Logic Model Checking

primary course website:
http://spinroot.com/cs118

TA: Mihai Florian, MC 305-16

course textbook:
The Spin Model Checker: Primer and Reference Manual

mflorian@cs.caltech.edu
gerard@spinroot.com
logic model checking
of multi-threaded software

• is it interesting? why?
  – does it touch on fundamental issues?
• is it relevant?
  – does it have any practical value?
• is it feasible?
  – is it a theoretical fad or the future?
“Observed ranges of defect removal efficiency for programming defect removal methods.”

<table>
<thead>
<tr>
<th>Removal Step</th>
<th>Lowest</th>
<th>Modal</th>
<th>Highest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Personal checking of design or docs</td>
<td>15</td>
<td>35</td>
<td>70</td>
</tr>
<tr>
<td>Informal group design reviews</td>
<td>30</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>Formal design inspections</td>
<td>35</td>
<td>55</td>
<td>75</td>
</tr>
<tr>
<td>Formal code inspections</td>
<td>30</td>
<td>60</td>
<td>70</td>
</tr>
<tr>
<td>Modeling or prototyping</td>
<td>35</td>
<td>65</td>
<td>80</td>
</tr>
<tr>
<td>Desk checking of code</td>
<td>20</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>Unit testing (single modules)</td>
<td>10</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>Function testing (related modules)</td>
<td>20</td>
<td>35</td>
<td>55</td>
</tr>
<tr>
<td>Integration testing (full system)</td>
<td>25</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>Field testing (live data)</td>
<td>35</td>
<td>50</td>
<td>65</td>
</tr>
<tr>
<td>Cumulative efficiency</td>
<td>93</td>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>

Capers Jones, *Programming Productivity*, 1986, Table 3-25, p. 179
some context

- requirements analysis
- high level design / system architecture
- detailed design / code structure
- coding
- testing
- operation

Phase where design decisions are made

Phase where design errors are found

The cost of fixing mistakes increases
the basic idea

- requirements analysis
- system engineering
- high level design
- system architecture
- detailed design
- code structure
- coding
- testing
- operation

build abstract design models
analyze them for potential violation of requirements
don’t proceed until the design is provably correct

exploit: automation, precise statements of design objectives, and the power of computers to perform in-depth analyses

reduce the reliance on testing as the primary means to catch errors ("defects")

finding errors earlier is generally smarter

reduce the reliance on testing as the primary means to catch errors ("defects")

exploit: automation, precise statements of design objectives, and the power of computers to perform in-depth analyses
what do real engineers do?

• characteristics of a mature engineering discipline:
  – distinguishes between requirements and specifications
  – uses engineering models to study designs
  – tries to predict the essential characteristics of a design before construction

• an engineering approach to safety-critical software design could look as follows:

  – requirements → logic
  – prototype → logic model
  – analysis → logic model checking
what does a software engineer do?

• instead of “model and analyze” programmers often rely on two other principles:
  – reuse
    • copy and adjust something that seemed to work before
  – trial and error
    • test and retest until it works

• after ~50 years, how well does this work?
  – testing often takes more than 50% of the development effort
  – major errors can and do slip through
  – classic software test methods were never meant to be applied to multi-threaded systems
example: the Microsoft Zune 30 GB

3. Wait until after noon GMT on January 1, 2009 (that’s 7 a.m. Eastern or 4 a.m. Pacific time).

My Zune 30 has been working fine today. Should I be worried?

Nope, your Zune is fine and will continue to work as long as you do not connect it to your computer before noon GMT on January 1, 2009 (7 a.m. Eastern or 4 a.m. Pacific time).

Note: If you connect your player to a computer before noon GMT on January 1, 2009, you’ll experience the freeze mentioned above— even if that computer does not have the Zune software installed. If this happens, follow the above steps.

What if I have rights-managed (DRM) content on my Zune?

Most likely, rights-managed content will not be affected by this issue. However, it’s a good idea to sync your Zune with your computer once the freeze has been resolved, just to make sure your usage rights are up to date.

What if I took advice from the forums and reset my Zune by disconnecting the battery?

This is a bad idea and we do not recommend opening your Zune by yourself (for one thing, doing so will void your warranty). However, if you’ve already opened it, do one of the following:

- Wait 24 hours from the time that you reset the Zune and then sync with your computer to refresh the usage rights; or
- Delete the player’s content using the Zune software (go to Settings, Device, Sync Options, Erase All Content); then re-sync it from your collection.
the code fragment

input : days elapsed since Jan 1, 1980
output: year + day of year

```c
year = 1980;
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    } else {
        days -= 365;
        year += 1;
    }
}
```

Q: December 31, 2008 was the 366th day of the year.
(2008 was a leap year: a multiple of 4, but not of 100 or 400).

Q: How many test-cases would the developer have needed to test this code?

