Formal Verification Overview

EECS 251B
Objectives

1. Formal modeling for verification
2. Properties
   1. Temporal logic
   2. SVA
3. Property checking approaches
   1. Explicit state model checking
   2. Symbolic model checking
   3. Symbolic model checking backends
## Verification Elements

<table>
<thead>
<tr>
<th>Modeling</th>
<th>Design</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL, Gate-level, TTL</td>
<td>Symbolic Model Checking</td>
<td>Temporal Logics: SVA, PCL, etc.</td>
</tr>
</tbody>
</table>

| Testbenches | Symbolic test cases | Concrete test cases |

<table>
<thead>
<tr>
<th>Verification Methods</th>
<th>Symbolic Model Checking</th>
<th>Explicit-state Model Checking</th>
<th>Simulation-based Verification</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Engines</th>
<th>SAT/SMT Solvers, BDDs</th>
<th>Automata-based</th>
<th>Fuzzers, Monitors</th>
</tr>
</thead>
</table>
# Verification Elements

## Design
- RTL, Gate-level, TTL

## Properties
- Temporal Logics: SVA, PCL, etc.

## Testbenches
- Symbolic test cases
- Concrete test cases

## Verification Methods
- Symbolic Model Checking
- Explicit-state Model Checking
- Simulation-based Verification

## Engines
- SAT/SMT Solvers, BDDs
- Automata-based
- Fuzzers, Monitors

---

Lecture 18

---

4/18/2023
## Verification Elements

<table>
<thead>
<tr>
<th>Modeling</th>
<th>Design</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTL, Gate-level, TTL</td>
<td>EECS 251B</td>
<td>Temporal Logics: SVA, PCL, etc.</td>
</tr>
</tbody>
</table>

### Testbenches
- Symbolic test cases
- Concrete test cases

### Verification Methods
- Symbolic Model Checking
- Explicit-state Model Checking
- Simulation-based Verification

### Engines
- SAT/SMT Solvers, BDDs
- Automata-based
- Fuzzers, Monitors

Lecture 18
Modeling for Verification
Model

• *Formal description* of the design exposed to verification/test
Model: a transition system

- Formal description of the design exposed to verification/test

- A transition system: $M = (S, S_0, T)$
- $S$: the set of states
- $S_0$: the set of initial states
- $T$: the transition relation
  - $T$ is a relation over states
  - $(s_1, s_2)$ in $T$ if model allows transitioning from $s_1$ to $s_2$
A transition system

Model of the design: \( M = (S, S_0, T) \)
- \( S \): the set of states
- \( S_0 \): the set of initial states
- \( T \): the transition relation

module counter (    
  input clk    
); 

    reg [2:0] count; 

    initial begin 
        count = 0;
    end

    always @(posedge clk) 
        count <= count + 1;

endmodule
module counter (input clk);

    reg [2:0] count;

    initial begin
        count = 0;
    end

    always @(posedge clk)
        count <= count + 1;

endmodule
A transition system with outputs

Model of the design: $M = (S, S_0, T, L)$
- $S$: the set of states
- $S_0$: the set of initial states
- $T$: the transition relation
- $L$: labels
  - \{wrap = 0, wrap = 1\}

```verilog
module counter (input clk, output wrap);
  reg [2:0] count;
  initial begin
    count = 0;
  end
  assign wrap = count == 7;
  always @(posedge clk)
    count <= count + 1;
endmodule
```
TS as a graph Moore machine

module counter ( 
    input clk, 
    output wrap); 

    reg [2:0] count; 

    initial begin 
        count = 0; 
        end 

    assign wrap = 
        counter == 7; 

    always @(posedge clk) 
        count <= count + 1; 

endmodule
TS with outputs and inputs

Model of the design: $M = (S, S_0, T, L, I)$
- $S$: the set of states
- $S_0$: the set of initial states
- $T$: the transition relation
- $L$: labels
- $I$: inputs
  - $\{\text{step} = 0, \text{step} = 1\}$

```verilog
module counter (  
    input clk,  
    input step,  
    output wrap);  
  reg [2:0] count;  
  initial begin  
    count = 0;  
  end  
  assign wrap =  
    count == 7;  
  always @(posedge clk)  
    if (step)  
      count <= count + 1;  
endmodule
```
module counter(
    input clk,
    input step,
    output wrap);

reg [2:0] count;

initial begin
    count = 0;
end

assign wrap =
    count == 7;

always @(posedge clk)
    if (step)
        count <= count + 1;

endmodule
Executions/Runs

Model of the design: \( M = (S, S_0, T) \)

