

# Formal Methods in System Design, Lec 3

University of Utah

School of Computing

Ganesh Gopalakrishnan, Instructor

# Why and how of formal methods

- Some requests
- Please use the Google groups actively
- Could one person please serve as ‘scribe’ for the important points covered in each class?
- The Scribe will get brownie (participation) points, and my gratitude
- The Scribe may also please discuss with others and find out what is unclear, and let me know!

# Why and how of formal methods

- Testing is inadequate, and model checking helps
  - Model checking has found bugs in silicon that HW execution did not run into
  - Helps you build/refine models
  - Skill set : practice , practice !
- Other ways of FV
  - Theorem proving
  - Symbolic execution
- Code level artifacts verified by numerous formally assisted testing methods
- Formal Methods are changing how software is being designed / deployed (already so in HW design)
- Bug hunting using formal methods is one of the crowning achievements of CS
- Read a recent CACM on Xavier Leroy's formal verification of a C-subset compiler compiling to PowerPC assembly code

# Why and how of formal methods

- We will study model checking for SAFETY and LIVENESS properties, using Murphi and SPIN respectively
- SAFETY
  - Bad things will not happen
  - Invariance is largely what we will deal with
  - Violations of Safety : Finite evidence will suffice
- LIVENESS
  - Good things will eventually happen
  - Violations of Liveness : have to present infinite executions where the claimed good thing is never found to happen!
  - Will need automata that can talk about the acceptance or non-acceptance of infinite runs (of cyclic runs)

# Sequential versus Concurrent

- We will deal with sequential and concurrent programming situations
- The idea of invariants is common to both
- Usually the “invariants” of interest for sequential programs are found “when the program finishes”
- Still a “sequential program” also maintains invariants (consistency of its data structures)

# Finite state transition systems

- Readings :
  - Norris Ip's dissertation
  - Ulrich Stern's dissertation
  - My book's chapter on Model Checking
- Supplementary readings
  - Clarke, Grumberg, Peled, "Model checking"
  - Baier and Katoen, "Principles of Model Checking"
- We will learn Boolean stuff and quantification right here – will be handy when we go to FOL
- Duality of "forall" and "exists" also exists in temporal quantifiers – hence a 'light weight' intro to FOL right here

# Finite state transition system

- Also known as FSTS, “Automata”, Kripke Structures, STS, TS, ...
- Finite number of States  $S$
- A transition relation over  $S$  (binary reachability relation, i.e. a “state graph”)
- One or more initial states

# Finite state transition system

- Example:
- Var a,b : 0..3;
- Start
  - a := 2;
  - b := 2;
- Rules
  - R1:  $a \geq b \rightarrow a := (2a+1) \% 4$ ;
  - R2:  $a \leq b \rightarrow b := 2(a+b) \% 4$ ;
  - R3:  $\text{odd}(a+b) \rightarrow a := (a+1) \% 3$ ;
- Reachable state space ??
- Invariants??

# Finite state transition system

- Draw reachable state space in class (diagram here)
- An invariant is an assertion (Boolean formula or Boolean expression) true in every state
- Come up with a catalog of invariants (to be filled)
- I1 :
- I2:
- I3:
- I4:
- I5:
- I6:
- I7:

# Strong and Weak assertions

- Which is stronger? Let P be “prime” and Q be “odd”
  - P
  - P & Q
  - P | Q
- How do stronger statements relate to weak statements as far as their characteristic sets go?
  - Each assertion is completely equivalent (in purpose) to the characteristic set
  - $A \Rightarrow B$  means  $\text{chset}(A)$  contained in  $\text{chset}(B)$
  - False has a chset of empty-set
  - False implies anything because the empty-set is contained in every other set!
- Readings: My book sections, and Bradley/Manna sections
- You must start practicing Logical assertions
- There will be a HW on them this week

# Strong and Weak Invariants

- You can write many “weak” invariants
- They don’t help you prove much
- Even type declarations are invariants
  - They help bound the ranges of variables
- The LIFE BLOOD OF Object Oriented Programming is the Class Invariants of the various classes you design
  - When an OO Program Crashes, it is usually because of broken class invariants
- The STRONGEST INVARIANT (sin) of a transition system is exactly its reachable set of states
- This is too much to ask for!
  - You need to know exactly how the program computes, exactly how many times loops iterate, etc.
  - So usually we “smoothen” the invariants and try to “get away” with weaker invariants (not as weak as “true” or “type invariants” but still plenty good to prove what we want)

# Invariants, Inductive Invariants, Why?

- An Inductive Invariant sort of “forgets the context”
- It supports flow insensitive reasoning
- An invariant  $I$  is inductive if
  - The initial state of a TS satisfies it
  - If ANY STATE  $s$  (not necessarily a reachable state) satisfies it, then each rule enabled at  $s$  preserves it!
- Which of the invariants you saw earlier are inductive and which are not?

# Invariants, Inductive Invariants, Why?

- Another example (from the web.. Will credit source soon)
- Let  $x$  be a Real number
- Consider the transition system where
- Init or Startstate says
  - $x := 2;$
- The only state transition:
  - $x := 2x - 1;$
- Is  $x > 0$  an invariant?
- Is it inductive?
- Is  $x > 1$  an invariant?
- Is it inductive?

