How Efficient are Software Verifiers for Hardware?

Authors: Rajdeep Mukherjee, Daniel Kroening, Tom Melham

FMCAD 2015
## Phases of Hardware Model Checking Tool

### DUT + Spec

<table>
<thead>
<tr>
<th>Design</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog RTL</td>
<td>SVA</td>
</tr>
</tbody>
</table>

### Design Granularity

<table>
<thead>
<tr>
<th>Bit-level Netlist</th>
<th>Word-level Netlist</th>
<th>Term-level Netlist</th>
<th>Software-Netlist</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIG, BLIF, PLA, EDIF, BAF, XNF, BENCH, BLIF-MV</td>
<td>BTOR</td>
<td>CLU</td>
<td>ANSI-C</td>
</tr>
</tbody>
</table>

### Verification Engines

Unbounded Technique

- K-induction
- Interpolation
- Abstract Interpretation
- Predicate Abstraction
- IC3/PDR

Bounded Technique

- Symbolic Execution
- BMC

### Proof Engines

**Boolean Level**

- SAT
- QBF
- PBS

**Word Level**

- SMT
- CSP
Background

- The Simple Art of SoC Design, by Michael Keating, Synopsys Fellow
  - *High-level design contains less bug than production-ready RTL design*,
  - *Scalable Verification at higher level of abstraction*
  - *Mature synthesis technology*

- Chakraborty et. al.: Word-level symbolic trajectory evaluation, CAV’15.
- Drechsler et. al.: Wolfram- A word level framework for formal verification, RSP’09.
- Kroening et. al.: Word level predicate abstraction and refinement for verifying RTL verilog, DAC’05.
- Kroening et. al.: Lifting propositional interpolants to the word-level, FMCAD’07.
- Lahiri et. al.: The UCLID decision procedure, CAV’04.
Key Message

Change in frontend of Verifier enables opportunity for efficient reasoning in the backend of the tools
Hardware Property Verification Flow

Design (Verilog) $\rightarrow$ Software-Netlist

Spec (SVA) $\rightarrow$ Word-level C Program

Analysis Engine

- BMC
- k-induction
- Predicate Abstraction
- IMPACT
- Abstract Interpretation
- IC3/PDR

Solver Engine

- SAT
- SMT

Hardware Verification using Software Analyzers
Where are we working?

- **C/C++ Algorithm**
- **C/C++ ISS**
- **C/C++ Transaction**
- **C/C++ Cycle-true**
- **Emulation**

Speed:
- 1 MHz
- 100 KHz
- 10 KHz
- 1 KHz
- 100 Hz
- 10 Hz

Clock Accuracy:
- 0%
- 100%

- RTL
module top(clk,a);

Signal declaration
input a,clk;
reg b,d,e;
wire a;

Continuous Assignment Statement
wire c = (e)? 1'b0 : d;
wire cond = a;

Always Clocked block
always @(posedge clk) begin
  b<=a;
  // non-blocking
  if(cond && b)
    e<=b;
  else
    e<=0;
  d<=c;
end
endmodule

Combinational Logic
Continuous Assignment Update
Forall t.  c(t) = if e(t) then 0 else d(t)
Forall t.  cond(t) = a(t)

Sequential Logic
Always Clocked block
Forall t.  b(t+1) = a(t)
  e(t+1) = if(cond(t) && b(t)) then b(t) else 0
  d(t+1) = c(t)
From Bits to Word to Term to Software-Netlist

Verilog

module top(Din, En, clk, Dout);
wire cs; reg ns;
input clk, Din, En;
output Dout;

// Combinational block
assign Dout = cs;
always @(Din or cs or En)
begin
if (En) ns = Din;
else ns = cs;
end
ff ff(ns, clk, cs);
endmodule

// Sequential block
module ff(Din, clk, Dout);
input clk, Din;
output Dout; reg q;
assign Dout = q;
always @(posedge clk)
q <= Din;
endmodule

Bit-level Netlist

Variable Map:
Inputs: top.clk=0, top.Din=1, top.En=2, top.ff.CLK=3, top.ff.Din=4,