Execution of \( M \) is a sequence of states

\[
q_0, q_1, q_2, q_3, q_4, \ldots \text{ such that:}
\]

\begin{itemize}
  \item \( q_0 \) is an initial state from \( S_0 \)
  \item For each \( i \), \((q_i, q_{i+1})\) is a valid transition from \( T \)
\end{itemize}
Executions/Runs

Model of the design: $M = (S, S_0, T)$

Execution of $M$ is a sequence of states $q_0, q_1, q_2, q_3, q_4, \ldots$ such that:

- $q_0$ is an initial state from $S_0$
- For each $i$, $(q_i, q_{i+1})$ is a valid transition from $T$

All traces: $S^*$
Properties
SVA: System Verilog Assertions

• A language fragment for specifying properties
• Accepted by many tools:
  • Jasper (Cadence)
  • VCS, VC Formal (Synopsys)
  • Questa (MentorGraphics)
  • SymbiYosys
  • ...
SVA: Examples

• Immediate assertions (nested within SV code):
  ```
  if (Req)
    assert (Resp);
  ```

• Concurrent assertions:
  ```
  assert property (@(posedge clk) Req |-&gt; Resp)
  
  assert property (@(posedge clk) Req |=&gt; Resp)
  
  assert property (@(posedge clk) Req |-&gt; #[t1:t2] Resp)
  ```
Temporal Properties

• A temporal property $P$ is a constraint on traces

• Partitions executions by satisfaction w.r.t. $P$
Temporal Properties: Examples

**P1** = if the wrap bit is high *then* it will be low for *next two* cycles

- Example traces?

```verilog
module counter (input clk, output wrap);
    reg [2:0] count;
    initial begin
        count = 0;
    end
    assign wrap = count == 7;
    always @(posedge clk)
        count <= count + 1;
endmodule
```
Temporal Properties: Examples

P1 = if the wrap bit is high then it will be low for next two cycles

P2 = the wrap bit is always eventually high

• Example traces?

```verilog
module counter (input clk, output wrap);
    reg [2:0] count;
    initial begin
        count = 0;
    end
    assign wrap = count == 7;
    always @(posedge clk)
        count <= count + 1;
endmodule
```
Temporal Logic

• A logic to specify properties over traces
  (reasoning system) (temporality)

• Many flavours:
  • Propositional temporal logic: atoms are plain Boolean variables
  • Signal temporal logic: atoms defined over real-valued signals
  • Metric temporal logic: quantitative constraints over time
  • Other variants: CTL, CTL*

We’ll focus on LTL (Linear Temporal Logic)
Atomic State Property

- Atomic property (*atom*) holds in a single state
- Forms the base (leaf) expression in a temporal logic formula

Examples:

1. Request input to the cache is high \(\text{(cache.req_valid_i)}\)
2. Buffer is empty \(\text{(buf_empty)}\)
LTL Formulae

- LTL formulae are built from sub-formulae, with atoms at the leaves
- Form := Atom | X Form | G Form | F Form | Form1 U Form1
  | Form1 && Form2 | ...
LTL Formulae

- LTL formulae are built from sub-formulae, with atoms at the leaves
- Form := Atom | X Form | G Form | F Form | Form1 U Form1
  | Form1 && Form2 | ...

