Verification of SoC Designs

- Simulation-based techniques
- Formal analysis
- Dealing with state explosion
- Verification of embedded software

Verification versus Test

```
Specification
  ↓ Design
  ↓ Verification
Implementation
  ↓ Hardware/Software
  ↓ Manufacture
  ↓ Test
Hardware
```
Verification Effort

- Verification portion of design increases to anywhere from 50 to 80% of total development effort for the design.

Percentage of Total Flaws

- About 50% of flaws are functional flaws.
  - Need verification method to fix logical & functional flaws
Verification Approaches

- Simulation
- Hardware Accelerated Simulation
- Emulation
- Formal Verification
- Semi-formal Verification
- Prototyping
- Basic verification tool
- Bigger coverage
- Faster speed, closer to final product

HW/SW Co-Design

- Concurrent design of HW/SW components
- Evaluate the effect of a design decision at early stage by “virtual prototyping”
- Co-verification
Verification Options

- Simulation Technologies
- Equivalence Checking
- Formal Analysis (Model Checking)
- Physical Verification and Analysis

Simulation Technologies

- Event-based Simulators
- Cycle-based Simulators
- Transaction-based Simulators
- Code Coverage
- HW/SW Co-verification
- Emulation Systems
- Rapid Prototyping Systems
- Hardware Accelerators
- AMS Simulation
- Numerical Simulation (MATLAB)
Static Technologies

• “Lint” Checking
  – Syntactic correctness
  – Identifies simple errors

• Static Timing Verification
  – Setup, hold, delay timing requirements
  – Challenging: multiple sources

Formal Techniques

• Theorem Proving Techniques
  – Proof-based
  – Not fully automatic

• Formal Model Checking
  – Model-based
  – Automatic

• Formal Equivalence Checking
  – Reference design ↔ modified design
  – RTL-RTL, RTL-Gate, Gate-Gate implementations
  – No timing verification
Physical Verification & Analysis

Issues for physical verification:
- Timing
- Signal Integrity
- Crosstalk
- IR drop
- Electro-migration
- Power analysis
- Process antenna effects
- Phase shift mask
- Optical proximity correction

Top-Down SoC Verification
Bottom-Up SoC Verification

- Components, blocks, units
- Memory map, internal interconnect
- Basic functionality, external interconnect
- System level

Platform Based SoC Verification

- Derivative Design
- Interconnect Verification between:
  - SoC Platform
  - Newly added IPs
System Interface-driven SoC Verification

Besides Design-Under-Test, all others are interface models

Traditional Testbench

- Problems of Traditional Testbench
  - Real-World Stimuli
  - System-Level Modeling
  - High-Level Algorithmic Modeling
  - Test Automation
  - Source Coverage
Executable Specification

- Procedural Language for Behavioral Modeling
  - Design Productivity
    - Easy to model complex algorithm
    - Fast execution
    - Simple Testbench
  - Tools
    - Native C/C++ through PLI/FLI
    - Extended C/C++ : SpecC, SystemC
- Verify it on the fly!
  - Test vector generation
  - Compare RTL Code with Behavioral Model
  - Coverage Test
Property Detection

Property detection: to decide whether a simulation run (trace) of a design satisfies a given property

trace (simulation run) → property detection module → yes / witness
property (specification) → no / counterexample
e.g., violation of mutual exclusion, critical₁ \( \not\equiv \) critical₂

Example: Properties written in PSL/Sugar

Specifying Properties (Assertions) in Industry Tools

- Open Vera Assertions Language (Synopsys)
- Property Specification Language (PSL) (IBM, based on Sugar)
  - Accelera driving consortium
  - IEEE Std. 1850-2005
- Accelera Open Verification Library (OVL) provides ready to use assertion functions in the form of VHDL and Verilog HDL libraries
- SystemVerilog is a next generation language, added to the core Verilog HDL
  - IEEE Std. 1800-2005
Formal Verification of SoCs

Property Checking

- Assumptions & Properties
- Model under Test
- Check Satisfaction of Properties
- Yes/No

Equivalence Checking

- Reference Model
- Model under Test
- Check Functional Equivalence
- Yes/No

State Explosion!