Wires:
Top.Dout=11, top.ff.Dout=5, top.cs=12, top.ns=!10

Latch: top.ff.q=5

Transition constraints:
!(var(5) & !var(12)) &
!(var(5) & var(12))
!(var(4) & !(var(2) & var(1))) & !(var(4) & !var(2) & var(7)) & !(var(4))
State constraints:
- top.Dout == top.cs
- top.ff.Dout == top.ff.q
- top.ff.Din == top.ns
- top.ff.clk == top.clk
- top.ff.Dout == top.cs
- top.ns == top.En ? top.Din : top.cs
- top.Din : top.cs

Transition constraints:
- next(top.ff.q) == top.ff.Din

Software-Netlist

_Bool nondet_bool();
struct s_ff { _Bool q; };
struct s_en { _Bool ns; struct s_ff sff; } sen;

// Sequential Logic
_Bool ff(_Bool CLK, _Bool Din, _Bool *Dout) {
  _Bool q_old;
  q_old = sen.sff.q;
  sen.sff.q = Din;
  *Dout = q_old;
  return;
}

// Combinational Logic
void top(_Bool clk, _Bool Din, _Bool En, _Bool *Dout) {
  _Bool cs;
  if(En) { sen.ns = Din; }
  else { sen.ns = cs; }
  ff(clk, sen.ns, &cs);
  *Dout = cs;
}

int main() {
  _Bool clk, En, Din, out;
  while(1) {
    Din = nondet_bool(); En = nondet_bool();
    top(clk, Din, En, &out);
  }
  return;
}
Equivalence of Verilog and Software-netlist

- Output model is bit-precise and cycle accurate

- For unsafe benchmarks, bugs are manifested at the same clock cycle

- For safe benchmarks, properties are proven to be k-inductive, where values of k are same in Verilog and software-netlist model
Proposed Verification Tool Flow

Verification at Bit-level, Word-level, Software-Netlist level
Classical Abstract Interpretation Based Tools

- Suitable for proving absence of run-time errors, hence requires less precision
- But for functional verification, need more precision by data and control partitioning

```c
typedef enum {false = 0, true = 1} _Bool;
int id(int a) {
    return a;
}
void main() {
    int x,y,z;
    _Bool c;
    if (c)
        x = -1;
    else
        x = 2;
    int k = id(x);
    z = x * k;
    __ASTREE_assert((z!=-2));
}
```
Hybrid tools for Software Verification

Combines BMC+ K-induction + Abstract Interpretation
K-induction at Bit-level, Word-level, Software-Netlist level
Results

• BMC on Software-netlist is > 2X faster than BMC on bit-level netlist

• Software Interpolation (IMPACT) and hardware interpolation-based techniques have comparable times.

• Bit-level PDR did not terminate for RCU and took > 4 hrs for FIFO controller and Buffer Allocation
Ongoing Work
Conflict Driven Clause Learning Solvers

BCP + Decision $\rightarrow$ Constructs an assignment

Learning infers new clauses

Model Theoretic Search $\rightarrow$ Guides $\rightarrow$ Proof Theoretic Search

SAT Solvers: Precise but inefficient
Abstract Interpreter: Efficient but imprecise

How to make SAT solvers more efficient?
$\rightarrow$ Choose a domain that’s better suited to your problem than the Boolean constants domain!

How to make Abstract Interpretation more precise?
$\rightarrow$ Wrap them in the SAT architecture!
Abstract Conflict Driven Clause Learning

New program analysis that embeds an abstract domain inside Conflict Driven Clause Learning algorithm of modern day SAT solvers.

What is ACDCL?
From AI Point of View → ACDCL is an abstract interpreter that uses Decision and Learning to increase transformer precision

From a decision procedure perspective → ACDCL is a SAT solver for program analysis constraints. It is a strict generalisation of propositional CDCL.

