Automated Software Validation Week 1: Introduction

Abhik Roychoudhury
Department of Computer Science
National University of Singapore

What is it about ?

- Techniques to help reliable software development.
- Checking program behavior
  - Typically checking whether desired invariants hold at program control points.
- What is the programming language ?
  - Conventional languages like C/Java
  - Deeper issues remain …

What kind of programs ?

- Conventional sequential programs
  - C like programs
- Multi-threaded software for distributed sys.
  - E.g. Multi-threaded Java
  - Many behaviors due to thread interleaving
- Reactive software
  - In continuous interaction with environment
  - e.g. control software in embedded sys.

Conventional development

- Collect software requirements
  - Programmers often do not collect complete sets of requirements.
- Write code
  - Good programming disciplines exist e.g. modular development
- Debug
  - Code walkthrough, Peer review, Testing
  - Again informal and/or incomplete.

So are we …

… going to look at program debugging ?

- YES
  - All our validation techniques can be used for software debugging
- NO
  - We will not only look at conventional software engineering activities like testing.

Why bother ?

- Testing etc. is incomplete.
  - Checking program behavior for a specific execution
  - No guarantees about program behavior
    - safety critical systems
    - Brake controller software of your car
  - Substantial effort spent anyway in generating "good" test cases, ensuring "good" coverage.

Spectrum of Techniques

- Static checking techniques *(focus of this course)*
  - Model Checking
  - Deductive proof techniques (e.g. Induction)
- Dynamic checking techniques
  - Monitoring, Invariant Detection
- Conventional debugging
  - Testing, Slicing (how to link with validation techniques)
  - Fault Localization

Static Checking

- Analyze program source code to establish invariants at control locations
  - Automated techniques
  - Deductive techniques
- Deductive techniques similar to constructing a proof of correctness by hand.
  - Involves guessing and proving loop invariants for loops in the program
  - Proof Assistants available to help mechanization.

Differences via Example

- `for (i = 1; i< 10; i++) {}`
- How to prove i > 0 always ?
  - Model checking
    - Generate a transition system whose states are
      - (Control Loc, Value of i)
    - Traverse the transition sys. to verify that i > 0 in all reachable states of the transition system.

Differences via Example

- `for (i = 1; i< 10; i++) {}`
- How to prove i > 0 always ?
  - Theorem Proving
    - Prove by induction on the iterations of the loop.
  - Static Analysis
    - Infer possible values of i at each control location (irrespective of how they are reached).
    - Check that all possible values are > 0

Automated Static Checking

- Difficulties in automation
  - Reasoning about infinite domains and structures in the memory store of the program
  - Reasoning about aliases in the memory store
    - Array indices
    - Pointers
- How to surmount these problems ?
  - Abstract the memory store (to a finite structure ?)

Model Checking

- Abstraction is designed for a specific program.
- Used for checking complex temporal properties (safety, liveness, response properties).
- User may have to dabble in constructing abstract model, in general.
  - Canonical abstractions (data abs.) available.
- Search based exact procedure at a certain level of abstraction
  - Provides detailed counter-example evidence.

Model Checking

- Inputs:
  - **finite state** transition system (implementation)
  - Temporal logic formula (specification language)
- Output:
  - True if the specification holds
  - A **counterexample behavior** if it does not
- Technique:
  - Implementation FSM is a finite graph.
  - Unfold and search this finite graph to check all behaviors.