Example: Let p be an atomic state property (e.g. req_1 is high). Then G p means p holds at all points along the trace (always p).
**LTL: Linear Temporal Logic**

<table>
<thead>
<tr>
<th>&lt;Formula&gt;</th>
<th>Meaning</th>
<th>Visually</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;pred&gt;</td>
<td><strong>Atom:</strong> Predicate &lt;pred&gt; holds at current step</td>
<td>?</td>
</tr>
<tr>
<td>X &lt;Formula&gt;</td>
<td><strong>Next:</strong> &lt;Formula&gt; holds at next step</td>
<td>?</td>
</tr>
<tr>
<td>F &lt;Formula&gt;</td>
<td><strong>Future/eventually:</strong> &lt;Formula&gt; holds at some point in the future</td>
<td>?</td>
</tr>
<tr>
<td>G &lt;Formula&gt;</td>
<td><strong>Globally:</strong> &lt;Formula&gt; holds on all suffixes of the trace</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>&lt;F1&gt; U &lt;F2&gt;</td>
<td><strong>Until:</strong> &lt;F1&gt; holds on all suffixes of the trace until &lt;F2&gt; is true</td>
<td>?</td>
</tr>
<tr>
<td>&amp;,&amp;,</td>
<td></td>
<td>, == &gt;</td>
</tr>
</tbody>
</table>
**SVA semantics as LTL**

How can you write the following SVA properties as LTL?

```
assert property (@(posedge clk) Req => #[1:2] Resp)
```

```
if (core_req_valid)
    if (is_store(core_req))
        assert (!store_buf_full || mem_req_gnt);
```
The overall picture

module counter (  
    input clk,  
    output wrap);  
  reg [2:0] count;  
  initial begin ...  
endmodule

DUT

SVA
always (@(posedge clk)  
(Req |-> Resp))

Formal
Model (M)

Formal Prop. P  
(e.g. in LTL)  
G(Req => X(Resp))
Verification Approaches and Engines
The verification problem

All traces: $S^*$

Don’t satisfy $P$

Executions of $M$

Satisfy $P$
Simulation driven verification

For a **DUT** and property **P**, try different inputs with the goal of **falsifying P**.
Model checking
Model checking problem

Do *all executions* of M satisfy P?
Model checking problem

Do all executions of M satisfy P?

\[ P = \text{wrap is not true in first 6 steps} \]
Model checking problem

Do all executions of $M$ satisfy $P$?

$P = \text{wrap is not true in first 6 steps}$

trace0: $count = 0, 0, 0, 0, 0, 0, 0, 0$
trace1: $count = 0, 1, 2, 2, 3, 4, 4, 5, 6$
trace2: $count = 0, 1, 1, 1, 2, 2, 2, 3, 3$

...
Model checking problem

Bounded Model Checking (BMC)
Do all executions of $M$ of length $d$ satisfy $P$?
Bounded model checking problem

Do all executions of M of length $d$ satisfy P?

Approach 1: try all traces individually!!
- $N^d$ traces

module counter (  
    input clk,  
    input step,  
    output wrap);  
  
  reg [2:0] count;  
  
  initial begin  
    count = 0;  
    end  
  
  assign wrap =  
    count == 7;  
  
  always @(posedge clk)  
    if (step)  
      count <= count + 1;  

endmodule
Bounded model checking problem

Do all executions of M of length \(d\) satisfy \(P\)?

Approach 1: try all traces individually!!
- \(N^d\) traces

Approach 2: use automata-theoretic techniques on underlying transition system
- explicit state model checking

```verilog
module counter (
  input clk,
  input step,
  output reg) counter (input clk, input step, output wrap);
    reg [2:0] count;

initial begin
  count = 0;
end

assign wrap = count == 7;

always @(posedge clk)
  if (step)
    count <= count + 1;
endmodule
```
Bounded model checking problem

Do all executions of $M$ of length $d$ satisfy $P$?

