# Advanced Unbounded CTL Model Checking Based on AIGs, BDD Sweeping, And Quantifier Scheduling

Florian Pigorsch, Christoph Scholl, Stefan Disch

Albert-Ludwigs-Universität Freiburg, Institut für Informatik

FMCAD 2006

# Outline

# Why another data structure for model checking?

- BDD based model checking fails on certain problems
  - e.g. blow-up when representing combinational multipliers
  - ...
- And-Inverter Graphs have been successfully used in:
  - Combinational Equivalence Checking (e.g. Mishchenko, Kuehlmann)
  - Bounded Model Checking (e.g. Kuehlmann)
  - Technology mapping
  - Various other verification/synthesis applications

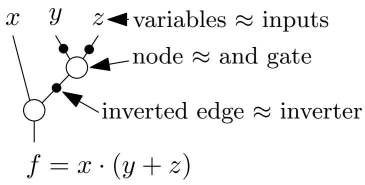
# Why another data structure for model checking?

- BDD based model checking fails on certain problems
  - e.g. blow-up when representing combinational multipliers
  - ...
- And-Inverter Graphs have been successfully used in:
  - Combinational Equivalence Checking (e.g. Mishchenko, Kuehlmann)
  - Bounded Model Checking (e.g. Kuehlmann)
  - Technology mapping
  - Various other verification/synthesis applications

Use And-Inverter Graphs as the underlying data structure for unbounded symbolic CTL model checking

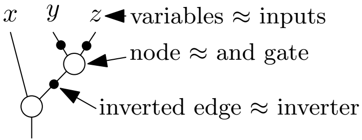
## And-Inverter Graphs

# And-Inverter Graphs



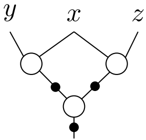

- Networks of 2-input and gates and inverters
- Simple data structure
- Every Boolean function can be represented by an AIG

# And-Inverter Graphs




$$f = x \cdot (y + z)$$

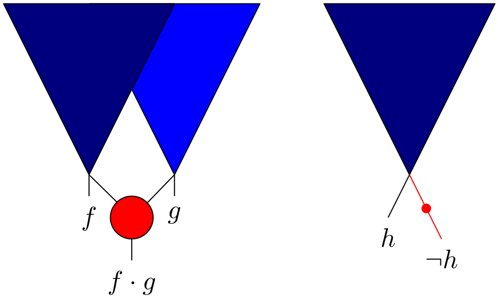
- Networks of 2-input and gates and inverters
- Simple data structure
- Every Boolean function can be represented by an AIG
- But: possibly redundant and non-canonical (in contrast to BDDs)




$$f = y \cdot x + x \cdot z$$

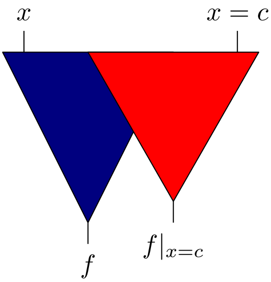
# Operations

- AND by adding a new and node, NOT by adding an inverted edge



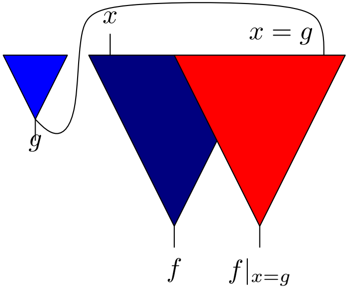
# Operations

- AND by adding a new and node, NOT by adding an inverted edge
- Cofactor by propagating constants



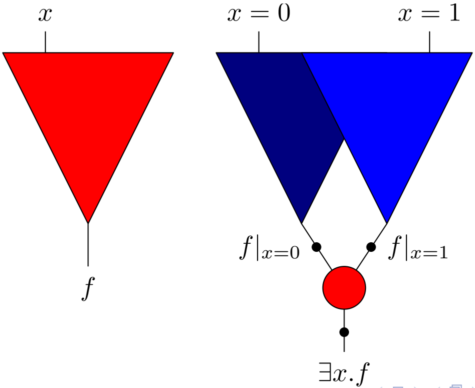
# Operations

- AND by adding a new and node, NOT by adding an inverted edge
- Cofactor by propagating constants
- Substitution of variables