<table>
<thead>
<tr>
<th>Decision Procedure</th>
<th>Abstract Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial Assignments</td>
<td>Abstract Domain</td>
</tr>
<tr>
<td>Decision</td>
<td>Restrict range of variables</td>
</tr>
<tr>
<td>Unit Rule</td>
<td>Best Abstract Clause Transformer</td>
</tr>
<tr>
<td>BCP</td>
<td>GFP Iteration</td>
</tr>
<tr>
<td>Learning</td>
<td>Generate program analysis constraint (Implicit form of Trace Partitioning)</td>
</tr>
</tbody>
</table>
Conclusion

- SoC designs are increasingly written at higher level than RTL
- Change in front-end of verifiers allows opportunity for scalable reasoning in the backend
- Verifiers must employ software-like representations of circuits
- Abstract domain needed: Bit-field domain, Interval, Octagon, Equality, Polyhedra
- Need bit-precise reasoning
- Need new solver based on ACDL
Thank You for Your Attention!

Systems Verification Group
www.cprover.org/hardware/v2c-isvlsi/
module top(clk,a);
input a,clk;
reg b,d,e;
wire a;

wire c = (e)? 1'b0 : b;
wire cond = b;

always @(posedge clk) begin
  b<=a;
  if(cond)
    e<=b;
  else
    e<=0;
  d<=c;
end
endmodule

struct state_elements_top {
  unsigned int b, e;
};
struct state_elements_top u1;
unsigned int c;
_Bool cond;

void top_ns(unsigned int clk, unsigned int a) {
    _Bool a_old, b_old, d_old, e_old;
    b_old = u1.b;
    d_old = u1.d;
    e_old = u1.e;

    b_old = u1.b;
    d_old = u1.d;
    e_old = u1.e;

    u1.b= a;
    if(cond)
      u1.e = b_old;
    else
      u1.e = 0;
    u1.d = c;
}

cond = u1.b;
c = (u1.e) ? 0 : u1.b;
Circuits with Loopback

module foo(c,a,z,y);
    input c;
    input [31:0] a;
    output [31:0] z;
    reg [31:0] y;
    output [31:0] y;

assign z[0] = c;
assign z[31:1] = a[30:0];

always@(posedge clk)
y <= z;
assert property1: ((a == z));
endmodule

type declarations
typedef unsigned int _u32;
// Module Verilog::foo
struct module_foo {
    _Bool c;
    _u32 a;
    _u32 z;
    _u32 y;
    _Bool clk;
};
// Module Verilog::top
struct module_top {
    _u32 loopback;
    _u32 y;
    _Bool data;
    struct module_foo f;
};
// top module
extern struct module_top top;

void main() {
    __CPROVER_assume(top.f.z == ((top.f.z & 0xffffffff) | (top.f.c & 0x1)));
    __CPROVER_assume(top.f.z == ((top.f.z & 0x00000001) | ((top.f.a & 0x7fffffff) << 1)));
    __CPROVER_assume(top.f.c == top.data);
    __CPROVER_assume(top.f.a == top.loopback);
    __CPROVER_assume(top.f.z == top.loopback);
    // Next state function
    __CPROVER_assume(top.f.y == top.f.z);
}
module foo(c,a,z,y);
    input c;
    input [31:0] a;
    output [31:0] z;
    reg [31:0] y;
    output [31:0] y;
    assign z[0] = c;
    assign z[31:1] = a[30:0];
    always@(posedge clk)
        y <= z;
    assert property1: ((a == z));
endmodule

module top();
    wire [31:0] loopback;
    wire [31:0] y;
    wire data;
    foo f(.c(data),.a(loopback),.z(loopback),.y());
endmodule

struct state_elements_foo{
    unsigned int y;
};
struct state_elements_foo sfoo;
void foo_nextstate(_Bool c, unsigned int a, unsigned int *z, unsigned int *y) {
    unsigned int tmp0;
    tmp0 = *z;
    sfoo.y = tmp0;
    *y = tmp0;
}
void foo_output(_Bool c, unsigned int a, unsigned int *z, unsigned int *y) {
    *z = (*z & 0xfffffffe) | (c & 0x1); // z[0] = c;
    *z = (*z & 0x00000001) | ((a & 0x7fffffff) << 1); // z[31:1] = a[30:0];
}
void main() {
    unsigned int y, loopback;
    for(int i=1;i<33;i++)
        foo_output(1, loopback, &loopback, &y);
    foo_nextstate(1, loopback, &loopback, &y);
}
Need Richer Domains like ..