Use of Model Checking

- Generate finite-state transition system like models from C/Java code
- Employ search on this model to verify invariants or other properties.
- If counter-example obtained by MC
  - Need to locate the bug from counterexample

An Example

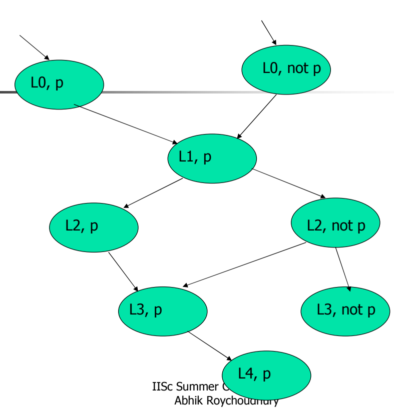
- x = 0; x = x + 1; x = x + 1;
- if (x > 2){ error }
- Is the error reachable ?
- Problem: domain of x is not finite

Step 1: Label the locations

- L0: x = 0;
- L1: x = x + 1;
- L2: x = x + 1;
- L3: if x > 2
- L4: error

Step 2: Abstract x

- The finite state transition system generated for the abstraction $\{x > 2\}$ is constructed. Use shorthand $p \equiv x > 2$. This finite state transition system shows the reachability of location L4.
- Do this now
  - How did we get x > 2 ??



Step 3: Construct TS & check

- We find 1 or more counter-examples
- Use them to refine abstraction

- Example:
  - $(L0, p), (L1, \neg p), (L2, p), (L3, p), (L4, p)$

Step 4: Refine abstraction & repeat

- How to analyze
  - $(L0, p), (L1, \neg p), (L2, p), (L3, p), (L4, p)$
  - To get refinement of our current abstraction
    - { x > 2}
- In this case,
  - {x = 0, x = 1, x =2, x > 2}
    - Is clearly sufficient …
- We need to turn this black art into science!

Parallel use of Model Checking

- Requirements → Code
- If the Req. are formal and complete (at a certain level of abstraction)
  - The requirements form a model.
  - We can search and validate this model.
- Note that:
  - Making the Req. → Code translation formal and/or automated is difficult.
  - Topic of research projects in certain domains.

Dynamic Checking

- Monitoring amounts to run-time checks during program execution.
  - Testing checks program traces during program development, not at run-time.
- Other run-time techniques try to infer bugs by detecting a deviation from "normal" behavior as a potential bug.
  - Needs to be confirmed by user.
  - Constructing program model based on observable traces.

Debugging via Slicing

- Slicing
  - Input: A var. V at control location L
  - Output: Part of the program code which affects the value of V at location L
- Can be static or dynamic
  - Static: Part of code which affects V at L for some exec
  - Dynamic: …. for a particular exec
- Give explanations of problematic executions (which are detected by validation techniques)

End Goal of the course

- Familiarity with host of debugging/ verification techniques beyond testing.
- Techniques help locate hard-to-detect bugs.
- Focus is on bug hunting (pragmatic) rather than proving systems correct (quest of a theoretician).

Assessment + Materials

- Term Paper : 20 %
- Assignments (3): 30%
- Final Exam : 50 %

- Course Web-page
  - http://drona.csa.iisc.ernet.in/~abhik
    - Lectures and readings available here.

Assignment 1 (SMV)

- We will deal with modeling and verification of low-level protocols e.g. bus protocols, cache coherence protocols.
- Part (A)
  - AMBA System on Chip Bus protocol deployed in ARM processors

Assignment 1 (SMV)

- The original AMBA AHB document runs to 60 pages
  - Simplify and model it in SMV.
  - I will provide you with a small model showing a starvation error since this is the first assignment's first part.

- Part (B) Wildfire Verification Challenge Problem --- Try to injected bugs in a cache coherence protocol by modeling the core in SMV.

Assignment 2 (SPIN)

- Find injected bugs in a simple transmission protocol from Holzmann's SPIN book – Warmup Exercise
- Model a real-life air traffic control system (developed and deployed by NASA in large airports like Dallas Fort Worth)

Assignment 2 (SPIN)

- You will deal with a portion of the controller --- dealing with weathe updates.
- I will provide you with a req. documents and you will have to formalize, model and find bugs from there.
  - Real bugs!!