- Number of Storage Elements vs. Number of States
- Data points showing exponential growth

- Number of latches in Itanium processor

- World population
- Stars in the Universe
- Protons in the Universe
Abstractions to Deal with Large State Spaces

- Model checking models need to be made smaller
- Problem: State-Space Explosion
- 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**
  - Slicing
  - Counterexample
  - Guided Refinement techniques
  - Data Abstractions Abstract Interpretation

- **Medium Property Dependence**
  - High Automation
  - Medium Automation

- **Low Property Dependence**
  - Low Automation
Verification of challenging problems with high level static analysis

**Antecedent Conditioned Slicing**

- RTL abstraction technique
- Applied to LTL formulas
  - \( G(a => c) \)
- Theoretically complex, practically effective
- USB 2.0 protocol verification

**Property checking**

- High level symbolic simulation
  - Symbolic simulation of antecedent
  - Symbolic simulation of all CFG nodes
- Domain aware analysis
  - Function-wise case splitting
- Decision procedure
  - Model checker

---

**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

### Conditioned Slicing

- Slices constructed with respect to set of possible input states
- Characterized by first order, predicate logic formula
- Augments static slicing by introducing condition
  - \(<C, I, V>\)
  - Constrains the program according to condition \(C\)
- Canfora et al
Example Program

```
begin
1:    read(N);
2:    A = 1;
3:    if (N < 0) {
4:        B = f(A);
5:        C = g(A);
6:    } else 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
```

Example Program: Static Slicing wrt <11, B>

```
begin
1:    read(N);
2:    A = 1;
3:    if (N < 0) {
4:        B = f(A);
5:        C = g(A);
6:    } else 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
```
Example Program:  
Conditioned Slicing wrt  
\(<(N<0), 11, B>\)

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

Verification Using Conditioned Slicing

- Slicing part of design irrelevant to property being verified
- Safety Properties of the form
  \(- G (\text{antecedent} \Rightarrow \text{consequent})\)
- Use \textit{antecedent} to specify states we are interested in

\textit{We do not need to preserve program executions where the antecedent is false}
Property checking: antecedent conditioned slicing

\[ h: G \left( A \Rightarrow C \right) \]

if \( (A) \)
\[ C = 1; \]
else
\[ C = 0; \]

Antecedent conditioned slicing on \( <A= \text{true}>, A, C \)

Semantic analysis

Example of antecedent conditioned slicing

\[
\begin{align*}
\text{always @ (clk) begin} & \quad \text{always @ (clk) begin} & \quad \text{always @ (clk) begin} \\
\text{case(insn) case(dec) case(ex)} & \quad & \\
\text{f_add: dec = d_add;} & \quad \text{d_add: ex = e_add;} & \quad \text{e_add: res = a+b;} \\
\text{f_sub: dec = d_sub;} & \quad \text{d_sub: ex = e_sub;} & \quad \text{e_sub: res = a-b;} \\
\text{f_and: dec = d_and;} & \quad \text{d_and: ex = e_and;} & \quad \text{e_and: res = a\&b;} \\
\text{f_or: dec = d_or;} & \quad \text{d_or: ex = e_or;} & \quad \text{e_or: res = alb;} \\
\text{endcase} & \quad \text{endcase} & \quad \text{endcase} \\
\text{end} & \quad \text{end} & \quad \text{end}
\end{align*}
\]

\[
h = \left[ G((\text{insn} == \text{f_add}) \Rightarrow XX(\text{res} == a+b)) \right]
\]
Example contd.

Checking the truth of the antecedent

Antecedent symbolic expression

Symbolic simulation of a node in CFG

Rewriter

Node retained

T: Retained
X: Retained
F: Not retained
Complexity of Antecedent Conditioned Slicing

- Symbolic simulation of all nodes in each process
- Expression computation over all processes in the program
  - Handles global predicates
- Symbolic simulation of the antecedent
- Looking forward in time
  - Depends on \( n \) in \((A \Rightarrow X^n C)\)
- Decision procedure for checking truth of antecedent
  - Could be arbitrarily hard
- Path traversal of all processes
  - Pruning non-retained nodes
- **Worst case: retain all nodes**

Correctness of Antecedent Conditioned Slicing

**Theorem:** An LTL formula \( h \) of the type, where \( h \) is

\[
G(a \Rightarrow c) \\
G (a \Rightarrow X^n c) \\
G (a \Rightarrow F^{\leq k} c)
\]

holds on the original program \( \text{iff} \) it holds on the antecedent conditioned slice.