Approach 1: try all traces individually!!
- $N^d$ traces

Approach 2: use automata-theoretic techniques on underlying transition system
- explicit state model checking

Approach 3: reason over several traces together
- symbolic model checking

```verilog
module counter (  
    input clk,  
    input step,  
    output wrap);
  
  reg [2:0] count;
  
  initial begin  
    count = 0;  
  end
  
  assign wrap =  
    count == 7;

  always @(posedge clk)  
    if (step)  
      count <= count + 1;

endmodule
```
Symbolic model checking
Symbolic model checking

Traces are over individual states

**trace1:** count = 0, 1, 1, 2, 3, 4, ..

**trace2:** count = 0, 1, 1, 1, 1, 1, ..

Symbolic traces are over *sets of states*

**st1:** \{count = 0, 1\}, \{count = 0, 1, 2\}, ...

```verilog
module counter (    
    input clk,    
    input step,   
    output wrap);  
  reg [2:0] count;  
  initial begin  
    count = 0;  
  end  
  assign wrap =  
    count == 7;  
  always @(posedge clk)  
    if (step)  
      count <= count + 1;  
endmodule
```
## Idea 1A: Representing a set of states

<table>
<thead>
<tr>
<th>Set of states</th>
<th>Symbolic representation $E(V)$ (as a formula over variables V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{count = 0, 1, 2, 3}</td>
<td>count[MSB:2] == 0</td>
</tr>
<tr>
<td>{count = 0, 2, 4, 6, 8, ...}</td>
<td>!count[0]</td>
</tr>
<tr>
<td>{(count = 1, flag = 1),</td>
<td>count == flag</td>
</tr>
<tr>
<td>(count = 0, flag = 0)}</td>
<td></td>
</tr>
</tbody>
</table>
Operations over sets of states

$S_1$ and $S_2$ are sets of states rep. by formulae $E_1(V)$ and $E_2(V)$.

Then:

• $S_1 \cup S_2$ is represented by $E_1(V) \lor E_2(V)$

• $S_1 \cap S_2$ is represented by ____?

• $\overline{S_1}$ is represented by ____?
Set of initial states

Model of the design: $M = (S, S_0, T, L)$

$S_0 = \{ \text{count} = 0 \}$
- $I(V)$: $\text{count} == 0$ (kind of silly)

More interesting examples:
$S_0 = \{ \text{count} = 0, \text{count} = 4 \}$
- $I(V)$: $\text{count}[1:0] == 0$

$S_0 = \{(a = 0, b = 0), (a = 2, b = 2), (a = 4, b = 4), ...\}$
- $I(V)$: $a = b \land \neg a[0]$
Idea 1B: Representing traces over sets of states

Representing one step of $T$

$T = \{(\text{count} = 0, \text{count'} = 1), (\text{count} = 1, \text{count'} = 2), \ldots, (\text{count} = 7, \text{count'} = 0)\}$

Two copies of the variables: pre-copy $V$ and post-copy $V'$

Find formula $T(V, V')$ representing all possible transitions?

Model $M = (S, S_0, T, L)$
Idea 1B: Representing traces over sets of states

Representing **one** step of $T$

$T = \{(\text{count} = 0, \text{count'} = 1), (\text{count} = 1, \text{count'} = 2), \ldots, (\text{count} = 7, \text{count'} = 0)\}$

Two copies of the variables: pre-copy $V$ and post-copy $V'$

Find formula $T(V, V')$ representing all possible transitions?

$T(V, V')$: $\text{count'} = (\text{count} + 1) \mod 8$

Model $M = (S, S_0, T, L)$
Idea 1: Representing all possible traces of $M$

Let $V_0, V_1, V_2, \ldots$ be fresh copies of variables in the model.

Some questions:

Q1. What does $I(V_0)$ represent?
Q2. What does $I(V_0) \land T(V_0, V_1)$ represent?
Q3. What does $I(V_0) \land T(V_0, V_1) \land T(V_1, V_2)$ represent?

Q_d+1. What does $I(V_0) \land T(V_0, V_1) \land \ldots \land T(V_{d-1}, V_d)$ represent?
Idea 1: Representing all possible traces of M

Let \( V_0, V_1, V_2, \ldots \) be fresh copies of variables in the model.