Assignment 3 (PVS)

- Theorem Proving

Term Paper

- Java Memory Model
  - New semantics of multi-threaded Java
  - Investigate building a memory model sensitive model checker for Java programs
  - Will involve formalizing the virtual machine or at least portions of it
    - Bytecode level checker.
  - I will get you started by describing a similar exercise in C# -- paper in FM 2006.

Sample Overview Readings

- *Software Analysis and Model Checking,* Gerard Holzmann, 2002.
- *Verification of Embedded Software: Problems and Perspectives*, Patrick and Radhia Cousot, 2001.
- *Automatically validating temporal safety properties of interfaces*, Thomas Ball and Sriram K. Rajamani, 2001
- *Trends in Software Verification*, Gerard Holzmann, 2003.

Dates, times

- Lecture:
  - Friday 10:00 – 12:00 noon
  - Thursday 2:00 – 4:00 PM
- Consultation
  - By e-mail appointment only
    - abhik@csa.iisc.ernet.in
- Schedule (Exam, assignments)
  - See course webpage for updates.
  - Final planned in the week of 25 – 29 June.
- Any administrative questions ?

Course Outline

- Introduction
- Property specification and checking
  - SMV, SPIN model checker
- Software Abstractions
  - Predicate abstractions and refinement
- Theorem Proving & Deduction
  - PVS theorem prover
- Software testing and debugging
  - Testing, Slicing, Dynamic Analysis.

To start with

- What kind of models?
  - Transition Systems
  - Extracted from a spec. in a modeling language
- Modeling language
  - Same as prog. lang but make state space finite.
  - Avoid heaps, dynamic allocation.
- How do they relate to code?
  - Translate prog. lang. to modeling language
  - Or generate specification in modeling language with program fragments embedded.

Transition Systems as Models

Abhik Roychoudhury
School of Computing, NUS

The purpose

- A transition system is supposed to serve as a model of system behavior.
  - Model is required for analysis and verification of program/system.
  - Once the model is derived, our technique focuses on space and time efficient search of the model (for verification).

What sort of a model ?

- Will capture evolution of the program/ circuit with the passage of time.
- Will contain information about internal values which are essential for establishing correctness
  - We consider functional correctness
- Will leave out low-level details e.g. the data values exchanged in a communication protocol

Transition system

- M = (S, S0, R)
  - S = Set of states (may not be finite)
  - S0 = Set of initial states
  - $R \subseteq S \times S$ is the transition relation
- Model M can then be subjected to verification.
- Need to associate states and transitions with the text of a program.

Why states/transitions are important ?

- We are looking at reactive systems
  - Never-ending evolution over time.
- Snapshots of evolution captured via states
  - Stop the evolution at any time and peek into it.
- One atomic step of evolution captured by transition

The association

- Associating states/transitions with pgm.
  - Program Variables
    - Program Counter (pc), Data variables
  - Program State
    - Valuation of program variables
  - Transition
    - Moving from one state to another by executing a program statement.

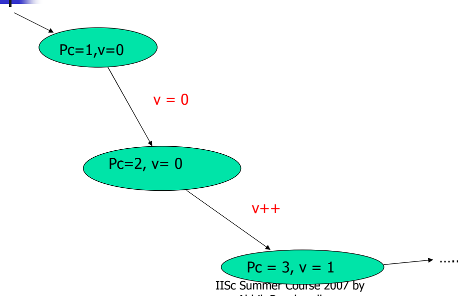
Example

- 1 v = 0;
- 2 v++;
- 3 …
  - What are the states ?
    - (value of pc, value of v)
  - How many initial states are there ?
    - No info, depends on the type of v

Example - Continued

- Assuming that the value of v before line 1 is executed is 0, draw the states and transitions corresponding to this program.

