Integrating FV Into Main-Stream Verification: The IBM Experience

Jason Baumgartner
IBM Corporation
baumgarj@us.ibm.com

Thanks to: Viresh Paruthi, Hari Mony, Wolfgang Roesner

November 12, 2006

`http://www.research.ibm.com/sixthsense`

Overview

- Intro to Hardware Models and Verification Methods
- Functional Verification at IBM
  - Can a High-End Processor be Fully Formally Verified?
  - Non-Intrusive Formal Verification
- Scalability in FV
  - Semi-Formal Verification
  - Multi-Algorithmic Reasoning
  - *SixthSense*: Transformation-Based Verification (TBV)
- Reusing Sim Testbenches in FV
- SixthSense Applications

Intro to Hardware Models and Verification Methods

Hardware Verification Tasks

- Numerous distinct verification tasks in the hardware design flow:

1. Checking equivalence of to-be-fabricated circuit to the HDL model
   - Necessary to validate synthesis
     - Since functional verification performed on HDL
   - *Combinational equiv checking (CEC) is the most prevalent industrial FV application*
     - Applicable only to 1:1 latchpoint equivalent designs
     - Emerging sequential equiv checking (SEC) technologies lift this restriction

- CEC / SEC also useful for a variety of other purposes
  - Refer to Thursday presentation by Paruthi

Hardware Verification Tasks

- Numerous distinct verification tasks in the hardware design flow:

2. **Functional implementation verification**
   - ∗ Does the implementation satisfy its specification?
     - · E.g., adherence of HDL to (micro)architecture properties
3. Protocol verification
   - ∗ Operates on abstract models of system, not implementation
   - ∗ Does the architecture implement the necessary functionality?
   - ∗ Are the protocols (memory coherency, bus protocols) correct?
4. . . .

Hardware Models

- Focus of this presentation: timing-independent functional verification
- HW can be modeled using HDL; as a netlist; as an automaton
- Reason about *signals* over *time*
- *Signals* driven by
  - Combinational logic (AND gates, inverters) has 0 delay
  - Sequential elements (registers, latches) evaluate every time-step
- *Time* refers to smallest observable granule of logic evaluation, dictated by clocking of sequential elements