Some questions:
Q1. What does \( I(V_0) \) represent?
Q2. What does \( I(V_0) \land T(V_0, V_1) \) represent?
Q3. What does \( I(V_0) \land T(V_0, V_1) \land T(V_1, V_2) \) represent?
Qd+1. What does \( I(V_0) \land T(V_0, V_1) \land \ldots \land T(V_{d-1}, V_d) \) represent?

All traces of length \( d!! \)
Idea 2: Representing the property

Suppose we have an LTL property $G p$ (e.g. $G (\text{req} \implies \text{resp})$)

Recall: this means that $p$ always holds.

- Represent $p$ as a Boolean formula $E_p(V)$ over the variables $V$!!
- Let $V_0, V_1, V_2, \ldots$ be fresh copies of variables in the model representing the values taken at step 0, 1, 2, ...
- Then $G p = ?$
Idea 2: Representing the property

Suppose we have an LTL property $G \ p$ (e.g. $G (\text{req} \Rightarrow \text{resp})$)

Recall: this means that $p$ always holds.

- Represent $p$ as a Boolean formula $E_p(V)$ over the variables $V!!$
- Let $V_0, V_1, V_2, \ldots$ be fresh copies of variables in the model representing the values taken at step 0, 1, 2, ...
- Then $G \ p = ?$

$$E_p(V_0) \land E_p(V_1) \land E_p(V_2) \land \cdots \land E_p(V_d)$$
Idea1+2: Symbolic model checking

Formal Model (M)

**Φ_M**

\[ I(V_0) \land T(V_0, V_1) \land \ldots T(V_{d-1}, V_d) \]

Formal Prop. P (e.g. in LTL)

**Φ_P**

\[ E_P(V_0) \land E_P(V_1) \land \ldots E_P(V_d) \]

G(Req => X(Resp))
Idea 1+2: Symbolic model checking

Formal Model (M)

\[ \Phi_M = I(V_0) \land T(V_0, V_1) \land \ldots T(V_{d-1}, V_d) \]

Formal Prop. P (e.g. in LTL)

\[ \Phi_P = E_P(V_0) \land E_P(V_1) \land \ldots E_P(V_d) \]

Is \( \Phi_M \land \neg \Phi_P \) satisfiable?
Symbolic MC Backends
Boolean (SAT)isfiability Problem

Given:
- A boolean formula: \( F(x_0, x_1, ..., x_n) \)

Find:
- Whether there exist boolean \( v_0, v_1, ..., v_n \) such that \( F(v_0, v_1, ..., v_n) \) evaluates to TRUE
- The values \( v_0, v_1, ..., v_n \) satisfying \( F \)
SAT: An example

\[ F(x_0, x_1, x_2) = (x_0 \text{ OR } x_1) \text{ AND } (x_2 \text{ OR } \neg x_0) \]

- Is \( F \) Satisfiable?

- What is a satisfying assignment?
SAT Solving

- Canonical NP-hard problem. However, fares well in practice. *Why?*

- Worst case vs. practical/industrial benchmarks
- Solvers have gotten better with time
  - Better algorithms
    - CDCL: Conflict Driven Clause Learning
    - BCP: Boolean Constraint Propagation
  - Better data structures
    - Implication graph
  - Heuristics
SAT Solvers

SAT Competition Winners on the SC2020 Benchmark Suite

- kissat-2020
- maple-lcm-disc-cb-di-v3-2019
- maple-lcm-dist-cb-2018
- maple-lcm-dist-2017
- maple-comps-drup-2016
- lingeling-2014
- abcdsat-2015
- lingeling-2013
- glucose-2012
- glucose-2011
- cryptominisat-2010
- precosat-2009
- minisat-2008
- berkmim-2003
- minisat-2006
- rsat-2007
- satelite-gti-2005
- zchaff-2004
- lmmat-2002
Satisfiability Modulo Theories (SMT)

Given:
- A boolean formula: $F(x_0: T_0, x_1: T_1, \ldots, x_n: T_n)$ where $x_i$ has type $T_i$

Find:
- Whether there exist boolean $v_0, \ldots$ of types $T_0, \ldots$ such that $F(v_0, v_1, \ldots, v_n)$ evaluates to TRUE
- The values $v_0, v_1, \ldots, v_n$ satisfying $F$
SMT: An example

Variables: \( (a: \text{bv64}), (b: \text{bv64}), (c: \text{bv64}) \)

\[ F(a, b, c) = (a + a = c) \land (a \times 2 = b) \land (b \neq c) \]

- Is \( F \) Satisfiable?