Example - Continued



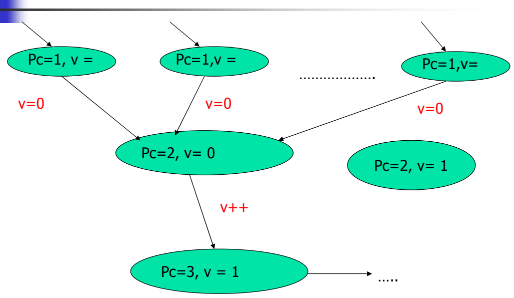
Too many states ?

- Defined by possible variable valuations
  - Determined by variable types
    - integer, float … : this a problem !
  - State space explosion is the central problem in model checking.
  - Not all of the states might however not be reachable from the initial state(s)
    - Do not appear in the model (i.e. if we were to draw it !)

Example

- 1 v = 0;
- 2 v++;
- 3 …
  - The states (pc = 2, v = 1), (pc =2, v = 2) etc are not reachable from the initial states.
  - This is true irrespective of the initial value of v at line 1.

Example - continued



Propositions

- We are trying to generate a high-level model of system behavior for validation
- With each state, we capture the truth/falsehood w.r.t. atomic propositions
  - [v == 0 ?]
  - [u == v ?]
- Where do we get these propositions ?
  - Depends on what properties we want to verify !
  - Related to variables in the program being verified.
  - We assume the set of all AP is given.

Why Propositions?

- We cannot/need not model everything!
  - Every control location
  - Value of every system variable …
- So, propositions denote the relationships among system variables that we are tracking in our model
  - B, where B is a boolean variable
  - U == 0
  - U == V
  - The language of the expressions that we can track depends on the language in which we describe our designs.

Why Propositions?

- In the case of hardware circuits/low-level protocols
  - Each signal can denote a proposition.
  - Certain signals need not be modeled
    - e.g. the bits on data bus, address bus
  - Certain signals modeled for studying functional correctness
    - e.g. control signals
- For programs
  - Conditions on var. values [v==0], [x> y]

Kripke Structures

- An unifying formalism based on transition systems
  - State transition graph (S, S0, R, L)
    - S is a (finite) set of states
    - $S0 \subseteq S$ is the set of initial states
    - $R \subseteq S \times S$ is the state transition relation such that every state has at least one successor
    - And …

Kripke Structures

  - $L : S \rightarrow 2^{AP}$ is a labeling function which assigns a set of atomic propositions to a state
    - L allows us to describe the truth/falsehood of a proposition in the various system states.
    - The propositions refer to valuations of state variables.
- Kripke structures are powerful enough to model behaviors of
  - sequential as well as concurrent systems,
  - software as well as hardware.

Obtaining Kripke Structures

- Obtaining Kripke Structure from a concurrent program directly is laborious.
- A model checking tool allows you to input the program in its modeling language, and then it extracts the Kripke Structure.
- You model the sequential threads separately, and specify a model of concurrency
  - e.g. asynchronous with shared variable communication

Obtaining Kripke Structures

- We can represent the states/transitions of a sequential/concurrent program via a standard logical language
  - First-order logic
  - Standard mechanism of extracting Kripke Structure from first-order representation
  - Still we need to define the transitions corresponding to every control construct in the programming language ---- laborious

Sequential program - Example

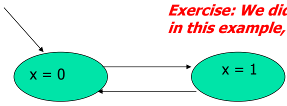
Initially x is 0

while true do

x := 1 – x;

endwhile

Exercise: We did not model the program counter in this example, Can you try to do so?



Concurrent Program

- A set of programs running independently, communicating from time to time, thereby performing a common task.
- Flavors of Concurrency
  - Synchronous execution
  - Asynchronous/interleaved execution
    - Communication via shared variables
    - Message passing communication

Concurrent Program - Example

P0 || P1

- l0: while true do
- l1: wait(turn = 0);
- l2: turn := 1;
- l3: endwhile

- m0: while true do
- m1: wait(turn = 1);
- m2: turn := 0;
- m3: endwhile

Models a crude protocol for entry/exit to critical section without modeling the critical section itself.