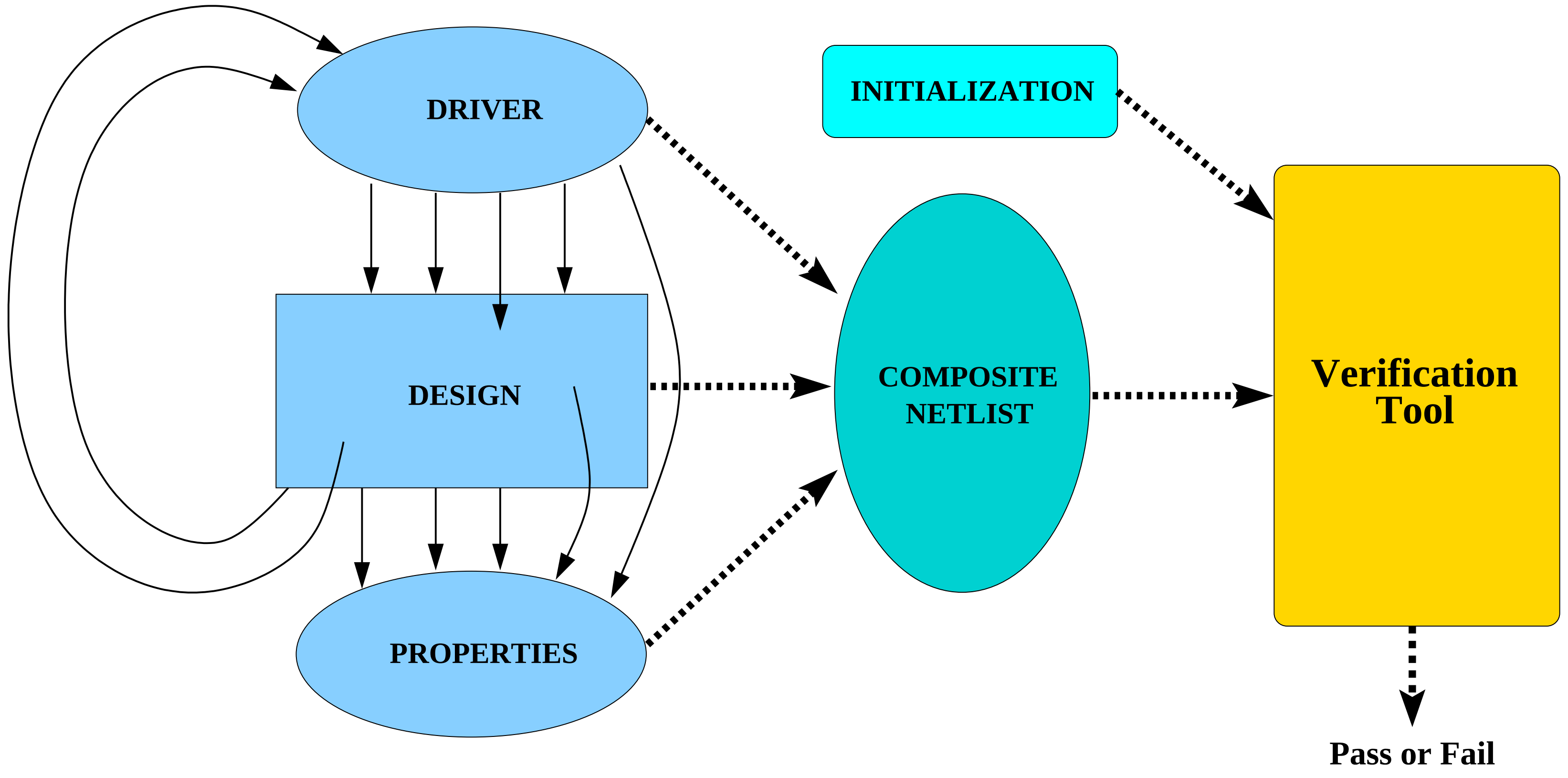
Modeling Flexibility

- Allows direct modeling of sequential hardware
- Many other types of problems may be modeled in this way
  - Protocols may be modeled using sequential, combinational primitives
  - Timing verification may be modeled using these primitives
    - To account for delays through combinational gates, model with both sequential and combinational primitives

Verification Testbenches

- We focus on verification of safety properties
  - Regardless of specification language, can be synthesized into *checker* logic
- Typically requires a *driver* encoding *input assumptions*
  - Can be specified as random logic to compose onto the design, or as *constraints*
- Typically requires an *initial state* specification
  - *Initial states* dictate *reachable states*

Verification Testbenches

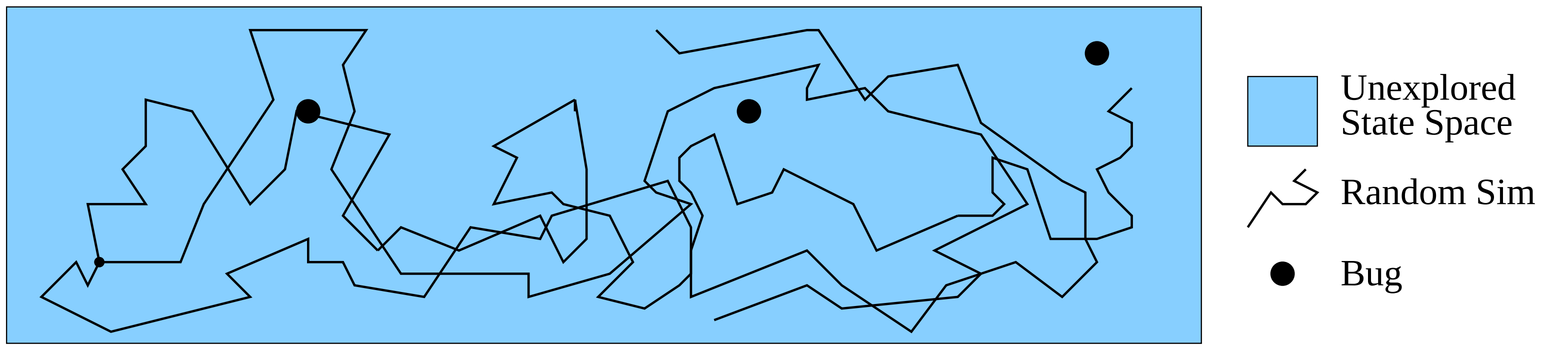


Testbench semantics: *Does there exist a trace from an initial state, consistent with driver, which violates any property?*

