# Static Program Transformations for Efficient Software Model Checking

Shobha Vasudevan

Jacob Abraham

The University of Texas at Austin

# Dependable Systems

- Large and complex systems
- Software faults are major concern
- Dependability achieved by
  - Testing
  - Debugging
  - Formal Verification

# Formal Verification: Model Checking

- Formal description of model
- Property specified in temporal logic (LTL, CTL etc)
- State space explosion for reasonably sized systems

# A Solution: Abstractions

- Model checking models need to be made smaller
- Smaller or “reduced” models must retain information
  - Property being checked should yield same result
- Balancing solution: Abstractions

# Program Transformation Based Abstractions

- Abstractions on Kripke structures
  - Cone of Influence (COI), Symmetry, Partial Order, etc.
  - State transition graphs for even small programs can be very large to build
- Abstractions on Program Text
  - Scale well with program size
  - High economic interest

Static Program Transformations

# Types of Abstractions

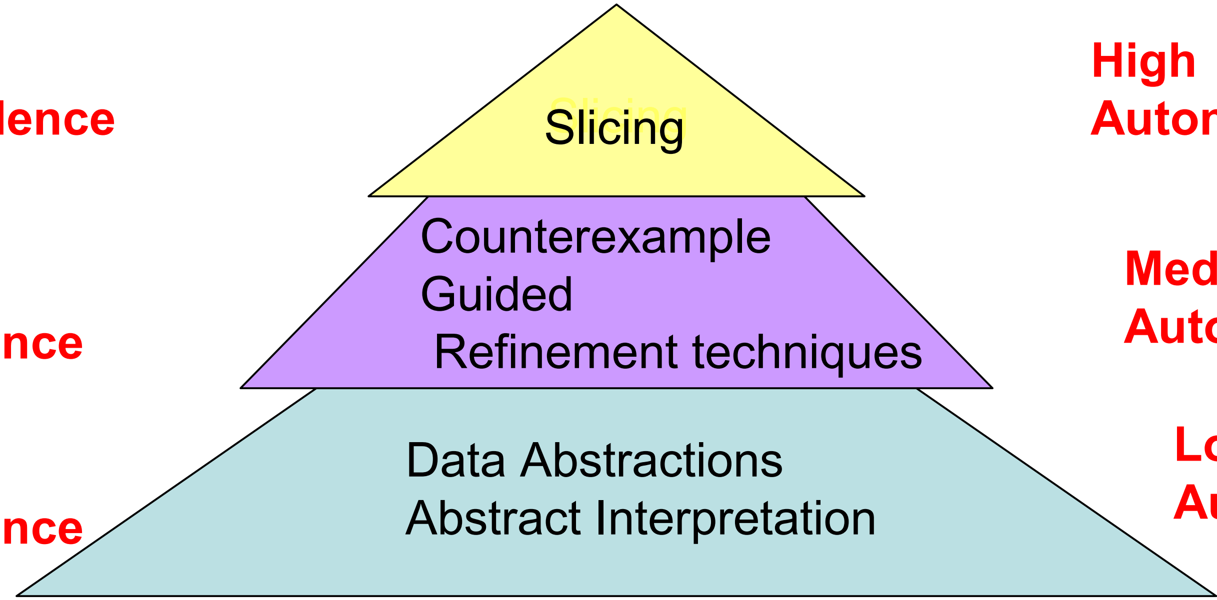
- Sound
  - Property holds in abstraction implies property holds in the original program
- Complete
  - Algorithm always finds an abstract program if it exists
- Exact
  - Property holds in the abstraction iff property holds in the main program

# Abstraction Landscape

**High**
**Property Dependence**

**Medium**
**Property Dependence**

**Low**
**Property Dependence**




**High**
**Automation**

**Medium**
**Automation**

**Low**
**Automation**

# Data Abstractions

- Abstract data information
  - Typically manual abstractions
- Infinite behavior of system abstracted
  - Each variable replaced by abstract domain variable
  - Each operation replaced by abstract domain operation
- Data independent Systems
  - Data values do not affect computation
  - Datapath entirely abstracted

# Data Abstractions: Examples

- Arithmetic operations
  - Congruence modulo an integer
    - *k replaced by k mod m*
- High orders of magnitude
  - Logarithmic values instead of actual data value
- Bitwise logical operations
  - Large bit vector to single bit value
    - *Parity generator*
- Cumbersome enumeration of data values
  - Symbolic values of data

# Abstract Interpretation

- Abstraction function mapping concrete domain values to abstract domain values
- Over-approximation of program behavior
  - Every execution corresponds to abstract execution
- Abstract semantics constructed once, manually

# Abstract Interpretation: Examples

- Sign abstraction
  - Replace integers by their sign
    - *Each integer K replaced by* one of {> 0, < 0, =0}
- Interval Abstraction
  - Approximates integers by maximal and minimal values
    - *Counter variable i replaced by lower and upper limits of loop*
- Relational Abstraction
  - Retain relationship between sets of data values
    - *Set of integers replaced by their convex hull*

# Counterexample Guided Refinement

- Approximation on set of states
  - Initial state to bad path
- Successive refinement of approximation
  - Forward or backward passes
- Process repeated until fixpoint is reached
  - Empty resulting set of states implies property proved
  - Otherwise, counterexample is found
- *Counterexample can be spurious because of over-approximations*
- Heuristics used to determine spuriousness of counterexamples