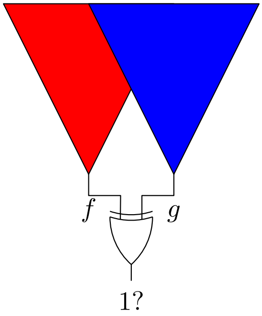
# Operations

- AND by adding a new and node, NOT by adding an inverted edge
- Cofactor by propagating constants
- Substitution of variables
- Quantification by cofactoring ($\exists x.f \equiv f|_{x=0} + f|_{x=1}$) (possibly expensive)



# Operations

- AND by adding a new and node, NOT by adding an inverted edge
- Cofactor by propagating constants
- Substitution of variables
- Quantification by cofactoring ($\exists x.f \equiv f|_{x=0} + f|_{x=1}$) (possibly expensive)
- Equivalence check of two nodes? $\Rightarrow$ SAT



# Are we ready for model checking?

We already have the needed operations for model checking:

- Basic Boolean operators
- Quantification
- Substitution
- Equivalence check for two nodes

# Are we ready for model checking?

We already have the needed operations for model checking:

- Basic Boolean operators
- Quantification
- Substitution
- Equivalence check for two nodes

But, plain AIGs are not enough:

- Too many redundant nodes
- Quantification will result in extremely large AIGs

# Are we ready for model checking?

We already have the needed operations for model checking:

- Basic Boolean operators
- Quantification
- Substitution
- Equivalence check for two nodes

But, plain AIGs are not enough:

- Too many redundant nodes
- Quantification will result in extremely large AIGs

We need to add some things to make model checking with AIGs feasible

# Functionally Reduced And-Inverter Graphs: FRAIGs

# FRAIGs

## FRAIG (A. Mishchenko)

A functionally reduced AIG does not contain two nodes representing the same Boolean function.

How to create FRAIGs?

# FRAIGs

## FRAIG (A. Mishchenko)

A functionally reduced AIG does not contain two nodes representing the same Boolean function.

How to create FRAIGs?
When creating a new node...

# FRAIGs

## FRAIG (A. Mishchenko)

A functionally reduced AIG does not contain two nodes representing the same Boolean function.

How to create FRAIGs?
When creating a new node...

- Find possibly equivalent candidate nodes using simulation

# FRAIGs

**FRAIG (A. Mishchenko)**

A functionally reduced AIG does not contain two nodes representing the same Boolean function.

How to create FRAIGs?
When creating a new node...

- Find possibly equivalent candidate nodes using simulation
- Solve the equivalence checking problems of the new node and candidate nodes with a SAT solver (MiniSAT)

# FRAIGs

## FRAIG (A. Mishchenko)

A functionally reduced AIG does not contain two nodes representing the same Boolean function.

How to create FRAIGs?
When creating a new node...

- Find possibly equivalent candidate nodes using simulation
- Solve the equivalence checking problems of the new node and candidate nodes with a SAT solver (MiniSAT)
- When finding an equivalent candidate: delete one of the two nodes

# FRAIGs

## FRAIG (A. Mishchenko)

A functionally reduced AIG does not contain two nodes representing the same Boolean function.

How to create FRAIGs?
When creating a new node...

- Find possibly equivalent candidate nodes using simulation
- Solve the equivalence checking problems of the new node and candidate nodes with a SAT solver (MiniSAT)
- When finding an equivalent candidate: delete one of the two nodes
- Use the feedback from the solver to strengthen the simulation values

# FRAIGs

## FRAIG (A. Mishchenko)

A functionally reduced AIG does not contain two nodes representing the same Boolean function.

How to create FRAIGs?
When creating a new node...