Simulation

- Numerous algos may be used to address the verification problem
- *Simulation* is the most predominantly used algorithm
  - Validates the design against specific sequences of input stimuli
    - Non-exhaustive: do not confuse with *simulation relation*
    - Useful for falsification only; proof-incapable
      - Suffers the so-called *coverage problem*

  + Evaluation against explicit tests enables **high scalability**



Simulation

- Sim specs can be written using a variety of languages
  - PSL, SVA are commonly-used assertion languages
  - IBM often uses the VHDL-based testbench language *BugSpray*
    - Augmented with HDL, PSL asserts
  - Simulation, however, often uses C/C++ type of languages
    - These type of languages cannot be readily reused in FV

Simulation

- Despite advances in FV, sim retains predominant status due to
  - Scalability: used for tasks too complex for FV
    - E.g., dispatch-to-completion checks for high-end processors
  - Legacy: tools, skills, methodology using sim are well-established
  - Reuse of verification IP: takes effort to re-write sim specs in a *formal* language

Model Checking

- + Proof-capable, unlike simulation
  - + Much more adept at finding corner-case bugs
- + Automated: easy to use, for smaller problems
  - - Substantial expertise, manual effort required for larger designs
    - ∗ Typically only applicable at block level
    - ∗ More difficult to cover (micro-)architectural properties
      - · A different type of *coverage problem*
  - + May be alleviated by abstraction, reduction algos
  - + *Semi-formal* algos stretch capacity for bug-hunting

Theorem Proving

+ Proof-capable; can be **much more scalable** than model checking
    - Though more difficult to use; often not a push-button solution

+ Can be used as front-end to model checking
    + *Light-weight* theorem proving can be fairly easy to use
    + Theorem prover used to decompose intractable high-level proof

- IBM is currently not extensively using theorem provers
    * Fully-automated frameworks are the focus of this presentation
    * Though - refer to Wednesday’s presentation by Sawada&Reeber

Functional Verification at IBM

Functional Verification at IBM

- *Simulation* remains the most prevalent verification platform
  - Scalable; flexible specification languages; legacy reasons
- Model checking began to make an impact at IBM ~ 10 years ago
  - *RuleBase*, developed by IBM Haifa Research Laboratory
  - Used by a variety of projects, including POWER3-POWER6
  - Quickly demonstrated its ability to expose corner-case bugs

Functional Verification at IBM

- Model checking has been used as a *complement* to simulation
- FV team **much smaller** than sim team
  - Not enough FV resources to attempt to verify entire design
  - Or even to *formally touch* each design component
- Verif + design teams prioritize among design components
  - Choose to deploy FV to most complex logic; hardest to test; ...
- Also use FV for *fire-fighting* to cope with late design bugs

Functional Verification at IBM

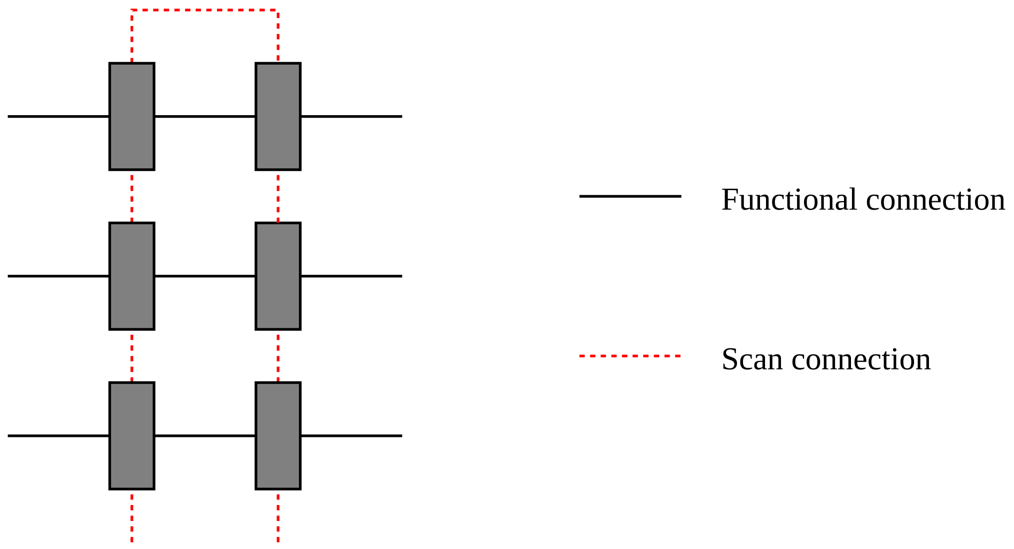
- Model checking was very effective at wringing out corner-case bugs
  - Many projects will estimate saving $\sim$1 chip fabrication due to FV
- Became clear that FV was a critical technology to be tapped into, to some extent
  - But how much?

