Trends in Software Validation

Abhik Roychoudhury

CS 6214

Areas related to SW verif.

- Formal Methods
  - Model based techniques
  - Proof construction techniques
- Program Analysis
  - Static Analysis – Abstract Interpretation
- Software Engineering
  - Testing, Monitoring
  - Debugging, Program Understanding

Your Expertise

- … should be in two of the three areas (not just in formal methods)
- For more theoretical work, you should have good grounding in
  - Formal methods & Program Analysis
- For more practical work, you need
  - Formal methods & Software Engineering

Research Trends – High level

- In each individual area, as before.
- OR,
  - specifically target the technology of each area to be available for SW verif.
  - Combine techniques in different areas to develop hybrid verification methods.
  - Develop techniques for specific application areas.

Research Trends – Ex 1

- Specifically target the technology of each area to be available for SW verif.
  - Example: Generating models from code
  - Makes model checking available for SW
  - Often requires lot of engineering work
    - E.g. working out the model generation rules for constructs of a PL
    - Lot of work in integrating analysis packages e.g. alias analysis etc.

Research Trends - Ex 2

- Combine techniques in different areas to develop hybrid verification methods
  - Model Checking and Abstraction
    - Currently a hot research direction
  - Model Checking and Theorem Proving
    - What kinds of deduction ?
  - Model Checking and Program Debugging techniques (for enhanced program comprehension)

Model Checking & Abstraction

- Use of boolean abstractions for making an inf-state pgm amenable to MC.
- Modified later to form a
  - Abstract-MC-Refine loop
- *How to refine the abstraction if MC produced a counter-example ?*
  - Relevant: [Dill/LICS01, Clarke/CAV00]
  - This area is quickly getting saturated

Model Checking & Abstraction

- Search optimizations for different iterations of the Abst-MC-Refine loop
- *No need to search entire state space in all runs of model checking*
  - Caching of counter-example free state spaces for invariant properties.
  - Relevant paper : [Henzinger/POPL 02]

Model Checking & Deduction

- Abstraction based techniques are used to maintain finite approximation of the memory store of a program.
- The control locations are assumed to be finite, and control flow is maintained exactly.
- Could the control itself be not finite ?
  - YES

Parameterized systems

- Infinite family of finite-state systems
  - e.g. n-process token ring for all n
  - Every member of this family is a finite state system.
  - You could look at the entire family as a single program whose control location is given by the unbounded vector
    - < State of Proc1, State of Proc2,…. >

Parameterized Systems

- Extensions: each member of the family may also be infinite-state e.g. infinite domain data variables …
- Applications: Distributed programs
  - User-level
  - System-level (e.g. cache coherence protocol)
- *How to verify Parameterized Systems ?*

Parameterized System Verif.

- Finding out specific families for which a certain control abstraction is safe.
  - Restricted to well-known but simple examples
- Finitely represent the state space of a parameterized system by a rich language [regular exp say] and model check over this representation.
  - Relevant Paper: [Pnueli/CAV97]
- [Not saturated yet – still lot of effort in CAV/TACAS conferences]

Model Checking and Induction

- Induct over the recursive definition of a process network / protocol
  - Points us to a larger problem: Integrating a proof rule like induction with model checking
  - *Integration and interfacing of model checkers within theorem provers*
  - *Relevant paper: "Inductively verifying invariant properties of parameterized systems", Roychoudhury and Ramakrishnan, Automated Software Engineering Journal, 2004.*

Integration with Thm Proving

- A theorem prover produces proofs.
- A model checker produces yes/no and counter-examples (if any).
- Modify a model checker to produce a proof/disproof which can be fed to the theorem prover.
- Relevant Papers
  - [Roychoudhury/PPDP00], [Namjoshi/CAV01]

The list so far …

- Generating models from code
- Refinement strategies for abstraction refinement based software verification
- Techniques for verifying parameterized protocols in distributed systems
- Tight integration of model checkers into existing theorem provers

Other trends

- Much work needed on integrating formal techniques like model checking with software development activities like debugging
- Example: *Localizing the cause of an error (in terms of source code line numbers) from a counter-example*
- Relevant paper: [Ball/POPL03]

More on integration

- Model Checking, Theorem Proving, Abstract Interpretation are all static checking techniques.
- In practice, many dynamic checking techniques exist for validation
  - Run-time monitoring
  - Record and replay (post-mortem)
- Combination of static and dynamic checking techniques is way open and speculative
  - *Many topics of interest here !!*

Why dynamic ?