- Find possibly equivalent candidate nodes using simulation
- Solve the equivalence checking problems of the new node and candidate nodes with a SAT solver (MiniSAT)
- When finding an equivalent candidate: delete one of the two nodes
- Use the feedback from the solver to strengthen the simulation values

A FRAIG is reduced by removing (functionally) redundant nodes

# How to handle pairs of equivalent nodes?

When we detect a pair of functionally equivalent nodes during FRAIG construction, we have to delete one of the two nodes.

# How to handle pairs of equivalent nodes?

When we detect a pair of functionally equivalent nodes during FRAIG construction, we have to delete one of the two nodes.
Two different simple node selection heuristics:

- $h_{keep}$: keep the old, existing node and delete the new node
- $h_{size}$: keep the node with the smaller cone size, delete the other node

# Speeding up Quantification

# Quantifier Scheduling: A Motivating Example

n-bit Carry-Ripple-Adder ( $\vec{s} = \vec{x} + \vec{y}$ )
Formula $\exists \vec{x}.s_n \cdot \overline{s_{n-1}}$

# Quantifier Scheduling: A Motivating Example

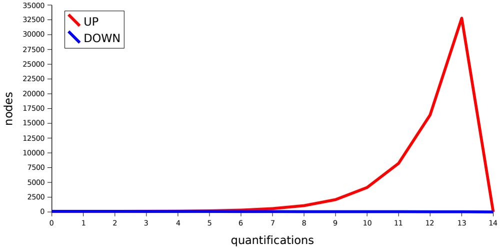
n-bit Carry-Ripple-Adder ( $\vec{s} = \vec{x} + \vec{y}$ )
Formula $\exists \vec{x}.s_n \cdot \overline{s_{n-1}}$

- quantification order UP: quantify $x_0$ first, then $x_1$, ...
- quantification order DOWN: quantify $x_{n-1}$ first, then $x_{n-2}$, ...

# Quantifier Scheduling: A Motivating Example

n-bit Carry-Ripple-Adder ( $\vec{s} = \vec{x} + \vec{y}$ )
Formula $\exists \vec{x}.s_n \cdot \overline{s_{n-1}}$

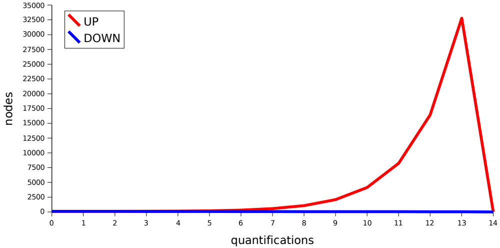
- quantification order UP: quantify $x_0$ first, then $x_1$, ...
- quantification order DOWN: quantify $x_{n-1}$ first, then $x_{n-2}$, ...



# Quantifier Scheduling: A Motivating Example

n-bit Carry-Ripple-Adder ( $\vec{s} = \vec{x} + \vec{y}$ )
Formula $\exists \vec{x}. s_n \cdot \overline{s_{n-1}}$

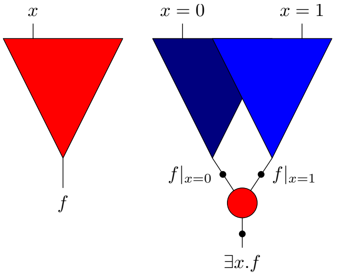
- quantification order UP: quantify $x_0$ first, then $x_1, \ldots$
- quantification order DOWN: quantify $x_{n-1}$ first, then $x_{n-2}, \ldots$




$\Rightarrow$ Quantification order is crucial!