In general, though, how many test-cases would a developer need to fully test:
• a square-root function
• a sorting routine
• a mutual exclusion algorithm
• a lock-free fifo queue algorithm
a large example of the effect of a small mistake...

a normal day in the life of a telephone system (Wednesday April 5, 1995)

~133 million long distance calls are handled
a not so ordinary day:
Monday January 15, 1990

1:23 pm
green - switch is up
white - switch is down
red - switch unreachable

1:43 pm
the day that “software glitch” became a mainstream term……

2 switches crash and reboot

requirement for telephone switches: 
≤ 15 seconds down-time per month
monday January 15, 1990 (MLK day)

(green: blocked long distance links)

50% of all calls were lost due to a small coding error (a redundant break statement). AT&T apologized by offering free long distance calls the next Valentine’s day: February 14, 1990
is this a new problem?

• a software crisis was first declared in 1968
  – complexity of software grew faster than our ability to control it
    • but a ‘large program’ then was only about 100K lines of code
    • today a large program is more like 100M lines of code
  – and most large software projects still work in crisis mode

• what did change
  – large programs are 1,000x larger
  – available memory is 100,000x larger
  – computers are 1,000,000x faster
  – but software is basically still developed and tested the same way as 30 years ago
  – we still cannot predictably produce reliable software

this is the single most important unsolved problem in computing science today
so it’s interesting – but is it feasible?

algorithmic improvements in automata theoretic software verification in the last few decades

Memory (Megabytes)

amount of memory needed to solve problem

amount of memory available to solve problem

a sample verification problem from 1980 tpc – a logic model of a telephone switch
all software analysis techniques have benefited from Moore's curve: because they are CPU intensive, they have tended to become twice as fast every 18 months
our ability to analyze software improves faster than software grows…

a rough indication
- **1968**: OS/360 = ~5 Million lines of assembly (~1 Million lines of C)
- **2003**: WindowsXP = ~64 Million Lines of C/C++ (35 years = 23x18 months)
  - INCREASE: ~$2^6$ (64 x)
  - MOORE’s CURVE: ~$2^{23}$ (>8 million x)

spacecraft software:
- **1968**: 30 lines of C equivalent
- **2003**: 550,000 lines of C
  - INCREASE: ~$2^{14}$ (18K x)

today
- **1 million** lines of C code can be compiled faster *and analyzed* far more thoroughly than **1 thousand** lines of C-equivalent code in 1968

model checking is an alternative engineering tool for verifying multi-threaded software designs – that similarly benefits from these trends
how does model checking work?  

a 2-viewgraph synopsis of the basic method

- **system:** $L(S)$ (the set of possible behaviors of $S$)
- **property:** $L(p)$ (the set of valid/desirable behaviors)
- **prove that:** $L(S) \subseteq L(p)$ (everything *possible* is also *valid*)
- **method:**
  
  to prove $L(S) \subseteq L(p)$

we can prove $L(S) \cap (\Sigma^\infty \setminus L(p)) = \emptyset$

which is the same as

$L(S) \cap L(\neg(p)) = \emptyset$
computing a language intersection

if intersection I is **empty** then  S satisfies  p
if intersection I is **non-empty** then it contains one or more counter-examples that show precisely *how* S can violate p
scope

- logic model checkers can catch a range of logic and functional design errors, especially errors related to concurrency and multi-threading
  - deadlock, livelock, starvation, tempo-blocking
  - race conditions, e.g. causing data corruption
  - locking problems, priority problems
  - resource allocation errors
  - reliance on relative speeds of execution of threads
  - violations of fixed system bounds (memory, stack, time)
  - incompleteness (unhandled event scenarios)
  - redundancy (dead code)
  - missing causal or temporal relations

using a model checker well requires:
- learning how to build logic models,
- learning how to use abstraction, and
- learning how to formalize requirements
zune resumed

```c
#define IsLeapYear(y) (((y%4 == 0) && (y%100 != 0)) || (y%400 == 0)

chan q = [0] of { short };

active proctype zune()
{
    short year = 1980;
    short days;

    do
        :: q?days ->
        S:
            do
                :: days > 365 ->
                    if
                        :: IsLeapYear(year)
                            :: days > 366 ->
                                days = days - 366;
                                year++
                            :: else /* do nothing */
                            fi
                        :: else ->
                            days = days - 365;
                            year++
                        fi
                :: else ->
                    break
            od;
    E: printf("Year: %d, Day %d\n", year, days)
    od
}
```

```c
init {
    /* jan 1, 2008 */
    if
        :: q!(days + 365)
        :: q!(days + 366)
        :: q!(days + 367)
    fi
}
```

```c
#define at_S zune@S
#define at_E zune@E
#include "zune.ltl"
```
zune resumed

```c
#define IsLeapYear(y) (((y%4 == 0) && (y%100 != 0)) || (y%400 == 0))

chan q = [0] of { short };

active proctype zune()
{
    short year = 1980;
    short days;

    do
        :: q?days ->
            :: days > 365 ->
                if
                    :: IsLeapYear(year) ->
                        if
                            :: days > 366 ->
                                days = days - 366;
                                year++
                            :: else /* do nothing */
                            fi
                        :: else ->
                            days = days - 365;
                            year++
                        fi
                    :: else ->
                        break
                    od;
            :: else ->
                break
            od;
        printf("Year: %d, Day %d\n", year, days)
    od
}
```