# Invariants, Inductive Invariants, Why?

- Again, invariants define the safe operating range of a system
- Coolest presentation:
  - Vaughan Pratt's analysis of the Pentium FDIV bug
  - Describes the SRT algorithm as a walk of a robot Robbie along the diagonals of a table
  - Oops, Robbie trips outside the allowed limits
  - That was the Pentium bug!
- Invariant == safe operating zone – that simple and fundamental !!!!!

# Invariants, Inductive Invariants, Why?

- Again, invariants define the safe operating range of a system
- Coolest presentation:
  - Vaughan Pratt's analysis of the Pentium FDIV bug
  - Describes the SRT algorithm as a walk of a robot Robbie along the diagonals of a table
  - Oops, Robbie trips outside the allowed limits
  - That was the Pentium bug!
- Invariant == safe operating zone – that simple and fundamental !!!!!

# Model checkers compute 'sin' for you!

- Model checkers are sinners
- They compute for you 'sin' always! So neat
- Sinning is good!
- But sinning is hard 😊 - eh?
- So people usually discover other invariants
- They usually “muck around” the rules they have
- If they have a program, they first try to milk the invariants found in the type declaration
- Then they chop the program, throw the statements into a bucket (flow insensitive) and shake the bucket till invariants rise to the top (like butter)

# What to do with an inductive inv?

- Say you found an inductive invariant  $\text{IndInv}$
- Then you would have shown
  - $\text{Init} \models \text{IndInv}$  meaning the initial state satisfies it
  - For all  $s : \text{IndInv}(s) \Rightarrow \text{IndInv}(\text{tau}(s))$  where  $\text{tau}$  is any “step” taken by the program
  - Then if we can show that  $\text{IndInv} \Rightarrow \text{Prop}$  where  $\text{Prop}$  is a property of interest, then we have shown that  $\text{Prop}$  is true also of ‘ $\text{sin}$ ’
  - This is because  $\text{sin} \Rightarrow \text{IndInv}$  , or for that matter any other invariant

# Invariant of BinSrch

- I showed you the invariant of BinSrch once it finishes its execution
- I wrote the invariant as
  - done & status -> ...
  - &
  - done & !status -> ...
- This was my way to “cheat” and make the “exit assertion” apply everywhere in the program (at those points in the program, “done” will be false and hence the invariant holds)
- If you “back off one step” and sit inside the BinSrch loop, then you get an inductive invariant for the loop!
- This is the BEST way to study Floyd/Hoare logic , or Hoare logic
- You can read more in Aaron Bradley / Manna, and in Gordon’s book
- Again this is a class where I am going to pull in from 1000s of places so you get the benefit of taking 100 ‘pedantic’ classes in one class... trying to 😊

# Invariant of BinSrch

- OK, let us discover the “loop invariant” for Binsrch
  - Write it here:
- This inductive invariant does not “know” whether the loop is gonna terminate
- Again showing the power of flow insensitive reasoning

# Now onto Murphi

- Your questions
  - About symmetry reasoning
  - About scalar sets
  - OK I have the right example for you

# About Asg1

- I'll try to have staged submissions for every assignment
- For initial submissions, I'll just make remarks in class or in the Google group
- For the final submission, please do include a typescript session of your run results
- We will not be able to re-run your code often.. (but we might!) – so your points depends on a nice 1-2 page writeup you write describing your solution and what you learnt
- Write, write, and write again – that is your career success story
- Submit a README, your program files, a typescript kind of output, and a writeup (PDF or Word) describing what you learned

break

# Invariant preservation based modeling

- We will study how to develop a model beautifully
- We will use model checkers as game solvers
- That is what they really are!

# Invariant preservation based modeling

- Boat, wolf, goat, cabbage
- Man with boat always, so model man as boat
- The usual rules
- What is a safe state?
- Can you design rules that preserve safety?
- Can a model checker compute for you the solution?
- Will be part of Asg2 – but due next week Tue (other parts of Asg2 will be due a week from next Tue)
- So Asg2 will be : mwgc, plus Lamport's Bakery Protocol!

# On the infinite coolness of studying locking protocols

- We will study material from The Art of Multiprocessor Programming
- Luckily, material available on the web
  - <http://www.cs.tau.ac.il/~shanir/multiprocessor-synch-2003/mutex/notes/mutex2.pdf>
- Their book is worth owning

# On the infinite coolness of studying locking protocols

- First study a mutex protocol that will deadlock upon interleaving
- Then study a mutex protocol that will deadlock when there is NO interleaving!
- Peterson's brilliant solution: combine the ideas in these two!
- Peterson's N process solution
- Now study paper/pencil proofs vs. Model Checking proofs

# Lamport's Bakery Protocol

- Only known (to me) mutual exclusion protocol which has two neat properties
  - Crash of a node does not hang the protocol
  - If a read/write overlap returns garbage for Read, then too no sweat!
  - Read Lamport's CACM paper + follow-ups from Leslie Lamport's webpage
  - You will model these protocols plus also follow Lamport's paper/pencil proofs

# Peterson, Lamport Bakery, Memory Consistency! The beautiful story!

- Peterson fails under TSO
- What about Lamport's?
  - Has property that there is no location with concurrent accesses!