Abstraction Refinement in Model Checking with Applications to C and Verilog

Edmund Clarke
Carnegie Mellon University
Model Checking and Temporal Logic

• Models are **State Transition Graphs** or **Kripke Structures**

• Obtain Kripke structure K from hardware/software design

• Verification
  - Formula = Specification
  - Evaluate varying formulas over K
Feasibility of Model Checking

Central Problem in Model Checking:

State Explosion Problem

State space too large for explicit construction.
Even worse for software.
Talk Outline

1. Review of model checking.
2. Counterexample Guided Abstraction Refinement.
3. SAT-Based Predicate Abstraction
4. Application to RTL-Level Verilog
LTL - Linear Time Logic

Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

\( a \)       “a is true now”
\( X a \)     “a is true in the neXt state”
\( Fa \)      “a will be true in the Future”
\( Ga \)      “a will be Globally true in the future”
\( a \cup b \) “a will hold true Until b becomes true”
Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

\[
\begin{align*}
\text{a} & : \text{“a is true now”} \\
\text{X a} & : \text{“a is true in the next state”} \\
\text{Fa} & : \text{“a will be true in the Future”} \\
\text{Ga} & : \text{“a will be Globally true in the future”} \\
\text{a U b} & : \text{“a will hold true Until b becomes true”}
\end{align*}
\]
Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

\[ a \] \quad \text{“a is true now”}
\[ X a \] \quad \text{“a is true in the next state”}
\[ Fa \] \quad \text{“a will be true in the Future”}
\[ Ga \] \quad \text{“a will be Globally true in the future”}
\[ a \mathbin{U} b \] \quad \text{“a will hold true Until b becomes true”}
LTL - Linear Time Logic

Determines Patterns on Infinite Traces

Atomic Propositions

Boolean Operations

Temporal operators

a

“a is true now”

X a

“a is true in the neXt state”

Fa

“a will be true in the Future”

Ga

“a will be Globally true in the future”

a U b

“a will hold true Until b becomes true”
Determines Patterns on Infinite Traces

Atomic Propositions
Boolean Operations
Temporal operators

a → a → a → a → b

a “a is true now”
X a “a is true in the neXt state”
Fa “a will be true in the Future”
Ga “a will be Globally true in the future”
a U b “a will hold true Until b becomes true”
Branching Time
CTL: Computation Tree Logic

EF g  “g will possibly become true”
CTL: Computation Tree Logic

AF $g$  “$g$ will necessarily become true”
CTL: Computation Tree Logic

AG $g$  "g is an invariant"
EG $g$  "g is a potential invariant"
CTL: Computation Tree Logic

CTL uses the temporal operators

AX, AG, AF, AU
EX, EG, EF, EU

CTL* allows complex nestings such as
AXX, AGX, EXF, ...
Model Checking

Hardware Description (VERILOG, VHDL, SMV)

Informal Specification

Transition System (Automaton, Kripke structure)

Temporal Logic Formula (CTL, LTL, etc.)

compilation

manual

algorithmic verification
Counterexamples

Hardware Description
(VERILOG, VHDL, SMV)

Informal Specification

Temporal Logic Formula
(CTL, LTL, etc.)

Safety Property:
bad state \text{unreachable:}
satisfied

Initial State
Counterexamples

Hardware Description (VERILOG, VHDL, SMV)

Transition System

Informal Specification

Temporal Logic Formula (CTL, LTL, etc.)

Safety Property: bad state unreachable

Counterexample
Counterexamples

Hardware Description (VERILOG, VHDL, SMV)

Transition System

Informal Specification

Temporal Logic Formula (CTL, LTL, etc.)

Safety Property:
bad state unreachable

Counterexample
Counterexample generation practically important.

Clarke, Grumberg, McMillan, Zhao 1995
Symbolic Algorithms for Trace Counterexamples
Counterexamples

Counterexample generation practically important.

Clarke, Grumberg, McMillan, Zhao 1995
Symbolic Algorithms for Trace Counterexamples
Talk Outline

1. Review of model checking.
2. Counterexample Guided Abstraction Refinement.
3. SAT-Based Predicate Abstraction
4. Application to RTL-Level Verilog
Given an abstraction function $h : S \rightarrow S_h$, the concrete states are grouped and mapped into abstract states:

**Existential Abstraction**

Preservation Theorem?
If M has partial behavior of N, we say that “N simulates M”: $M \leq N$
Preservation Theorem

\[ \text{ACTL} = \text{universal fragment of CTL} \]
\[ \text{(AX, AG, AF, AU, atomic negation)} \]

\[ \text{ACTL}^* = \text{universal fragment of CTL}^* \]

*Theorem* (Clarke et al)  Let \( f \) be an ACTL specification.

\[ \text{If } M \leq N \text{ and } N \models f \text{ then } M \models f. \]
If M has partial behavior of N, we say that
“N simulates M”: \( M \preceq N \)
Preservation Theorem

Atomic formula $f$ respects $h$ if $f$ does not distinguish concrete states within abstract state.