- Static checking methods are
  - Either non automated
    - Theorem Proving
  - Or of high complexity and inaccurate
    - Model Checking, needs abst. Refinement.
- Debugging is typically for a single program run
  - Testing a program for a selected input.

Dynamic Slicing

- Criterion
  - Program Input
  - Control location (a selected line of text)
  - Variable (could be an object or a field of an object)
- Output
  - All program lines which directly or indirectly affect the criterion.

Example 1

```
0    scanf("%d", &A);
1.   V = A;
2    W = X;
3    U = V;
4    printf("%d\n", U);
```

Criterion
(A=0, 4, U)

Slice
{0,1,3,4}

Just follows through a chain of data dependences.

Example 2

```
0    scanf("%d", &A);
1.   If (A == 0){;
2        W = X;
3        U = A;
4    }
5    printf("%d\n", U);
```

Criterion (A == 0, 5, U)

Slice {0,1,3,4}

Follow through chain of control and data dependences.

Dynamic Dependence Graph

- G = (V, E)
  - V = All statement occurrences for the test input under consideration.
  - E = Data and Control dependences between statement occurrences.
- Slicing Criterion
  - A node in the DDG
- Slice computation
  - Nodes reachable from slicing criterion

Data dependences

V := 1;
…
U := V

An edge from a variable usage to the latest definition of the variable.

A[i] := 1;
…
U := A[j]

→ Do we consider this data dependence edge ?

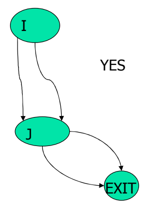
→ Remember that the slicing is for an input, so the addresses are resolved

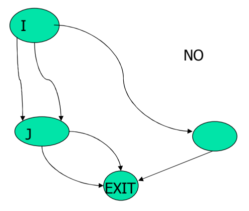
→ We thus define data dependences corresponding to memory locations rather than variable names.

Control Dependences

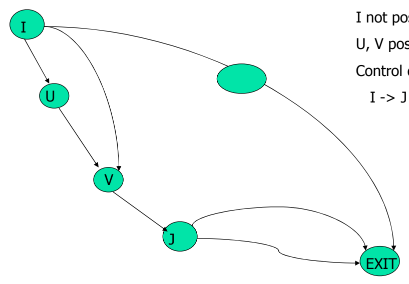
Post-dominated: I,J – nodes in Control Flow Graph

I is post-dominated by J iff all paths from I to EXIT pass through J





Control Dependences



I not post-dom by J

U, V post-dom by J

Control dependence

I -> J

Dynamic Slice Illustration

```
0    scanf("%d", &A);
1.   If (A == 0){;
2      W = X;
3      U = A;
4    }
5    printf("%d\n", U);
```

Criterion: 5,U with input A == 0

Trace: <0,1,2,3,4,5>   Slice = {0,1,3,5}

Data dependences encountered   5 -> 3, 1 -> 0

Control dependence encountered 3 -> 1

Omission Errors

```
1 b = 1;
2 x = 1;
3 if (a > 1) {
4     if (b > 1){
5         x = 2
      }
}
6 ... = x
```

Criterion

6,x with input a= 2

Trace = <1,2,3,4,6>

Slice = {2,6}

The bug could be in line 1 leading to a wrong omission of line 5's execution, thereby affecting the slicing criterion.

-- **[Wang + Roychoudhury 2004]**

Can extend the notion of dynamic slice to include this, but the slice keeps on getting bigger. We need mechanisms to pinpoint the error !