States

- Global State = (pc0, pc1, turn)
  - pc0 $\in$ { l0, l1, l2, l3 }
  - pc1 $\in$ { m0, m1, m2, m3 }
  - turn $\in$ { 0, 1 }
- Total = 4 * 4 * 2 = 32 possible states
  - Not all of them might be reachable from the initial states.

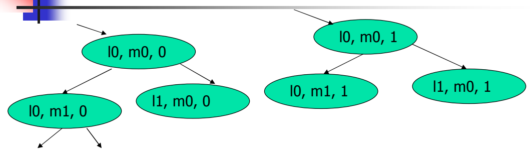
Propositions

- Define propositions to capture information about variable valuations. Possible propositions in our ex.
  - pc0 = l0, pc0 = l1, pc0 = l2, pc0 = l3
  - pc1 = m0, pc1 = m1, pc1 = m2, pc1 = m3
  - turn = 0, turn = 1
- Clearly the proposition pc0 = l0 is true in any state
  - (pc0 = l0, pc1 = ?, turn = ??)
- Clarifies the labeling function L in Kripke Structures.

System Properties

- Propositions can be used to define interesting properties
  - It is never the case that pc0 = l2 and pc1 = m2
- The above property defines mutually exclusive access to the critical section.
- We will study a logic for describing such properties in the next class.

State Transition Graph



There are two initial states.

Finite number of possible states, so the construction (of the Kripke Structure) must terminate.

Verification of the mutual exclusion property can be achieved by inspection (of the constructed Kripke Structure) in this simple example.

Control and data variables

- State = valuation of control and data vars.
- In our example
  - *pc0, pc1* are control variables.
  - *turn* is a shared data variable.
- To generate a finite state transition system
  - Data variables must have finite types, and
  - Finitely many control locations

Data variables

- Data variables often do not have finite types
  - integer, …
  - Usually abstracted into a finite type.
  - An integer variable can be abstracted to {-,0,+}
    - Just store the information about the sign of the variable.
  - ***Caution: Coming up with these abstractions is a whole new problem which we will discuss later in this course.***

Control Locations

- # of control locations of a program is always finite ?
  - NO, because your program may be a concurrent program with unboundedly many processes or threads (parameterized system).
  - Can employ control abstractions
    - Unbounded # of processes with same behavior (captured as FSM)
    - Abstract the count of processes in each state of the FSM
    - Example: Cache Coherence Protocols, Distributed Controllers.
    - We will not consider these abstractions in this course.

Using Kripke Structures

- Let us revise and recapitulate
  - the use of Kripke structures for modeling behaviors of
    - asynchronous concurrent systems
      - Multi-threaded programs/protocols
      - Model each thread as a Kripke Structure and then compose these Kripke Structures.
- Synchronous concurrent Systems can also modeled using Kripke Structures.
    - Hardware circuits, Bus Protocols

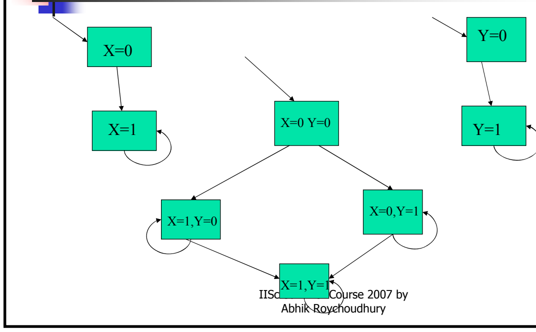
What is system behavior ?

- What do we mean when we say Kripke structures model "system behavior" ?
  - For now, consider the infinite traces of the Kripke structure as representing behavior.
    - We consider Kripke structures where each state has at least one successor state.
    - Represents an ongoing evolution of states in a reactive system.
    - Non-determinism in evolution due to concurrency.