**Theorem** (Clarke, Grumberg, Long)
If $\varphi$ is an ACTL* specification where the atomic formulas respect $h$, then $M \leq M_h$.

**Corollary** Preservation Theorem applicable:

$M_h \models \varphi$ implies $M \models \varphi$.

Converse implication is not valid!
Spurious Behavior

AGAF red
“Every path necessarily leads back to red.”

Spurious Counterexample:
<go><go><go><go> ... Artifact of the abstraction!
How to define

Abstraction Functions?

Abstraction too fine

⇒ State Explosion

Abstraction too coarse

⇒ Information Loss

Automatic Abstraction Methodology
Automatic Abstraction

$M_h$

Spurious

Spurious counterexample

Validation or Counterexample

$M$

Initial Abstraction

Refinement

Refinement

Correct!

Original Model
Counterexample-Guided Abstraction Refinement

- Generate initial abstraction
- Model check
- Generate counterexample $T_h$
- Check if counterexample is spurious
- Refinement: $T_h$ is spurious
- $M_h \models \varphi$
- $M_h \not\models \varphi$
- Stop: $T_h$ is not spurious

CEGAR
Counterexample-Guided Abstraction Refinement
Counterexample-Guided Abstraction Refinement

- **Predicate abstraction**
  - $M_h \models \varphi$
  - $M_h \notmodels \varphi$

- **Model check**
  - **Generate counterexample** $T_h$

- **Refinement**
  - New predicates

- **Check if counterexample spurious**
  - $T_h$ is spurious
  - $T_h$ is not spurious

- **Stop**
1. Review of model checking.
2. Counterexample Guided Abstraction Refinement.
3. SAT-Based Predicate Abstraction
4. Application to RTL-Level Verilog
Predicate Abstraction

- Use **predicate abstraction** to prove assertions or safety properties

- Successfully applied for verification of **C programs** (SLAM, MAGIC, BLAST)
  - Recent application: **Verification of RTL-level Verilog**

- Predicate abstraction produces over-approximation
Example for Predicate Abstraction

C program

```c
void main() {
    int i;
    i=0;
    while(even(i))
        i++;
}
```

Predicates

```plaintext
p_1 ↔ i=0
p_2 ↔ even(i)
```

Boolean program

```c
void main() {
    bool p1, p2;
    p1=TRUE;
    p2=TRUE;
    while(p2)
    {
        p1=p1?FALSE:nondet();
        p2=!p2;
    }
}
```

[Graf, Saidi '97]
[Ball, Rajamani '00]
Using theorem prover for abstraction

Predicates

\[ p_1 \iff i = 0 \]
\[ p_2 \iff \text{even}(i) \]

Basic Block

\[ i++; \]

Formula

\[ i' = i + 1 \]

Query

\[ i \neq 0 \land \overline{\text{even}(i)} \land \]
\[ i' = i + 1 \land \]
\[ i' \neq 0 \land \overline{\text{even}(i')} \]
Using theorem prover for abstraction

**Predicates**

\[ p_1 \iff i = 0 \]
\[ p_2 \iff \text{even}(i) \]

**Basic Block**

\[ i++; \]

**Formula**

\[ i' = i + 1 \]

**Query**

\[ i \neq 0 \land \overline{\text{even}(i)} \land \\
   i' = i + 1 \land \\
   i' \neq 0 \land \text{even}(i') \]

... and so on ...
Problems with existing tools

- Large number of expensive theorem prover calls – slow ($2^n \leq 2^n$)

- Theorem prover works on natural numbers, but Verilog uses bit-vectors $\Rightarrow$ false positives

- Most theorem provers support only few operators (+, -, $<$, $\leq$, …), no bitwise operators
Abstraction of a basic block

- Use a SAT solver for computing abstraction of a basic block
- Successfully used for abstraction of C programs
- New application: RTL-Level Verilog
- Create a SAT instance which relates:
  - Initial value of predicates
  - Basic block
  - Final value of predicates
Abstraction of a basic block

\[ p_1 \iff i = 0 \]
\[ p_2 \iff \text{even}(i) \]

\[ i' = i + 1 \]

\[ p'_1 \iff i' = 0 \]
\[ p'_2 \iff \text{even}(i') \]

\[ \exists i, i' : (p_1 \iff i = 0) \land (p_2 \iff \text{even}(i)) \land i' = i + 1 \land (p'_1 \iff i' = 0) \land (p'_2 \iff \text{even}(i')) \]