Fault Localization

```
1 b = 1;
2 x = 1;
3 if (a > 1) {
4     if (b > 1){
5         x = 2
      }
}
6 ... = x
```

Criterion

6,x with input a= 2

Trace = <1,2,3,4,6>

Slice = {2,6}

The bug could be in line 1.

How to locate the bug from execution trace ?

What is a bug / fault ?

- Two possibilities
  - The programmer has clear intuition about certain unacceptable behaviors and states them as (typically) invariant properties
    - G (pc == 6 => x == 1)
  - More likely: The programmer discovers undesirable behaviors during testing
    - For input a==0, I did not expect x == 1 at pc == 6
    - Bug !

So…

- We have
  - Weak Error Specification
  - Counter-example trace (the run showing the "undesirable behavior")
- Contrast this with static checking via MC
  - Precise Error Specification
  - No counter-example – we are attempting to find it via MC.

Fault Localization via MC

- Take a precise error spec. as invariant
  - MC and find counter-example trace σ
  - Find transitions in σ which do not appear in any correct trace
    - Using precise notion of "correct" trace.
    - Need a separate inter-procedural analysis algorithm to collect transitions in correct traces
    - Compares code coverage between correct and incorrect traces – seq. of stmts forgotten
- Ball, Naik and Rajamani – POPL 2003.

Another approach based on …

- … comparison of correct/incorrect runs.
- Classify the runs for a large pool of inputs as failing or successful – no Temporal logic properties required.
- Failing run is usually given – encountered during testing / debugging.
- Compare code coverage of failing & succ. Runs
  - (Stmts executed in f – Statements executed in s) for all s
- Choose the s with smallest distance and report corresponding distance
  - Renieris/Reiss – ASE 2003,
  - Wang/Roychoudhury ASE 2005

Overall comments

- Opportunities exist – but saturation (in terms of good work) will come in few years, I think !
- Basic References
  - Dynamic Program Slicing – Agrawal and Horgan, PLDI 1990.
  - A Survey of Program Slicing Techniques, Frank Tip, 1995.

Trends at high-level …

- Specifically target the technology of each area to be available for SW verif.
- Combine techniques in different areas to develop hybrid verification methods.
- Develop techniques for specific application areas.
  - Some thoughts for embedded software/protocols/interfaces.

ES

- A computing system which is part of a "larger system" (read – device).
- The larger system constitutes the environment – in continuous interaction.
- The computing system implements a specific functionality.
  - A dedicated computer implemented by a combination of hardware and software.

ES examples

- Automobiles
- Train control systems
- Avionics / Flight control
- Nuclear Power Plants
- Inside medical devices (for image manipulation) and other purposes
- Safety first ! Validation of these control software more important

ES examples

- Or more vanilla
- HDTV
- Washing Machines
- Microwave
- Controllers for other household devices such as Air-con
- Finally, smart room / wear (e.g. GA Tech)

FV for ES – The reality

- Is it any different ?
  - Current verif. Techniques should scale up for ES Hardware.
  - Embodies hardware/software interaction: often real-time.
  - Reuse of vendor provided IP blocks
    - **Implementation not known for Intellectual property reasons**
  - No single person/team knows the entire design.

FV for ES - Consequences

- A realistic ES design typically consists of:
  - **Number of IP blocks**
  - **Connected to one/more system bus via bridges**
- View an IP block as a hardware subroutine – Need **REUSE**.
- Reuse IP blocks designed by others (and provided by vendors).
- Each IP block comes with *interface specification*
  - **Input and output signals**
  - **Some timing diagrams**

FV for ES - Issues

- Synthesizing component interface software reliably.
  - Need high level modeling (UML is one choice).
  - Synthesize executable description from high level models.
- Still you might be using other's components !!
  - Static verification of unknown components
    - *Assume guarantee reasoning*
    - Extract assumptions/guarantees from interface spec.
- *Lot of opportunities in research in the ES area.*
  - *But cannot be routine application of general-purpose methods.*