# Counterexample Guided Refinement

- Predicate Abstraction
  - Predicates related to property being verified (User defined)
  - Theorem provers compute the abstract program
  - Spurious counterexamples determined by symbolic algorithms
  - Some techniques use error traces to identify relevant predicates

# Counterexample Guided Refinement

- Lazy Abstraction
  - More efficient algorithm
  - Abstraction is done on-the-fly
  - Minimal information necessary to validate a property is maintained
    - Abstract state where counterexample fails is “pivot state”
    - Refinement is done only “from the pivot state on”

# Program Slicing

- Program transformation involving statement deletion
- "Relevant statements" determined according to *slicing criterion*
- Slice construction is completely *automatic*
- Correctness is *property specific*
  - Loss of generality
- Abstractions are sound and complete

# Specialized Slicing Techniques

- Static slicing produces large slices
  - Has been used for verification
  - Semantically equivalent to COI reductions
- Slicing criterion can be enhanced to produce other types of slices
  - Amorphous Slicing
  - Conditioned Slicing

# Our Contribution: Specialized Slicing for Verification

- Amorphous Slicing
  - Static slicing preserves syntax of program
  - Amorphous Slicing does not follow syntax preservation
  - Semantic property of the slice is retained
  - Uses rewriting rules for program transformation

# Example of Amorphous Slicing

```
begin
   i = start;
   while (i <= (start + num))
        {
        result = K + f(i);
        sum = sum + result;
        i = i + 1;
        }
end
```

LTL Property: G sum > K
Slicing Criterion: (end, {sum, K})

# Example of Amorphous Slicing

Amorphous Slice:

```
begin
    sum = sum + K + f(start);
    sum = sum + K + f(start + num);
end
```

Program Transformation rules applied

- Induction variable elimination
- Dependent assignment removal

- Amorphous Slice takes a fraction of the time as the real slice on SPIN

# Amorphous Slicing for Verification

- Similar to term rewriting
  - Used by theorem provers for deductive verification
- What is different?
  - Theorem provers try to prove entirely by rewriting
  - We propose a hybrid approach
    - Rewriting only part of the program, based on slicing criterion
    - Model checking the sliced program

# Conditioned Slicing

- Theoretical bridge between static and dynamic slicing
- Conditioned Slices specify initial state in criterion
  - Constructed with respect to set of possible inputs
  - Characterized by first order predicate formula
- Yields much smaller slices than static slices

# Conditioned Slicing for Verification

- Safety properties specified as:
  - Antecedent => Consequent

- For these properties, *antecedent can be used to specify the initial states of interest*
  - We do not need states where antecedent is not true
  - Static slices preserves all possible executions

# Conditioned Slicing for Verification

- Abstractions created by conditioned slicing of antecedents in formula
  - *Antecedent Conditioned Slices*
- Exact abstractions
- Automatic construction of slices

# Example Program

```
    begin
1:          read(N);
2:          A = 1;
3:          if (N < 0)
                {
4:                  B = f(A);
5:                  C = g(A);
                }
            else
6:              if (N > 0)
                    {
7:                      B = f'(A);
8:                      C = g'(A);
                    }
                else
                    {
9:                      B = f''(A);
10:                         C = g''(A);
                    }
11:             print(B);
12:             print(C);
    end
```

# Static Slice of Program

```
     begin
1:          read(N);
2:          A = 1;
3:          if (N < 0)
                {
4:                   B = f(A);
5:                   C = g(A);
                }
            else
6:              if (N > 0)
                    {
7:                       B = f'(A);
8:                       C = g'(A);
                    }
                else
                    {
9:                       B = f"(A);
10:                           C = g"(A);
                    }
11:             print(B);
12:             print(C);
     end
```

Slicing criterion:

<(N<0), 11, B>

# Conditioned Slice of Program

```
    begin
1:        read(N);
2:        A = 1;
3:        if (N < 0)
              {
4:                B = f(A);
5:                C = g(A);
              }
          else
6:            if (N > 0)
                  {
7:                    B = f'(A);
8:                    C = g'(A);
                  }
              else
                  {
9:                    B = f"(A);
10:                       C = g"(A);
                  }
11:           print(B);
12:           print(C);
    end
```

Slicing criterion:

<(N<0), 11, B>

# Preliminary Experimental Results

- Group Address Registration Protocol (GARP) and X.509 authentication protocol
- SPIN model checker
  - Memory limit of 512 MB given
  - Max search depth of $2^{20}$ steps
- All properties were in the form

  Antecedent => Consequent

# Preliminary Experimental Results

| Property | Unsliced* | Conditioned Sliced | Property Proved |
|---|---|---|---|
| P1 | 91.65 | 1.72 | Yes |
| P2 | 145.78 | 8.44 | Yes |
| P3 | 145.36 | 8.41 | Yes |
| P4 | 154.96 | 1.95 | Yes |
| P5 | 117.81 | 10.23 | Yes |

*Static slicing in SPIN was enabled

# Conclusions

- Abstraction techniques are evaluated by
  - Degree of automation vs. Manual effort
  - Property dependence vs. Generic nature
  - Exact vs. Over-approximation
- “Software reliability is the grand challenge of the next decade”
  - Abstractions are the powerful candidate solutions to this challenge
  - Need integration of all abstraction techniques into an optimal framework