Computing abstract transitions
Abstract transitions

\[(p_1 \Leftrightarrow i = 0) \land (p_2 \Leftrightarrow \text{even}(i)) \land \]
\[i' = i + 1 \land \]
\[(p'_1 \Leftrightarrow i' = 0) \land (p'_2 \Leftrightarrow \text{even}(i'))\]

Equation passed to the SAT solver

Satisfying assignments

\[(!p_1 \& !p_2 \& !p'_1 \& p'_2) \quad 00 \) \quad 01\]

\[(!p_1 \& !p_2 \& p'_1 \& p'_2) \quad 00 \) \quad 11\]

...........................and so on
Use SAT solver!

1. Generate query equation with predicates as free variables

2. Transform equation into CNF using Bit Vector Logic

One satisfying assignment matches one abstract transition

3. Obtain all satisfying assignments = most precise abstract transition relation
Abstraction of a basic block

Use SAT solver!

1. Generate query equation with predicates as free variables

2. Transform equation into CNF using Bit Vector Logic

   One satisfying assignment matches one abstract transition

3. Obtain all satisfying assignments = most precise abstract transition relation
Advantages of using SAT

Use SAT solver!
1. Generate query equation with predicates as free variables
2. Transform equation into CNF using Bit Vector Logic
   One satisfying assignment matches one abstract transition
3. Obtain all satisfying assignments = most precise abstract transition relation

This solves two problems:
1. Now can do all C and Verilog operators, including *, /, %, <<, @, [x:y] etc.
2. Bit vector semantics taken

No more unnecessary spurious counterexamples!
1. Review of model checking.
2. Counterexample Guided Abstraction Refinement.
3. SAT-Based Predicate Abstraction
4. Application to RTL-Level Verilog
Application to RTL-Level Verilog

module main (clk)
input clk;
reg [10:0] x, y;
initial x = 100, y = 200;
always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Verilog program

Property:
AG (x = 100 or x = 200)

Initial set of predicates:
{x = 100, x = 200}

Word level predicates
Abstracting Transition Relation

Initial state

\[ x = 100, \ x = 200 \]

Transition Relation

\[ x' = y \]
\[ y' = x \]

Final state

\[ x' = 100, \ x' = 200 \]

\[ \exists x, y, x', y' : \]
\[ (p_1 \iff x = 100) \land (p_2 \iff x = 200) \land \]
\[ x' = y \land y' = x \land \]
\[ (p'_1 \iff x' = 100) \land (p'_2 \iff x' = 200) \]

Equation passed to the SAT solver

Computing abstract transitions
Obtain transitions

\[ \exists x, y, x', y' : \]
\[ (p_1 \iff x = 100) \land (p_2 \iff x = 200) \land \]
\[ x' = y \land y' = x \land \]
\[ (p'_1 \iff x' = 100) \land (p'_2 \iff x' = 200) \]

Computing abstract transitions

... and so on ...
Abstract Model

module main (clk)
input clk;
reg [10:0] x, y;
initial x= 100, y= 200;
always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Verilog program

Property:
AG (x = 100 or x = 200)

Initial set of predicates:
{x = 100, x = 200}

Initial state

Failure state
Counterexample-Guided Abstraction Refinement

CEGAR

Predicate abstraction

refinement new predicates

model check

generate counterexample $T_h$

check if counterexample spurious

$M_h \models \varphi$

$M_h \not\models \varphi$

$T_h$ is not spurious

$T_h$ is spurious

stop
Model checking

module main (clk)
input clk;
reg [10:0] x, y;

initial x= 100, y= 200;

always @ (posedge clk)
begin
 x <= y;
 y <= x;
end
endmodule

Verilog program

Abstract Model

Initial state

Failure state
Model checking

module main (clk)
input clk;
reg [10:0] x, y;

initial x = 100, y = 200;

always @(posedge clk)
begin
  x <= y;
y <= x;
end
endmodule

Verilog program

Abstract Model

Counterexample

Initial state

Failure state
Counterexample-Guided Abstraction Refinement

CEGAR

Predicate abstraction

model check

generate counterexample $T_h$

check if counterexample spurious

$M_h \models \varphi$

$M_h \not\models \varphi$

refinement new predicates

$T_h$ is spurious

$T_h$ is not spurious

stop

$C$, SpecC, Verilog
Simulation of the counterexample

- Counterexample in the abstract model
  - $[1, 0] \rightarrow [0, 0]$ (length = 1)
  - Each state is a valuation of $h \ x = 100, \ x=200 i$