Complexity of High-End Processors

- POWER4 processor
  - **1.5 million** lines of VHDL   **174 million** transistors
- *Moore's law*: complexity increases for POWER5, POWER6, . . .
  - Increase in # *transistors / chip* contributes somewhat to complexity
    - ∗ *Modularity* alleviates *complexity / transistor* ratio
      - · Integrate N identical μproc cores on chip
    - ∗ Increased RAM size alleviates *complexity / transistor* ratio
  - Increase in *speed* contributes significantly to HDL complexity
    - ∗ *CEC methodology* requires 1:1 latch correspondence between circuit, HDL

Complexity of High-End Processors

- CEC methodology forces HDL to acquire circuit characteristics
  - Word-level operations, isomorphisms broken by self-test logic
    - ∗ Self-test logic: much more intricate than mere *scan chains*
    - ∗ And reused for *functional* obligations: initialization, reliability, . . .
    - ∗ Refer to Monday presentation by Glökler



Complexity of High-End Processors

- CEC methodology forces HDL to acquire circuit characteristics
  - Word-level operations broken; bit-level control coalesced due to synthesis requirements
    - *Use of common building-blocks* alters vector bundling
      - E.g., project may provide highly-optimized 8-bit arrays
  - Placement issues: redundancy added to HDL
    - E.g. lookup queue needs to route data to 2 locations which reside in different areas of chip
    - Lookup queue may need to be replicated

Complexity of High-End Processors

- CEC methodology forces HDL to acquire circuit characteristics
  - Timing issues: arithmetic ops pipelined asymmetrically
    - E.g. due to propagation delays from LSB to MSB
  - Power-savings logic complicates even simple pipelines
- Design HDL becomes difficult-to-parse bit-level representation
- Industrial FPU example: 15,000 lines VHDL vs. 500 line ref model
- Even RAM implementations tend to become very complex
  - Refer to Thursday presentation by Jacobi for more examples

Can a High-End Processor be Fully Formally Verified?

- SMT, word-level techniques are difficult to apply
  - Abstraction inherent in uninterpreted functions, removal of bitvector nonlinearity is lossy
  - More critically: difficult to find places to attempt such abstractions
- Need to operate on a more abstract model?
  - Developing, maintaining such a model is **very expensive**
  - Not clear that high-level proofs would be feasible even with such an abstraction...
    - $*$ But would certainly help proof, and even falsification, efforts

Can a High-End Processor be Fully Formally Verified?

- Can we efficiently obtain, maintain a more abstract model?

- Emerging SEC technologies: a promising direction
  - Can prove correspondence of abstract vs. CEC model
    - ∗ Else, we are not really verifying the implementation...
  - Eliminate the need for CEC-compatible HDL?
    - + Directly correspond abstract model to circuit
    - + *Saves development effort*: simpler functional abstract model
    - - Though requires greater synthesis sophistication
      - · Automated synthesis inadequate for POWER-style design
    - - And still requires validation of self-test logic
      - · To be done on circuit model?

Can a High-End Processor be Fully Formally Verified?

- Rigorous assertion-based, compositional methodologies may also be used to “scale” FV to larger designs
  - Adequate assertions may enable an inductive proof
    - And imply a proof decomposition strategy
- Though no verif technique has scaled near a POWER processor
- Tend to be **very manually intensive** for larger designs

Can a High-End Processor be Fully Formally Verified?

- Simulation is easier to deploy
  - No need for abstract model
  - No need for manual decomposition
  - No need for copious assertions
- Though sim has its own limitations and drawbacks!
  - Incomplete; misses bugs
  - Methodologies to strive for high coverage are time-consuming in themselves
- Nonetheless: skills, experience, tools, perceived risk are all reasons that sim remains predominant validation methodology

Can a High-End Processor be Fully Formally Verified?

