A)lice and (B)ob want to exchange common secret over insecure channel
- after protocol, both sides must be convinced about their partner's authentity
- no intruder may decrypt the secret

remark: common secret useful as session key. . .

assumption: perfect cryptography and uncompromised public keys

notation:

- $\langle M \rangle_C$ means that message M is encrypted with agent C's public key (only C can decrypt message)
- agent C can generate random number N_C (a nounce)

The Needham-Schröder Protocol

protocol:

1. A generates nounce N_A and sends message $\langle A, N_A \rangle_B$ to B
   "I'm Alice and this is my secret."
2. B generates nounce N_B and sends message $\langle N_A, N_B \rangle_A$ to A
   "I'm Bob, since I could decrypt your secret and here is mine."
3. A sends $\langle N_B \rangle_B$ to B "I'm really Alice, since I could decrypt your secret."

analysis:

- after step (2), A is convinced to deal with B and accepts $\langle N_A, N_B \rangle$ as common secret
- after step (3) B is also convinced to deal with A and accepts $\langle N_A, N_B \rangle$ as common secret

question: do you trust this protocol?

The Needham-Schröder Protocol

intruder I may

- intercept, read, store, answer messages
- take part in the protocol (with false identity)
- only use his private key for decryption

The Needham-Schröder Protocol

the attack:

1. A sends initial message $\langle A, N_A \rangle_I$ to I
   (I masquerading as B)
2. I masquerading as A sends message $\langle A, N_A \rangle_B$ to B
3. B sends message $\langle N_A, N_B \rangle_A$ to I
4. I forwards this message to A
5. A sends $\langle N_B \rangle_I$ to I
6. I sends $\langle N_B \rangle_B$ to B

analysis:

- after step (1), I knows A's secret
- after step (5), I knows also B's secret
- after step (6), B believes to deal with A

The Needham-Schröder Protocol

repair: send $\langle B, N_A, N_B \rangle_A$ as second message. . .

remark: flaw was detected 17 years after publication of protocol by model checking

- model agents in guarded command language with primitives for message channels and sending/receiving of messages
- model intruder as non-deterministically as possible
- model system properties in temporal logic:

$$G(\mathit{finished}(A) \wedge \mathit{finished}(B) \rightarrow \mathit{partnerA} = B \wedge \mathit{partnerB} = A)$$

$$G(\mathit{finished}(A) \wedge \mathit{partnerA} = B \rightarrow \neg \mathit{knowsnonceA}(I))$$

$$G(\mathit{finished}(B) \wedge \mathit{partnerB} = A \rightarrow \neg \mathit{knowsnonceB}(I))$$

- model checking explores state space and finds violating protocol run
- more about model checking later. . .

Mutual Exclusion

task: N processes access shared ressource "critical section", but only one at a time

question: what does correctness of distributed systems mean?

answer: for instance

- safety: never two or more processes in critical section (mutex-property)
- no deadlock: if several processes try to enter critical section,
  one must eventuall succeed
- no starvation: if a process tries to enter critical section,
  it will eventually succeed

Mutual Exclusion

instructions: (process may stop only in non-critical section)

```
loop
non_critical_section;
pre_protocol;
critical_section;
post_protocol;
end loop;
```

question: how to refine loop?

Mutual Exclusion

first attempt: (2 processes)

```
turn : integer range 1..2=1;
task body P1 is                          |  task body P2 is
begin                                    |  begin
  loop                                   |    loop
  non_critical_section_1;                |    non_critical_section_2;
  loop exit when turn=1; end loop;       |    loop exit when turn=2; end loop;
  critical_section_1;                    |    critical_section_2;
  turn:=2;                               |    turn:=1;
  end loop;                              |    end loop;
end P1;                                  |  end P2;
```

Mutual Exclusion

properties:

- the program has the mutex-property
  (when $P1$ and $P2$ are both in critical section and wlg $P1$ enters first, then first $turn = 1$ and then $turn = 2$. But $P1$ could not have reset it; a contradiction.)
- the program is deadlock-free
  (it is always the case that $turn = 1$ or $turn = 2$. So either $P1$ or $P2$ will make progress.)
- the program is starvation-free
  (no process can repeat loop infinitely often, without alternation, since $turn$ is always reset)

but: program can fail in absence of competition
($P2$ can die in non-critical section without ever setting $turn$ to 1.)

Mutual Exclusion

second attempt: (idea: use local variable of other process)

```
c1,c2 : integer range 0..1=1;
task body P1 is                          |  task body P2 is
begin                                    |  begin
  loop                                   |    loop
  non_critical_section_1;                |    non_critical_section_2;
  c1:=0;                                 |    c2:=0;
  loop exit when c2=1; end loop;         |    loop exit when c1=1; end loop;
  critical_section_1;                    |    critical_section_2;
  c1:=1;                                 |    c2:=1;
  end loop;                              |    end loop;
end P1;                                  |  end P2;
```

Mutual Exclusion

proposition: program has deadlock

proof: consider run

1. $(c1, c2) = (1, 1)$
2. $c1 := 0$ in $P1$
3. $c2 := 0$ in $P2$
4. $P1$ tests $c2$ without success
5. $P2$ tests $c1$ without success

Mutual Exclusion

third attempt: (resolution of deadlock)

```
c1,c2 : integer range 0..1=1;
task body P1 is                          |  task body P2 is
begin                                    |  begin
  loop                                   |    loop
  non_critical_section_1;                |    non_critical_section_2;
  c1:=0;                                 |    c2:=0;
  loop                                   |    loop
    exit when c2=1;                      |      exit when c1=1;
    c1:=1;                               |      c2:=1;
    c1:=0;                               |      c2:=0;
  end loop;                              |    end loop;
  critical_section_1;                    |    critical_section_2;
  c1:=1;                                 |    c2:=1;
  end loop;                              |    end loop;
end P1;                                  |  end P2;
```

Mutual Exclusion

proposition: program has starvation

proof: consider run

1. $(c1, c2) = (1, 1)$
2. $c1 := 0$ in $P1$
3. $c2 := 0$ in $P2$
4. $P2$ tests $c1$ and sets $c2 := 1$
5. $P1$ goes through cycle
   a) test $c2$
   b) go through critical section
   c) set $c1 := 1$
   d) go through non-critical section
   e) set $c1 := 1$
6. $P2$ sets $c2 := 0$
7. goto (4)

Mutual Exclusion

Dekker's algorithm: (analogous for $P2$)

```
c1,c2 : integer range 0..1=1; turn : integer 1..2=1;
task body P1 is
begin
  loop
  non_critical_section_1;
  c1:=0;
  loop
    exit when c2=1;
    if turn=2 then c1:=1; loop exit when turn=1; end loop; c1:=0;
    end if;
  end loop;
  critical_section_1;
  c1:=1; turn:=2;
  end loop;
end P1;
```

Mutual Exclusion

proposition: Dekker's algorithm has desirable correctness properties

observation: it is much more complex than initial attempt

question: how to prove correctness properties?

answer: using formal methods