Simulation equation

\[(x = 100) \land (y = 200) \land (x = 100) \land \neg (x = 200) \land (x' = y) \land (y' = x) \land \neg (x' = 100) \land \neg (x' = 200)\]

Initial values of the registers

Predicate values in the first state of the counterexample

Transition relation

Predicate values in the second state of the counterexample

So counterexample is spurious
Counterexample-Guided Abstraction Refinement

$C$, Spec$C$, Verilog

Predicate abstraction

$M_h \models \phi$

Model check

$M_h \not\models \phi$

Generate counterexample $T_h$

Check if counter-example spurious

$T_h$ is not spurious

Refinement

New predicates

$T_h$ is spurious

Stop
Refinement

• Let length of spurious counterexample be $k$

• Take \textit{weakest pre-condition of property} for $k$ steps with respect to transition functions

• Pick atomic predicates from weakest precondition
Refinement

Transition functions

\[ x' = y \]
\[ y' = x \]

New predicates

\[ y = 100, y = 200 \]

AG \[ (y = 100 \land y = 200) \]

\[ x' = y \]
\[ y' = x \]

\[ (y = 100 \land y = 200) \]

spurious counterexample

length = 1

weakest precondition

Holds after one step

\[ (x' = 100 \land x' = 200) \]
Abstract again

module main (clk)
input clk;
reg [10:0] x, y;
initial x= 100, y= 200;
always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Verilog program

Property:
AG (x = 100 or x = 200)

Updated set of predicates:
{x = 100, x = 200, y=100, y=200}

New abstraction

Initial state 1001
Model check

0110
Model checking

module main (clk)
input clk;
reg [10:0] x, y;
initial x = 100, y = 200;
always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Verilog program

Property:
AG (x = 100 or x = 200)

Updated set of predicates:
{x = 100, x = 200, y=100, y=200}

New abstraction

Initial state → 1001 → 0110 → Property holds!
Result

module main (clk)
input clk;
reg [10:0] x, y;
initial x = 100, y = 200;
always @ (posedge clk)
begin
  x <= y;
  y <= x;
end
endmodule

Verilog program

Property:
AG (x = 100 or x = 200)

Property holds!
Handling large number of predicates

- Number of transitions in the abstract model is exponential in number of predicates

- With large number of predicates
  - SAT solver generates too many satisfying assignment
  - Is it necessary to create one SAT equation for abstraction

- Partition the predicates into smaller sets of predicates

- Abstract the transition relation with respect to each set
**Predicate Partitioning**

**Initial state**
- \( x = 100, \ y = 200 \)
- \( y = 100, \ y = 200 \)

**Transition relation**
- \( x' = y \)
- \( y' = x \)

**Final state**
- \( x' = 100, \ x' = 200 \)
- \( y' = 100, \ y' = 200 \)

\[ \mathcal{AE} \]

- \( x = 100, \ x = 200 \)
- \( y' = x \)

- \( y = 100, \ y = 200 \)
- \( x' = y \)

\[ \text{Partition} \]
Predicate Partitioning

• Introduces over-approximation

• Refinement as needed
  – Refinement using weakest pre-condition to derive new predicates
  – Refinement by finding relationship between existing predicates (due to predicate partitioning)

• See technical report for more details
## Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of code</th>
<th>#Latches</th>
<th>#Variables</th>
<th>Time</th>
<th>#Predicates</th>
<th>#Iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache coherence</td>
<td>549</td>
<td>43</td>
<td>170</td>
<td>103s</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>mpeg decoder 1</td>
<td>1215</td>
<td>567</td>
<td>800</td>
<td>130s</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>mpeg decoder 2</td>
<td>1215</td>
<td>567</td>
<td>800</td>
<td>208s</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>SDLX</td>
<td>898</td>
<td>41</td>
<td>81</td>
<td>218s</td>
<td>39</td>
<td>2</td>
</tr>
<tr>
<td>Miim</td>
<td>841</td>
<td>83</td>
<td>237</td>
<td>70s*</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>PI-Bus</td>
<td>1020</td>
<td>312</td>
<td>863</td>
<td>16s</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

* Using lazy abstraction
References

Counterexample-Guided Abstraction Refinement. CAV‘00.

Predicate Abstraction of ANSI-C Programs Using SAT. FMSD‘2004


QUESTIONS?