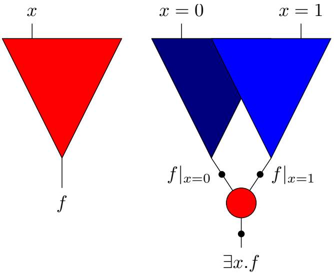
# Multiple Quantifications

- One quantification operation may double the AIG's size



# Multiple Quantifications

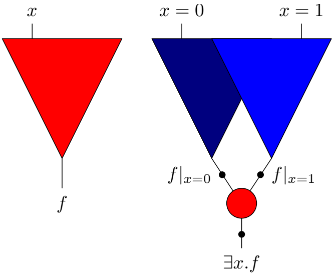
- One quantification operation may double the AIG's size




- A series of quantifications may lead to an exponential blow-up

# Multiple Quantifications

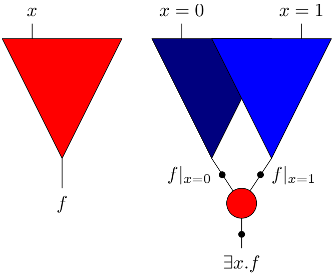
- One quantification operation may double the AIG's size




- A series of quantifications may lead to an exponential blow-up
- How to avoid the blow-up?

# Multiple Quantifications

- One quantification operation may double the AIG's size




- A series of quantifications may lead to an exponential blow-up
- How to avoid the blow-up?
- Find a good quantification schedule!

# A greedy algorithm for quantifier scheduling

## Greedy quantification

```
greedy_quantify( f, vars )
    res ← f;
    while vars ≠ ∅
        bestvar ← NULL; bestsize ← ∞;
        for all v ∈ vars
            if expected_size( res, v ) < bestsize
                bestsize ← expected_size( res, v ); bestvar ← v;
        res ← quantify( res, bestvar );
        vars ← vars \{ bestvar };
    return res;
```

# Expected quantification result size?

- How to compute the expected size of the quantification result?

# Expected quantification result size?

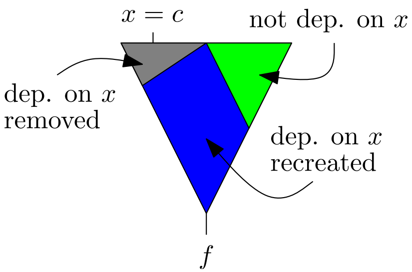
- How to compute the expected size of the quantification result?
- One could actually perform quantifications by all variables to get the exact sizes. Too expensive!

# Expected quantification result size?

- How to compute the expected size of the quantification result?
- One could actually perform quantifications by all variables to get the exact sizes. Too expensive!
- Estimate the resulting size of one quantification step by simulating the two constant propagations:

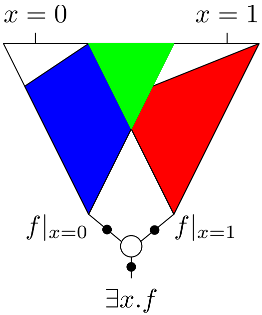
# Expected quantification result size?

- How to compute the expected size of the quantification result?
- One could actually perform quantifications by all variables to get the exact sizes. Too expensive!
- Estimate the resulting size of one quantification step by simulating the two constant propagations:



# Expected quantification result size?

- How to compute the expected size of the quantification result?
- One could actually perform quantifications by all variables to get the exact sizes. Too expensive!
- Estimate the resulting size of one quantification step by simulating the two constant propagations:



# Combining AIGs and BDDs: BDD Sweeping

# BDD Sweeping: Combining advantages of AIG and BDD representations

- “Classical” notion of BDD sweeping by A. Kuehlmann: Detection of functionally equivalent AIG nodes by BDD construction

# BDD Sweeping: Combining advantages of AIG and BDD representations

- "Classical" notion of BDD sweeping by A. Kuehlmann: Detection of functionally equivalent AIG nodes by BDD construction
- Our functionally reduced AIGs don't contain such nodes (achieved by SAT)!