- *Full FV* of **this type** of design requires expensive, risky paradigm shift
  - Likely a very good idea, though likely needs to be eased into...
- How can we *ease into* a wide-spread FV paradigm?
  - Offer tangible *return on investment (ROI)* and *resource savings*
    - Worst outcome: person-months spent developing formal specs, merely to choke FV tool
  - Enable non-experts to leverage the technology
    - Cannot expect large verif+design team to all have PhDs in FV!
  - Do not *require* a radical change in design paradigm to *enable* FV
    - Need for reuse of IP, skills, tools, methodologies is a high barrier

Non-Intrusive Formal Verification

- Rather than push (and wait) for a giant leap toward adoption of full-blown FV, our philosophy is non-intrusive

- *Ease of use*
  - People accept the fact that they need to set up sim testbenches
  - Goal: make formal "as easy" to use as sim

- Requires *scalability+automation, without altering design methodology*
  - Lessens risk of negative ROI to develop a formal spec, just to choke FV tool
  - Lowers required expertise for large-scale testbenches
  - Enables reuse of specs across FV, sim
    - Refer to Tuesday panel discussion by Roesner

No, we're not Crazy

- Rather than push (and wait) for a giant leap toward adoption of full-blown FV, our philosophy is non-intrusive

- Improving design methodology through demonstrated cost effectiveness is a *slow* but sure byproduct

- And there are increasingly cases of systematic prethought proof methodologies
  - Which we wholeheartedly foster!

Scalable FV

So - how do we achieve scalability+automation?

1. Tuning the system for importing and manipulating LARGE designs

2. Integrate falsification as well as proof threads

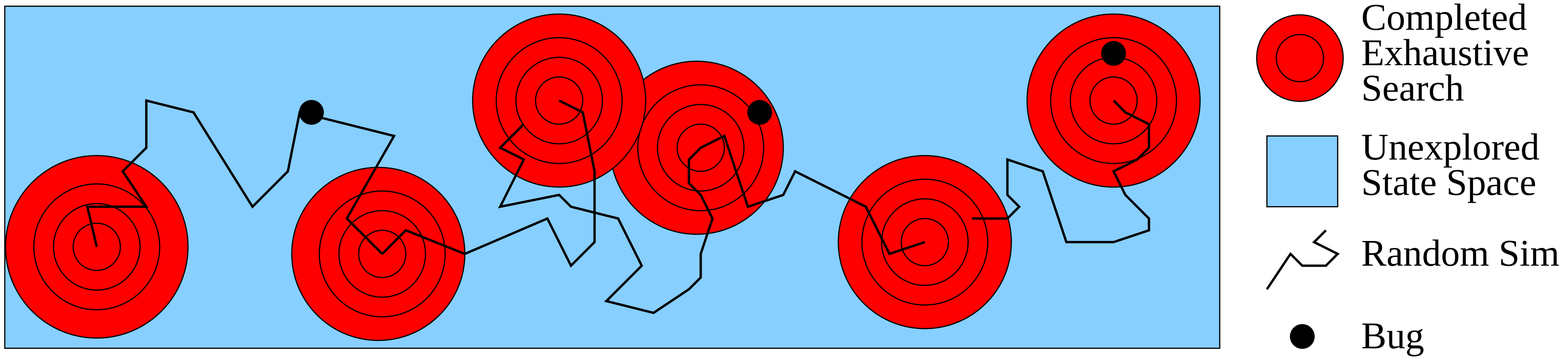
   - *Semi-formal falsification* improves ROI of formal spec

3. Integrate a variety of algorithms

   - Every problem is different
   - Different proof algorithms have different strengths / weaknesses
   - Abstraction algorithms can yield HUGE reductions
   - Reduction / transformation algos can yield dramatic reductions

Semi-Formal Verification

- A mechanism to leverage formal algos in a *resource-bounded way*
- Refers to a *hybrid search* paradigm
  - Often leverages simulation to reach *deep* states
  - Formal search triggered from deep states
  - Resource-bounded formal search *amplifies* simulation
- Many intricacies in how to best orchestrate these algos