**Proof intuition:**

For a Kripke structure of the slice, all states satisfy \( a \Rightarrow c \).
These include states of the original Kripke structure that satisfy \( a \).
Thus all states of the original that satisfy \( a \) must satisfy \( h \).
All states of the original that satisfy \( \neg a \), satisfy \( a \Rightarrow c \) vacuously.
Example of Antecedent Conditioned Slicing

```verilog
always @ (clk) begin
    case(insn)
        f_add: dec = d_add;
    endcase
end

always @ (clk) begin
    case(dec)
        d_add: ex = e_add; e_add: res = a+b;
    endcase
end

always @ (clk) begin
    case(ex)
        e_add: res = a+b;
    endcase
end
```

Single instruction behavior for f_add instruction

\[ h = [G((insn == f_add) \Rightarrow XX(res == a+b))] \]

Experimental Results

- Verilog RTL implementation of USB 2.0 function core
- Properties taken from specification document
  - Safety properties expressed in LTL
  - Mostly control based, state machine related
- Used Cadence SMV-BMC
  - Circuit too big for SMV
  - Used a bound of 24
- 450 MHz, Ultra Sparc dual processor with 1 GB RAM
### Results on USB $G(a => c)$ Properties

CPU Seconds, 450 MHz dual UltraSPARC-II with 1 GB RAM

#### Results of Antecedent Conditioned Slicing

- Temporal property verification for USB 2.0
- Safety properties of the form
  - $G(a => Xc)$
  - $G(a => a U_s c)$
- Liveness Properties
  - $G(a => Fc)$
- USB has many interacting state machines
  - Approximately $10^{33}$ states
- Bound of 50
Example Properties of the USB

- \( G((\text{crc5err}) \lor \neg(\text{match}) \Rightarrow \neg(\text{send_token})) \)
  - If a packet with a bad CRC5 is received, or there is an endpoint field mismatch, the token is ignored
- \( G((\text{state} == \text{SPEED\_NEG\_FS}) \Rightarrow X((\text{mode\_hs}) \land (T1_{gt\_3\_0ms}) \Rightarrow (\text{next\_state} == \text{RES\_SUSPEND})) \)
  - If the machine is in the speed negotiation state, then in the next clock cycle, if it is in high speed mode for more than 3 ms, it will go to the suspend state
- \( G((\text{state} == \text{RESUME\_WAIT}) \land \neg(\text{idle\_cnt\_clr}) \Rightarrow F(\text{state} == \text{NORMAL})) \)
  - If the machine is waiting to resume operation and a counter is set, eventually (after 100 mS) it will return to normal operation

Results on Temporal USB Properties

CPU Seconds, 450 MHz dual UltraSPARC-II with 1 GB RAM
Verification of challenging problems with high level static analysis

- Antecedent conditioned slicing
- Domain aware analysis
  - Instruction wise case splitting
- Decision procedure
  - Model checker

• Reason with the entire state of the machine (Burch and Dill)
• Enhancements use theorem proving techniques
  - Significant manual component
  - Construct complicated invariants
  - High-level model based

Automatic techniques do not scale to instruction level verification

Single instruction verification

- Obtain single instruction machine by antecedent conditioned slicing
  - Antecedent is instruction word
- Property is $G (I \Rightarrow R)$ where
  - $I = i_1 \land X_i_2 \land XXi_3 \ldots X^n_i_n$
    - $i_t$ represents the antecedent in pipeline stage $t$
  - $R$ is the result of $I$ in terms of its target register values

Model checking of instruction $I$
Interaction between instructions

**Lemma:** Instructions should write back only to target register only on writeback stage

Notion of processor correctness

**Theorem:** The instruction slices, when executed in the same sequence as the corresponding instructions in the original pipelined machine, will produce the same result as the original pipeline.
**Results of OR1200 verification**

<table>
<thead>
<tr>
<th>Class</th>
<th>Insn</th>
<th>SMV Time(s)</th>
<th>Memory usage (KB)</th>
<th>SMV Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>l.add</td>
<td>25.65</td>
<td>23796</td>
<td>DNF</td>
</tr>
<tr>
<td>ALU</td>
<td>l.sub</td>
<td>24.7</td>
<td>24018</td>
<td>DNF</td>
</tr>
<tr>
<td>ALU</td>
<td>l.addi</td>
<td>21.6</td>
<td>19658</td>
<td>DNF</td>
</tr>
<tr>
<td>ALU</td>
<td>l.xor</td>
<td>24.84</td>
<td>24831</td>
<td>DNF</td>
</tr>
<tr>
<td>ALU</td>
<td>l.and</td>
<td>23.28</td>
<td>21727</td>
<td>DNF</td>
</tr>
<tr>
<td>ALU</td>
<td>l.or</td>
<td>24.01</td>
<td>22761</td>
<td>DNF</td>
</tr>
<tr>
<td>MAC</td>
<td>l.mul</td>
<td>25.28</td>
<td>49831</td>
<td>DNF</td>
</tr>
<tr>
<td>MAC</td>
<td>l.mulu</td>
<td>26.63</td>
<td>22801</td>
<td>DNF</td>
</tr>
<tr>
<td>BRANCH</td>
<td>l. bf</td>
<td>132.63</td>
<td>44281</td>
<td>DNF</td>
</tr>
<tr>
<td>BRANCH</td>
<td>l. bnf</td>
<td>139.47</td>
<td>46350</td>
<td>DNF</td>
</tr>
<tr>
<td>BRANCH</td>
<td>l. j</td>
<td>57.36</td>
<td>31969</td>
<td>DNF</td>
</tr>
<tr>
<td>BRANCH</td>
<td>l. jalr</td>
<td>54.09</td>
<td>31073</td>
<td>DNF</td>
</tr>
<tr>
<td>BRANCH</td>
<td>l. cmov</td>
<td>159.64</td>
<td>30094</td>
<td>DNF</td>
</tr>
</tbody>
</table>

