

Formal Methods in System Design

University of Utah

School of Computing

Ganesh Gopalakrishnan, Instructor

Greg Szubzda, half-TA

<http://www.eng.utah.edu/~cs6110>

Bugs

A bug is ugliness in the eyes of every beholder

(just thought of this)

Please try to come up with something better!

Bugs

A bug is something that hurts a significant number of stakeholders

Unfortunately, in today's digital world, there are many stakeholders

Bugs

The harm done due to bugs is relative

My Emacs copy has bugs. But I seldom run into it.

So I would not bother to look for them or fix them.

But if Emacs were used as an essential component in a piece of software involving the unsuspecting public, it certainly merits looking into it.

Bugs

...can be in the specs (false ad)

.. Or the implementation

... or in one's expectations! (requirements change; user expectations grow)

Best bug mitigation: Systematically reduce one's expectations!!

One of the main goals of this course

- Cultivate some awareness of what types of systems have what types of bugs
- Calibrate the severity, and the cost of fixing them
 - MOST BUGS ARE VERY EXPENSIVE TO LOCATE AND FIX
 - E.G. typing “l” when “j” was meant
- KNOW WHAT IS POSSIBLE TO DO (as a fix) if one had the resources
 - i.e. don’t remain clueless as to how to fix certain bugs
 - Of course there is a whole arsenal of testing methods that we won’t have time to look at
 - Experience shows that new methods based on “deeper” (formal) analysis holds considerable promise, given that conventional testing has not improved things for a while now...

Just for fun: Let's try to brainstorm and see where these systems may have bugs and what one may do about them...

- Modern multi-core CPUs
 - Pipeline stages
 - Floating point units
 - On-chip firmware
 - BIOS
- Compilers of various kinds
 - Embedded
 - Conventional
 - HPC
- Operating Systems
 - Small system OS
 - Enterprise OS
 - Real-time OS

Just for fun: Let's try to brainstorm and see where these systems may have bugs and what one may do about them...

- Libraries
 - Sequential libraries (string, etc)
 - Concurrency libraries and APIs and “languages”
 - MPI, OpenMP, Pthreads, CUDA
- High-end SIMD engines (CUDA etc)
- Web browsers
- Cloud

OK, you get the idea..

- It will be good to study a few concrete systems
- Understand a few modern debugging tools
- Apply them (during your projects) on example codes
- Or write new code and apply FV at the same time! (Best practice if possible)
- FV in HW and SW
- Why it is crucial to learn a ton about HW design (why it matters more for SW people also..)

Formal Methods

- Rigorous methods for correct design
- Vast topic (impromptu remarks by me for 15 mins, surveying FV from 1965 when Dijkstra started it all); maybe Turing did a bit of that too
- How it compares with Static Analysis
- How it differs from Testing
- How it ensures better quality of software/hardware
- How industries swear by it
- How compilers started using it (DFA, PDA), then how AI started the Logic agenda, then Temporal Logics, now back to SMT etc
- <http://www.cs.utah.edu/cav2011> and its workshops

Course Description

- Offered at Utah since 1986 (my first class!)
- It has kept with the trends
- This year
 - Emphasis is on Mathematical Logic a bit more
 - Want to tell you about SMT solvers
- Won't go from here to there in one leap

Plan

0. Functional programming in Ocaml (subliminally)
1. Murphi and explicit state enumeration
2. BDDs and symbolic search
3. First Order Logic and SMT
4. Applications of Logic/SMT
 - CUDA kernel analysis
 - Pipelined CPU analysis
 - Software testing (C testing)

Plan

1. Why Murphi and explicit state enumeration

- Teaches you the simplest FV method using the simplest tool available
- Yet a high-impact tool, considering how little “sales” has gone into it
- Learn about assertion writing, invariants, transition systems, fixed-points, quantification, parameterized verification, symmetry
- Specify protocols, debug, watch error traces, learn about your model
- Use as an every-day tool (if in doubt, Murphi it!)
 - As easy as breathing after a while!

Fantastic projects in the offing:

- . Build a multi-core Murphi
- . Build SMT-assist to analyze Murphi guards (who knows what you’ll find ?!)
- . Can publish these!