Semi-Formal Verification

- Cynical view: *merely prolongs the agony of sim-based methodology*
  - Since proof-incapable in itself
- However, very useful in practice
  - Extends bug-hunting power of BMC to *deep* bugs in large designs
  - We view this as an *enabling technology* to wider-spread FV
    - Critical to convince people to develop formal vs. sim-only specs
    - Critical to disperse formal spec development, FV deployment
    - Even if a rigorous proof methodology is to be attempted, SFV useful to wring out higher-level bugs earlier

Proof Algorithms

- A robust set of proof engines is central to a formal toolset
- BDD-based reachability is a cornerstone proof technique
  - Prone to memout above a few hundred state variables
    - Often requires reduction / abstraction to bring down design size
  - Despite advances in complete SAT-based techniques, BDDs are often superior on *reduced designs*
- SAT-based interpolation can outperform BDDs in cases
  - Typically *shallow*, complex designs

Proof Algorithms

- Induction is a critical proof technique
  - Simplest variant: *can "any state which does not violate property" transition to one which violates property?*
- Fast and scalable: SAT check on unfolded design
  - A robust tool needs to leverage induction to avoid excessive resource consumption by falsification, reachability threads
- Though unfortunately, often inconclusive
  - Not reachability-based: cannot discern *valid* from *unreachable* fail
  - Even with *complete* variant, fine line: *inductive* vs. *intractable*
    - Since *completeness* pushes from NP to PSPACE

Proof Algorithms

1. k-induction: *can "any state which does not violate property within k time-frames" transition to one which violates property in $k+1$?*

   + Tightens overapproximate *starting states* of simple induction
   + Can be made *complete* by unique-state constraints

   - Though may require intractably large k
   - And unique-state constraints dominate cost of SAT check

"Checking Safety Properties Using Induction and a SAT-Solver" FMCAD00

"Temporal Induction by Incremental SAT Solving" BMC03

Proof Algorithms

2. van Eijk-style induction

   - Try to concurrently prove *sets of invariants*, e.g. register equivalences at time 0
   - If any invariant cannot be inductively proven, prune from the set and repeat proof attempt
   - Once terminates, can leverage proven invariants in various ways
     - Often a stronger proof technique than k-induction, since *invariant set* prunes more unreachable states than property alone
     - Can be combined with k-induction

"Sequential equivalence checking without state space traversal" DATE98

"SAT-Based Verification without State Space Traversal" FMCAD00

Proof Algorithms

- We have found van Eijk-style induction, augmented with a richer set of algorithms, to be a very powerful proof technique
  - Generalizes inductive proof frameworks

1. Use semi-formal analysis to guess gate-equivalence (modulo inversion)
2. Leverage an arbitrary set of algos to prove suspected equivalence
   - Induction filters out easy cases; abstraction / reduction then reachability / interpolation discharge the rest
3. If any candidates cannot be proven, refine and goto 2; else break

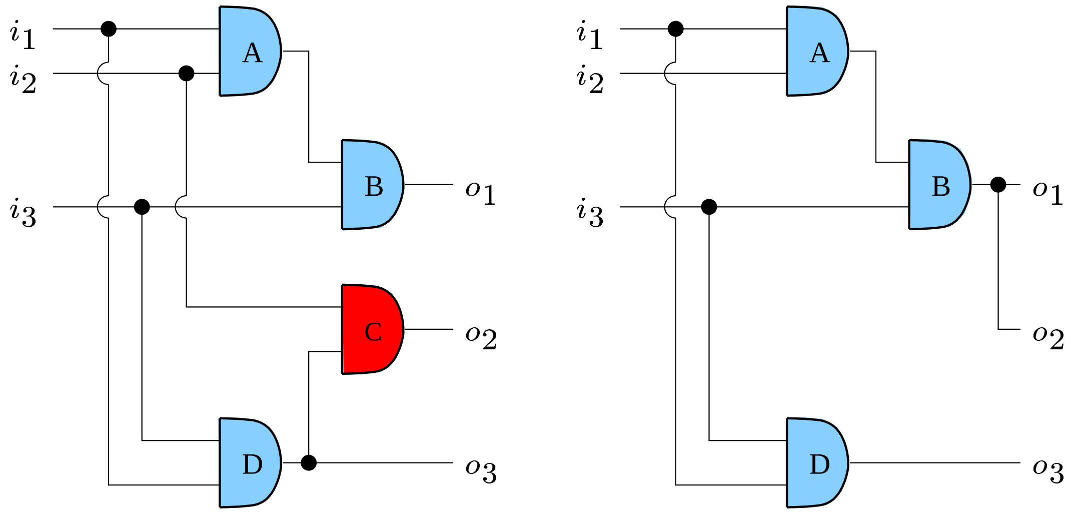
“Exploiting Suspected Redundancy without Proving It” DAC05