- What is a satisfying assignment?
**SAT vs. SMT**

\[ F(a: \text{bv}K, b: \text{bv}K, c: \text{bv}K) = (a + a = c) \text{ AND } (a \ll 2 = b) \text{ AND } (b \neq c) \]

- **K = 64**
- **K = 256**
- **K = 1024**
- **K = 4096**

Milliseconds (log scale)
SAT/SMT Solvers

• Z3: [https://github.com/Z3Prover/z3](https://github.com/Z3Prover/z3)
• CVC5: [https://github.com/cvc5/cvc5](https://github.com/cvc5/cvc5)
• Yices: [https://yices.csl.sri.com/](https://yices.csl.sri.com/)

• Some solvers are specialized for particular theories:
  • Boolector (bitvectors + arrays + UFs):
    • [https://github.com/Boolector/boolector](https://github.com/Boolector/boolector)
  • SMTInterpol (linear real + integer arithmetic):
    • [http://ultimate.informatik.uni-freiburg.de/smtinterpol/](http://ultimate.informatik.uni-freiburg.de/smtinterpol/)
  • STP (bitvectors + arrays):
    • [https://stp.github.io/](https://stp.github.io/)
BDD: Binary Decision Diagrams

- Representation of a Boolean formula
BDDs: Tooling

- Many BDD libraries:
  - JavaBDD: [https://javabdd.sourceforge.net/](https://javabdd.sourceforge.net/)
  - pytest-BDD: [https://github.com/pytest-dev/pytest-bdd](https://github.com/pytest-dev/pytest-bdd)
BDDs: Industrial Application

```prolog
File

; VARS "q[2:0] r[2:0];
q::bool list
r::bool list
; // Force creation
q fseq r fseq ();
; // Order the variables sequentially
var_order ((md_expand_vector "q[2:0]"")@md_expand_vector "r[2:0]"));
["q[2:0]","r[2:0]","a","b","c[3:0]","d[1:0] [2:0]","q","v"]
it::string list
draw_bdds T [q = r];
; // Order the variables by interleaving them MSB to LSB
var_order (interleave [(md_expand_vector "q[3:0]"),
    (md_expand_vector "r[3:0]"))])
;
["q[3]","r[3]","q[2]","r[2]","q[1]","r[1]","q[0]","r[0]","a","b","c[3:0]","d[1:0][2:0]","q","v"]
it::string list
draw_bdds T [q = r];
;
Interrupt

Help
```
BDDs: Industrial Application

```plaintext
: VARS "q[2:0] r[2:0]";
q::bool list
r::bool list
: // Force creation
q fseq r fseq ();
: // Order the variables sequentially
var_order ((md_expand_vector "q[2:0]"))@((md_expanse ["q[2:0]","r[2:0]","a","b","c[3:0]","d[1:0] [2:0]
it::string list
draw_bdds T [q = r];
: // Order the variables by interleaving them
var_order (interleave ((md_expand_vector "q[3:0]"
(md_expand_vector "r[3:0]"));
[q[3],"r[3],"q[2],"r[2],"q[1],"r[1],"q[0] [2:0],"q[0],"q[0]"
it::string list
draw_bdds T [q = r];
```

4/18/2023
BDDs: Industrial Application

https://github.com/TeamVoss/VossII
Closing remarks

Abstract models, Modelling DSLs

4/18/2023

4/18/2023
Additional reading

• Model Checking by Clarke et al. MIT Press

• Handbook of Model Checking

  • http://www.eecs.berkeley.edu/~sseshia/pubdir/SMT-BookChapter2e.pdf