While you are working on Murphi, you’ll be given challenges from Jason Hickey’s
Ocaml book

Knowing Functional Programming helps you immensely in so many ways (conceptual,
practical)

Plan

2. Why BDDs and symbolic methods

- Simplest reinforcement of minimal DFAs
- Simplest way to understand fixed-points
- Can code-up your own BDD package in Ocaml
- Can do Tic-Tac-Toe (that you did in Murphi) again in BDDs
- Recall Logic and Boolean reasoning
- Understand what made Formal Verification a success, and led to Model Checking
- Nice prelude to First Order Logic

Plan

3. Why First-Order Logic

- See <http://research.microsoft.com/en-us/um/redmond/projects/z3/SMT@MS.pdf>
- <http://research.microsoft.com/en-us/um/redmond/projects/z3/>
- http://research.microsoft.com/en-us/um/redmond/projects/z3/SMT@MS_linz2008.pdf
- This is the first time that “thought” and “code logic” is being sliced and diced, and analyzed with acceptable efficiency (in many cases) using mechanical methods
 - Much like numerical analysts have sliced and diced real world data, force fields, etc., and analyzed them using Linear Algebra packages and PDE solvers
 - Automatic test generation, compilation, system analysis
 - Program testing, figuring out browser vulnerabilities, etc.

Plan

4. Applications

- Making it fun using topics such as CUDA analysis
- Multicore versions of existing sequential code (e.g. make Murphi multicore)
- Have Eddy Murphi – can make it more parallel + distributed
 - Right now, it uses threading in a fixed way
- Pipelined CPU design – nice use of symbolic reasoning
- Other projects welcome

Verification approaches

Verify models or code?

- It is good to build models and verify the models whenever that is possible
 - In rare instances, verifying the real artifacts can be extremely difficult – e.g. hardware realizations of cache coherence protocols – so much so that these realizations are not directly verified as a whole
 - Often, the code is available (or only the code is available; nobody bothers to build models; or the distance between the model and code is so huge; or the code has evolved to outmode the design model)
- For tricky concurrency protocols, do build models
- For computation oriented situations, often building the actual code and verifying it using dynamic / symbolic methods is the most practical approach
 - Verification budgets are also NOT unlimited!

State transition tables/languages

- State tables are the way many models are written
 - “If in state, upon condition, go to this new state”
 - Good historic example: NASA autonomous navigation backpack modeled using SCR table notations embedded in PVS
- Murphi is a language based approach for state transition tables
- The language is unusually refreshing and well done (evolved based on domain needs)

Murphi based explicit state verification

Simple “Unity-style” language

Example : Let’s code this up...

```

                flag[0] = 0;          flag[1] = 0;
                turn;
P0: flag[0] = 1;          P1: flag[1] = 1;
   turn = 1;             turn = 0;
while (flag[1] == 1 && turn == 1) ;   while (flag[0] == 1 && turn == 0) ;

// critical section          // critical section ... ..
// end of critical section   // end of critical section

flag[0] = 0;                flag[1] = 0;
```

...in Murphi (from the distribution)

```
-- Engineer: C. Norris Ip
```

```
Const
```

```
  N: 2; -- and two only
```

```
Type
```

```
  pid: Scalarset(N);
```

```
  label_t: Enum{L0, -- : non critical section; Q1 := true;
```

```
    L1, -- : turn := 1;
```

```
    L2, -- : wait until not Q2 or turn = 2
```

```
    L3, -- : critical section
```

```
    L4 -- : Q1:= false;
```

```
  };
```

...in Murphi (from the distribution)

Var

P: Array [pid] Of label_t;

Q: Array [pid] Of boolean;

turn: pid;

...in Murphi (from the distribution)

Ruleset i: pid Do

Rule "execute assign Qi true"

P[i] = L0 ==>

Begin

Q[i] := true;

P[i] := L1;

End;

Rule "execute assign turn i"

P[i] = L1 ==>

Begin

turn := i;

P[i] := L2;

End;

...in Murphi (from the distribution)

```
Ruleset j: pid Do
  Rule "execute wait until"
    P[i] = L2
    & j != i ==>
  Begin
    If ( !Q[j]
      | turn = j )
    Then
      P[i] := L3;
    End; --If
  End;
End; --Ruleset
```

...in Murphi (from the distribution)

Rule "execute critical section"

$P[i] = L3 \implies$

Begin

$P[i] := L4;$

End;

Rule "execute assign Qi false"

$P[i] = L4 \implies$

Begin

$Q[i] := \text{false};$

$P[i] := L0;$

End;

...in Murphi (from the distribution)

```
Startstate
Begin
  For k:pid Do
    P[k] := L0;
    Q[k] := false;
  End; --For
  turn := i;
End;

End; --Ruleset
```

...in Murphi (from the distribution)

Invariant "mutual exclusion"

```
! Exists i1: pid Do
  Exists i2: pid Do
    ( i1 != i2
      & P[i1] = L3 -- critical
      & P[i2] = L3 -- critical
    )
  End --exists
End; --Exists
```

Work for you!

- Download and install Murphi
- Try out the binary search and its variants
 - Try the Jon Bentley variant (that 90% of programmers got wrong) where low or high were not updated as “mid+1” or “mid-1” respectively)
 - What is this bug really?
- Get ready for some Murphi exercises to be assigned this Thursday
- Get going on learning Ocaml