Octagon Domain Analysis:

\[ \text{Intervals: } x = [1, 10], y = [0, 9] \]

Equality Domain Analysis:

\[ X == Y, X != Y \]

typedef enum {false = 0, true = 1} _Bool;
int id(int a) {
    return a;
}
void main() {
    int x,y,z;
    _Bool c;
    if (c)
        x = -1;
    else
        x = 2;
    int k = id(x);
    Intervals: x = [-1, 2], k = [-1, 2]
    Equality: (x == k)
    z = x * k;
    __ASTREE_assert((z!= -2));
}
Semantic Loop Unrolling

Program P:

```c
int main() {
    int init = 0;
    int i = 0;
    float x = 0.0, div = 0.0;
    while (i < 10) {
        if (init) {
            x += x / div;
        } else {
            init = 1;
            x = 1.0;
            div = 2.0;
        }
        i++;
    }
}
```

Unroll = 0:

- One invariant for all loop iterations:
  - \( i \in [0, 9] \); \( \text{init} \in [0, 1] \); \( \text{div} \in [0.0, 2.0] \)
- False alarm: potential runtime error

Unroll = 1:

- One invariant for first loop iterations:
  - At entry: \( i \in \{0\} \); \( \text{init} \in \{0\} \)
- One invariant for all other loop iterations:
  - \( i \in [1, 9] \); \( \text{init} = 1 \); \( \text{div} = 2.0 \)
- No alarms reported
struct state_elements_top {
    unsigned int b, d, e, c;
};
struct state_elements_top u1;

void top(unsigned int clk, unsigned int a) {
    _Bool b_old, d_old, e_old;
    _Bool cond;
    b_old = u1.b;
    d_old = u1.d;
    e_old = u1.e;
    cond = a;
    u1.c = (u1.e) ? 0 : u1.d;
    u1.b = a;
    if (cond && b_old)
        u1.e = b_old;
    else
        u1.e = 0;
    u1.d = u1.c;
}

State-holding elements
struct state_elements_top {
    unsigned int b, d, e, c;
};
struct state_elements_top u1;

void top(unsigned int clk, unsigned int a) {
    _Bool b_old, d_old, e_old;
    _Bool cond;
    b_old = u1.b;
    d_old = u1.d;
    e_old = u1.e;
    cond = a;
    u1.c = (u1.e) ? 0 : u1.d;
    u1.b = a;
    if (cond && b_old)
        u1.e = b_old;
    else
        u1.e = 0;
    u1.d = u1.c;
}
Software-Netlist Model

• A software-netlist model is defined as six tuple, 
  \[ \text{SN} = \langle \text{In}, \text{Out}, \text{Seq}, \text{Comb}, \text{Init}, \text{Asgn} \rangle, \] 
  where
  \[ \text{In, Out, Seq, Comb, Init} \] are input, output, sequential/state-holding, combinational / stateless signals and initial states respectively. \text{Asgn} is a finite set of assignments to \text{Out}, \text{Seq} and \text{Comb} where,

  • \( \text{Asgn} ::= \text{CAsgn} | \text{SAsgn} \)
  • \( \text{CAsgn} ::= (V_c = \text{bvExpr}) | (V_c = \text{bool}), \ V_c \in \text{Comb} \cup \text{Out} \)
  • \( \text{SAsgn} ::= (V_s = \text{bvExpr}) | (V_s = \text{bool}), \ V_s \in \text{Seq} \)
  • \( \text{bvExpr} ::= \text{bvconst} | \text{bvvar} | \text{ITE}(\text{cond}, \text{bv}1 \ldots \text{bv}n) | \text{bvop}(\text{bv}1 \ldots \text{bv}n), \ \text{cond} \in \text{bool}, \ \text{bvi} \in \{\text{bvconst}, \text{bvvar}\} \)
  • \( \text{bool} ::= \text{true} | \text{false} | \neg b | b1 \land b2 | b1 \lor b2 | \text{bvrel}\{b1 \ldots bn\}, (n \geq 1) \)

Cycle accurate and Bit-precise Model

Generated following synthesis semantics