- OpenRISC 1200
- 32-bit scalar RISC processor
- 5 stage integer pipeline
- Publicly available
- Intended for portable/embedded applications

All instructions of a pipelined processor were verified
Verification of challenging problems with high level static analysis

**Sequential equivalence checking**
- High level symbolic simulation of RTL implementation
- High level symbolic simulation of System level spec
- Domain aware analysis
  - Sequential compare points obtained using heuristics
- Decision procedure
  - SAT solver

Term Rewriting for Arithmetic Circuit Checking

- Significant success with RTL Term level reductions
- Verification of arithmetic circuits at the RTL level using *term rewriting*
- RTL to RTL equivalence checking
- Verified *large multiplier designs* like Booth, Wallace Tree and many optimized multipliers using this rewriting technique
Term Rewriting Systems: Example

- Terms: $GCD(x, y)$
- Rewrite rules:
  - $GCD(x, y) \Rightarrow GCD(y, x)$ if $x > y$, $y \neq 0$
  - $GCD(x, y) \Rightarrow GCD(x, y-x)$ if $x \cdot y$, $y \neq 0$
- Initial term: $GCD(initX, initY)$

VERIFIRE

- Dedicated Arithmetic Circuit Checker
- Vtrans: Translates Verilog designs to Term Rewriting Systems
- Vprover: Proves equivalence of Term Rewriting Systems
  - Iterative engine
  - Returns error trace if proof not found
  - Maintains an expanding rule base for expression minimization
  - Incomplete, but efficient engine
Results on Multipliers

<table>
<thead>
<tr>
<th>Wallace Tree</th>
<th>VERIFIRE</th>
<th>Commercial Tool 1</th>
<th>Commercial Tool 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 X 4</td>
<td>14s</td>
<td>10s</td>
<td>9s</td>
</tr>
<tr>
<td>8 X 8</td>
<td>18s</td>
<td>18s</td>
<td>16s</td>
</tr>
<tr>
<td>16 X 16</td>
<td>25s</td>
<td>Unfinished</td>
<td>Unfinished</td>
</tr>
<tr>
<td>32 X 32</td>
<td>40s</td>
<td>Unfinished</td>
<td>Unfinished</td>
</tr>
<tr>
<td>64 X 64</td>
<td>60s</td>
<td>Unfinished</td>
<td>Unfinished</td>
</tr>
</tbody>
</table>

System Level Equivalence Checking

- Sequential equivalence checking
  - Verifying two models with different state encodings
- System specifications as system level model (SLM)
  - Higher level of abstraction
  - Timing aware models
- Design concept in RTL needs checking
  - Retiming, power, area modifications
  - Every change requires verification against SLM
- Simulation of SLM
  - Tedious to develop
  - Inordinately long running times
Equivalence Checking Using Sequential Compare Points

- Variables of interest (observables) obtained from user/block diagram
  - Primary outputs / Relevant intermediate variables
- Symbolic expressions obtained for observables assigned in a given cycle (high level symbolic simulation)
- Introduce notion of sequential compare points
  - Identification with respect to relative position in time
  - Identification with respect to space (data or variables)
- Symbolic expressions compared at sequential compare points
- Comparison using a SAT solver in this work
  - Other Boolean level engines can also be used

Algorithm

1. M: System level model
   V: RTL model
   O: list of observables
2. Construct the control flow graph for both M and V
3. Compute symbolic expression at sequential compare point C using high level symbolic simulation for both M and V
4. Check equivalence of symbolic expressions at sequential compare point C using a SAT solver
   - If satisfiable: Obtain Proof
   - If not satisfiable: Error Trace

For all sequential compare points C
Correctness theorem

Theorem:
Let two systems $M$ and $V$ such that, $PI(M) = PI(V)$ and $PO(M) = PO(V) = PO$.
Let $n$ be the longest cycle length taken to obtain all primary outputs in both systems.
Let $M$ and $V$ be compared at every point $C = (t,d)$ such that $t \leq n$.
Let $\sim_c$ be the simulation relation that denotes the symbolic expression equality at $C$.
Then, for all $C$, $V \sim_c M \Rightarrow V \sim_c 0$. 