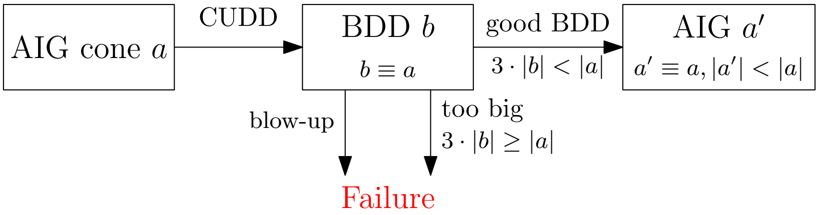
# BDD Sweeping: Combining advantages of AIG and BDD representations

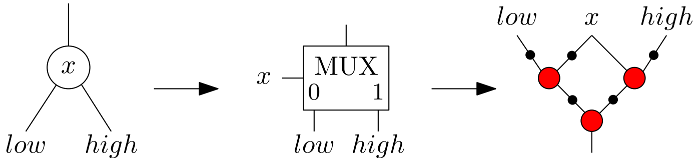
- "Classical" notion of BDD sweeping by A. Kuehlmann: Detection of functionally equivalent AIG nodes by BDD construction
- Our functionally reduced AIGs don't contain such nodes (achieved by SAT)!
- But: BDD representations of Boolean functions in model checking are not always large...

# BDD Sweeping: Combining advantages of AIG and BDD representations

- "Classical" notion of BDD sweeping by A. Kuehlmann: Detection of functionally equivalent AIG nodes by BDD construction
- Our functionally reduced AIGs don't contain such nodes (achieved by SAT)!
- But: BDD representations of Boolean functions in model checking are not always large...
- Therefore: Use "good" BDD representations to restructure AIGs!

# BDD Sweeping Algorithm



# Application of BDD Sweeping

- We apply BDD sweeping to the results of quantifications
- We limit the number of created BDD nodes to avoid a blow-up
- Heuristics ensure that BDD-sweeping is used less frequently if the BDD node limit was reached in the past

# Experimental Results

# Our AIG based Model Checker

- We use a standard CTL model checking algorithm based on fix point iteration
- The transition function and the characteristic functions of state sets are represented by AIGs
- Alternatives for pre-image computation:
  - transition relation based:

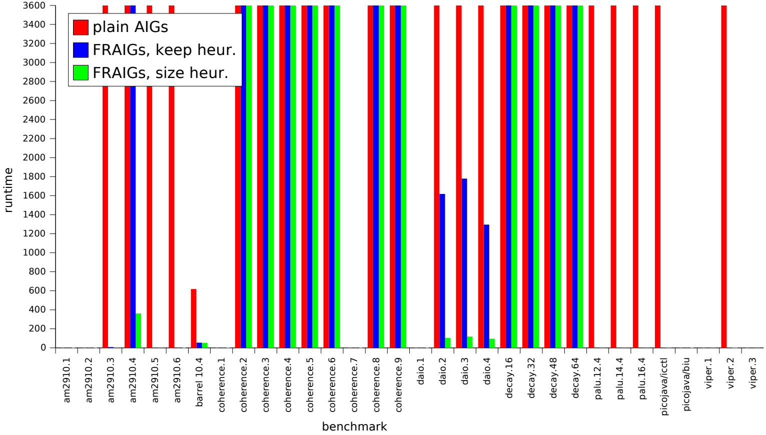
$$\chi_{Sat(EX\ \phi)}(\vec{q},\vec{x}) := \exists \vec{q}' \exists \vec{x}' (\chi_R(\vec{q},\vec{x},\vec{q}') \cdot (\chi_{Sat(\phi)}|_{\vec{q}\leftarrow\vec{q}',\vec{x}\leftarrow\vec{x}'})(\vec{q}',\vec{x}'))$$

  - transition function based:

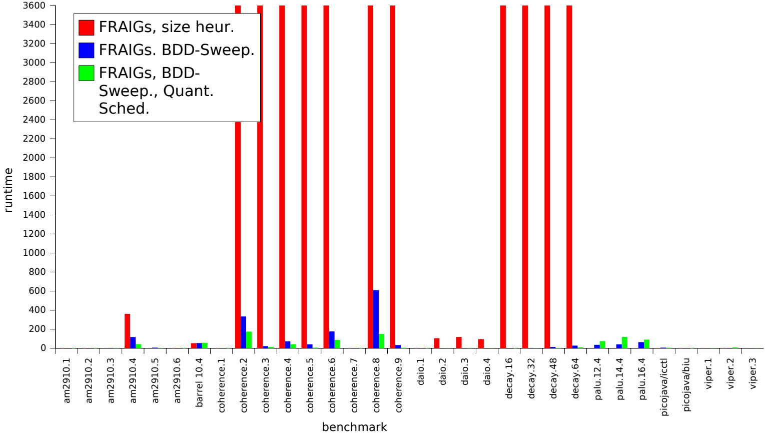
$$\chi'_{Sat(EX\ \phi)}(\vec{q},\vec{x}) := \exists \vec{x}' (\chi_{Sat(\phi)}|_{\vec{q}\leftarrow\vec{\delta}(\vec{q},\vec{x}),\vec{x}\leftarrow\vec{x}'})(\vec{q},\vec{x}'))$$

# Impact of Functional Reduction and Node Selection Heuristics

No BDD sweeping, no quantifier scheduling

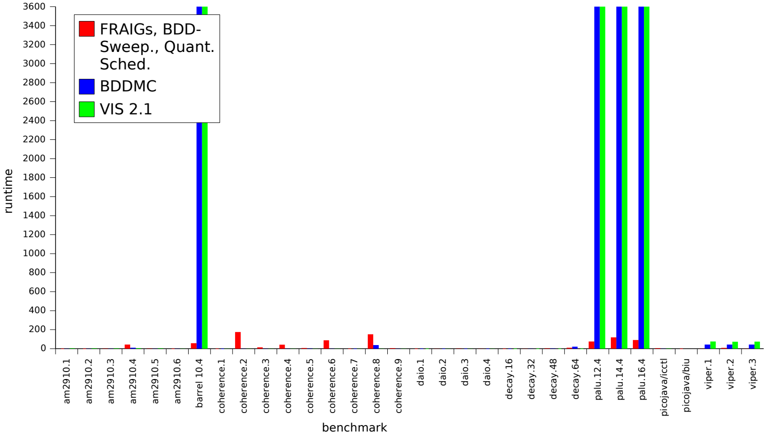


# Impact of BDD Sweeping and Quantifier Scheduling



# Comparison with BDD based model checkers

- VIS: VIS 2.1, sifting, no reachability analysis
- BDDMC: our model checker with AIGs replaced by BDDs



# Summary

- Successful unbounded CTL model checking based on And-Inverter Graphs (up to 2000 quantifications)

# Summary

- Successful unbounded CTL model checking based on And-Inverter Graphs (up to 2000 quantifications)
- Made possible by using
  - Functionally Reduced And-Inverter Graphs
  - Simple node selection heuristics
  - BDD sweeping
  - and Quantifier Scheduling

# Summary

- Successful unbounded CTL model checking based on And-Inverter Graphs (up to 2000 quantifications)
- Made possible by using
  - Functionally Reduced And-Inverter Graphs
  - Simple node selection heuristics
  - BDD sweeping
  - and Quantifier Scheduling
- Outperforms BDD based MCs on various benchmarks...
- and has comparable runtimes on most other benchmarks

# Future and Related Work

- Optimize heuristics (node selection, application of BDD sweeping)
- Lazier AIG compression instead of complete functional reduction
  - Time limited SAT to skip hard SAT instances
- Evaluate recent AIG rewriting techniques
- Try structural SAT instead of CNF based SAT

# Future and Related Work

- Optimize heuristics (node selection, application of BDD sweeping)
- Lazier AIG compression instead of complete functional reduction
  - Time limited SAT to skip hard SAT instances
- Evaluate recent AIG rewriting techniques
- Try structural SAT instead of CNF based SAT
- At ATVA06 we presented a hybrid model checker based on AIGs and linear constraints over the reals

Thank you for your attention!