$ spin –f '!(at_S -> <> at_E)' > zune.ltl
$ spin –a zune.pml  # create model checker
$ cc –o pan pan.c   # compile
$ ./pan –a           # run (0.031 sec)
$ spin –t –p –l zune.pml  # replay error trace

spin: trail ends after 179 steps

<<<<<START OF CYCLE>>>>>
175:    proc  0 (zune) line  12 "zune.pml" (state 2)   [(days>365)]
177:    proc  0 (zune) line  14 "zune.pml" (state 3)   [IsLeapYear(year)]
179:    proc  0 (zune) line  21 "zune.pml" (state 7)   [else]
spin: trail ends after 179 steps
but, real concurrency problems are much more devious and harder to catch

Mars pathfinder mission from 1997:
first use of C and a commercial real-time operating system (VxWorks) by JPL (COTS components)

two typical software requirements:
• 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
mutual exclusion
processing loop

- idle
  - wait for lock
    - lock
    - run
      - access data bus
  - unlock
add a second (lower priority) process

H: high priority

L: low priority

L cannot run unless state is: i,-,-

but the executions can be interleaved in time, in the real system

but H can always run (it has priority)
a typical model checking query: when a process requests a resource, will it always eventually get it?

intersect possible behavior with invalid behavior

invalid endstate: red has priority, but blue holds the lock
the system deadlocks
happened on Mars in July 1997
why focus on spin?

• spin targets *software verification*, not hardware verification
• based on standard theory of $\omega$-automata and linear temporal logic
  – an example of the *automata theoretic* approach
• ACM Software Systems Award 2001 (winner 2002: Java)
• available freely and actively maintained
• large user-base (roughly equally in academia and in industry)
• used in mission-critical and safety critical software development
• annual Spin user workshops have been held since 1995
most of the theory was developed in the 60s and 70s but not fully exploited until 20-30 years later

1936: first theory on computability, e.g., Turing machines

1940-50: first computers

1955: early work on tense logics (predecessors of LTL)

1950-55: early work on software crisis

1960: early work on $\omega$-automata theory,

1968: two terms introduced: software crisis software engineering

1968: two terms introduced: $\omega$-regular properties

1970-75: key theoretical developments underlying Spin

1975: Edsger Dijkstra’s paper on Guarded Command Languages
1978: Tony Hoare’s paper on Communicating Sequential Processes
1976-1979: first experiments with reachability analyzers (e.g., Jan Hajek: ‘Approver’)

1977: Amir Pnueli introduces linear temporal logic for system verification
1980: earliest predecessor of Spin: ‘pan’ (Bell Labs)

1981: Ed Clarke and Allen Emerson introduce the term ‘model checking’ and the logic CTL*
1989: Spin first released publicly

1989: verification of class of $\omega$-regular properties

1993: BDDs and the SMV model checker (Ken McMillan, CMU)


1995: LTL conversion in Spin. (Doron Peled)

2001: support for embedded C code in Spin version 4.0
2003: breadth-first search mode added in Spin version 4.1
2007: multi-core algorithms added in Spin version 5.0

1986: Pierre Wolper and Moshe Vardi define the automata theoretic framework for LTL model checking

1986: Mazurkiewicz paper on trace theory

1986: Pierre Wolper and Moshe Vardi define the automata theoretic framework for LTL model checking

the two most influential logic model checking systems:

Spin: an explicit state LTL model checker based on automata theoretic verification method targeting software verification (asynchronous systems)

SMV: a symbolic CTL model checker targeting hardware circuit verification (synchronous systems)

(there are hundreds of other types of model checkers)
Course Outline

• **introduction**
  – what makes designing distributed systems hard?
  – the concept of a verification model
  – modeling concurrency, formalizing correctness claims
    • formalization of models with the Spin model checker
  – the role of abstraction

• **theoretical foundation**
  – omega automata and temporal logic
  – verification algorithms
  – search optimization

• **model checking in practice**
  – model building and model driven software verification
  – multi-core and swarm verification / challenges
Course Website

- urls:
  - http://spinroot.com/cs118
    - all lecture notes, examples, exercises, course assignments, background material, etc.

- focus:
  - automata-based, explicit-state logic model checking
  - on-the-fly verification
  - linear temporal logic (LTL)

- textbook:
  
  *The Spin Model Checker: Primer and Reference Manual*

- additional texts (optional):
  - see the website
“seek simplicity and distrust it.”
Alfred North Whitehead (1861-1947)