Proof intuition:
The base case is at time $t=0$, at initial state.
The induction hypothesis is relieved using a lemma that proves that at any cycle $t$, if the two systems have the same value for the symbolic expression of all variables $d$, $\sim_c$ holds at that cycle.
If all primary outputs are generated by cycle $n$, the relation holds.

Viterbi Decoder: SystemC Specification
Viterbi Decoder Implementation 1

Decomposition of Equivalence Checking between SystemC and Implementation - 1
Proof of FF Buffer (8 cycles)

Proof of Trellis Computation (2 cycles)

Proof of Matdec DecisionTable (2 cycles)
### Results of using a SAT solver

<table>
<thead>
<tr>
<th>Block/Function</th>
<th>Number of clauses in the CNF formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>448</td>
</tr>
<tr>
<td>LESSTHAN</td>
<td>32</td>
</tr>
<tr>
<td>Trellis Condition in the butterfly</td>
<td>14336</td>
</tr>
<tr>
<td>Trellis computation in each stage of butterfly</td>
<td>28672</td>
</tr>
<tr>
<td>Trellis per butterfly</td>
<td>57344</td>
</tr>
<tr>
<td>MatDec each stage of butterfly</td>
<td>896</td>
</tr>
<tr>
<td>MatDec per butterfly</td>
<td>1792</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design</th>
<th>Number of clauses in the CNF formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monolithic Trellis</td>
<td>1892352</td>
</tr>
<tr>
<td>RTL decomposition (Design 1)</td>
<td>59136</td>
</tr>
<tr>
<td>RTL decomposition (Design 2)</td>
<td>59136</td>
</tr>
</tbody>
</table>

### Results

<table>
<thead>
<tr>
<th>Block/Function</th>
<th>Number of variables</th>
<th>Number of symbolic variables generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Butterfly</td>
<td>128</td>
<td>66</td>
</tr>
<tr>
<td>Trellis (monolithic)</td>
<td>2304</td>
<td>2112</td>
</tr>
<tr>
<td>Trellis (decomposed)</td>
<td>128</td>
<td>66</td>
</tr>
</tbody>
</table>
Verifying Embedded Software

- Software Testing
  - Execute software for test cases
  - Analogous to simulation in hardware
- Testing Criteria
  - Coverage measures
- Formal analysis of software
  - Model Checking
  - Theorem Proving

Path Testing

- Assumption: bugs affect the control flow
- Execute all possible control flow paths through the program
  - Attempt 100% path coverage
- Execute all statements in program at least once
  - 100% statement coverage
- Exercise every branch alternative during test
  - Attempt 100% branch coverage
Software Verification

- Formal analysis of code
- Result, if obtained, is guaranteed for all possible inputs and all possible states
- Example of software model checker: SPIN
- Problem: applicable only to small modules

State Explosion

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”
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:

\[
\begin{align*}
\text{begin} \\
\text{sum} &= \text{sum} + K + f(\text{start}); \\
\text{sum} &= \text{sum} + K + f(\text{start} + \text{num}); \\
\text{end}
\end{align*}
\]

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
  - 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

Example Results – Conditioned Slicing

- 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 $\Rightarrow$ Consequent
## Experimental Results

<table>
<thead>
<tr>
<th>Property</th>
<th>Unsliced</th>
<th>Conditioned Sliced</th>
<th>Property Proved</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>91.65</td>
<td>1.72</td>
<td>Yes</td>
</tr>
<tr>
<td>P2</td>
<td>145.78</td>
<td>8.44</td>
<td>Yes</td>
</tr>
<tr>
<td>P3</td>
<td>145.36</td>
<td>8.41</td>
<td>Yes</td>
</tr>
<tr>
<td>P4</td>
<td>154.96</td>
<td>1.95</td>
<td>Yes</td>
</tr>
<tr>
<td>P5</td>
<td>117.81</td>
<td>10.23</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Static slicing in SPIN was enabled*