Behaviors

- What are the behaviors of the asynch. concurrent system ?
  - The sequential programs P_i need not be terminating
  - Behaviors (Traces) of the concurrent program formed by interleaving transitions of programs P_i
- Simple Example
  - Initially x is 0, y is 0
  - do{ x := 1} forever || do { y := 1 } forever

Interleavings



Fairness

- P0 manipulates X
- P1 manipulates Y
- In the global state <X=0, Y=1>
  - P0 or P1 could make a move.
  - We allow the behavior that P1 always makes a move (self-loop)
  - System is stuck at <X= 0, Y=1>
  - Unfair execution !

Fair Kripke Structures

- M = (S, R, L, F)
  - S, R, L as before.
  - $F \subseteq 2^S$ is a set of fairness constraints.
  - Each element of F is a set of states which must occur infinitely often in any execution path.
- In our example, F = {{<X=1,Y=1>}}
  - Avoid getting stuck at
    - <X=0,Y=1> or <X=1, Y = 0>

Granularity of a step

- Once a "thread" of a concurrent program is scheduled
  - How long can it execute without interruption from other "threads" ?
    - One statement ?
    - One instruction ?
  - This determines possible behaviors
    - Examples on this now !

One statement at a time

- Consider an asynchronous composition of two processes i.e. in
- any time step only one of them makes a move. These processes communicate via a single shared variable x. Both processes are executing the following infinite loop:
  - ***while true do x := x + x***
- Every time one of the processes is scheduled, it atomically
- executes x := x + x and then again another process is scheduled. The initial value of x is 1.
- What will be the values of x reached during system execution and why ?

One instruction at a time

- Suppose the infinite loop is compiled by a naïve compiler as follows. The sequence of instructions
- executed by process A and process B are shown.
- The processes are running asynchronously, and each time a process is scheduled, only its next instruction is executed
- atomically. Initially x = 1.

What values will x reach during system execution in this situation ? Explain your answer. Note that x is a shared global variable and reg_A^i, reg_B^i are local registers in processes A and B respectively.

A version of this problem is originally credited to the Thread Game by Prof. J. Strother Moore (University of Texas).

One instruction at a time

- L1: reg_A^1 = x
- reg_A^2 = x
- reg_A^3= reg_A^1+reg_A^2
- x = reg_A^3
- go to L1

- L2: reg_B^1 = x
- reg_B^2 = x
- reg_B^3= reg_B^1+reg_B^2
- x = reg_B^3
- go to L2

What are the possible values of x at the end of the program?

Process Communication in Asynchronous Execution

- Shared Variables
  - Mutual exclusion example discussed earlier
    - Programs communicate by setting and resetting the shared variable turn
- Message Passing
  - Typically, asynch. Msg. passing = Each program has its own FIFO queue(s) for receiving messages.
    - Also can be Synchronous message passing = Handshake between sender and receiver.

States/Transitions for Message passing

- Global state should take into account message queue contents.
- As before, in each time step
  - Only one program makes a move
- If the queues grow unboundedly, then what about the number of possible states, need to be careful !
- Here is a program to look at
  - do { 1 ! 2 (msg) } forever || do { 2 ? 1 (msg) } forever
    - 1 !2 --- 1 send to
- ***We do not discuss/use message passing in this course.***

Summary

- Kripke Structures as a formal model of behaviors of reactive systems
- Powerful enough to model behaviors of
  - Sequential as well as concurrent programs
  - Programs as well as circuits.
- Essentially a state transition graph with
  - Labeling of states (important for verification)
- We now need:
  - Language for specifying properties (Temporal Logics)
  - Technique for verifying these properties (Model Checking)
  - An efficient model checking tool
    - **SPIN for software/protocols [systems with program counter]**
    - **SMV for synchronous systems [systems driven by common clock --- circuits and low-level protocols]**

Keywords

- State = Valuation of program variables
  - May often include the program counter
- Transitions: In general a relation
  - In the absence of non-determinism, it is a function.
- Trace : (Infinite) Sequence of states
  - Captures computations of the system