Transformation Algorithms

- Two classes: *property-preserving reductions* and *abstractions*
- Both are extremely useful to proofs as well as falsification efforts
- We will now discuss various example transformation algorithms
  - These are some of the transforms we use within *SixthSense*

Combinational Optimization Algorithms

- BDD- and SAT-sweeping, to merge gates equivalent modulo inversion in arbitrary state



- Can be enhanced through leveraging observability don't cares

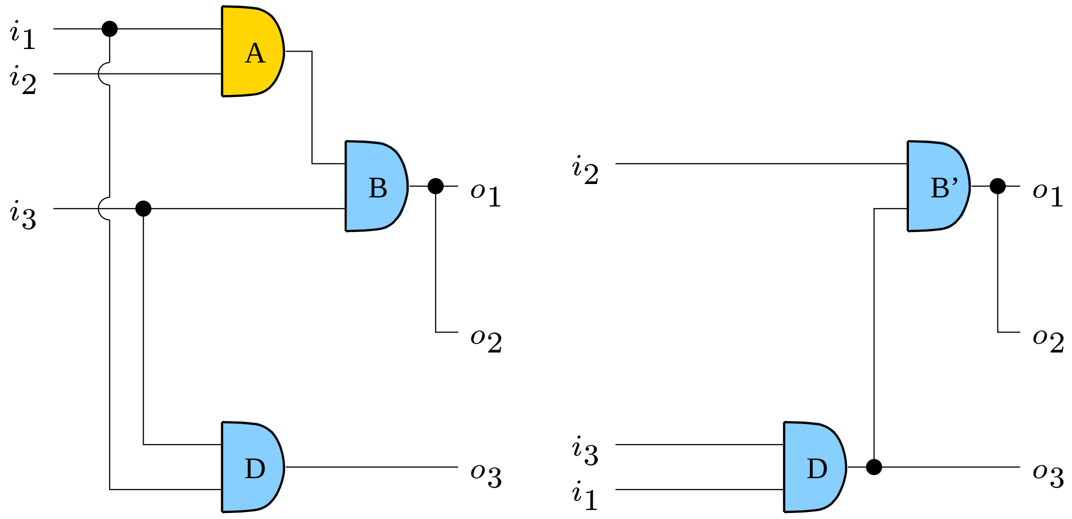
  "Robust Bool. Reasoning for Equiv. Checking and Functional Pro. Verif" TCAD02

  "Dynamic T.R. Simplification for Bounded Prop. Checking" ICCAD04

  "SAT Sweeping with Local Observability Don't-Cares" DAC06

Combinational Rewriting Algorithms

- Logic rewriting algorithms, to simplify logic expressions



- Lowering gate count greatly enhances SAT-based reasoning
- Also tends to enhance reduction potential of other algorithms

  "DAG-Aware Circuit Compression For Formal Verification" ICCAD04

  "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis" DAC06

  "Factor Cuts" ICCAD06

Sequential Redundancy Removal

- Use *van Eijk* induction to identify gates equivalent modulo inversion
  - Unlike combinational variant, performs true sequential analysis
  - Greater reduction potential at a greater cost
- The resulting invariants correlate to safe merges
- Simplifies design for subsequent reasoning
  - A further generalization of induction

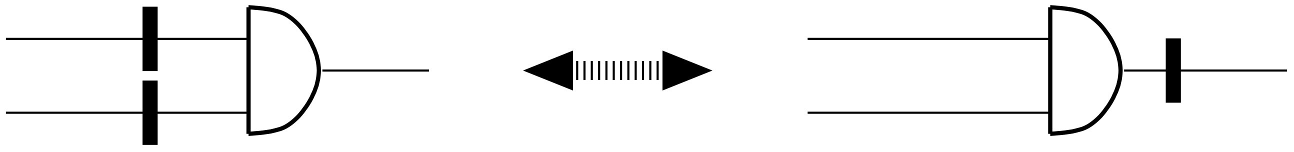
"SAT-Based Verification without State Space Traversal" FMCAD00

"Exploiting Suspected Redundancy without Proving It" DAC05

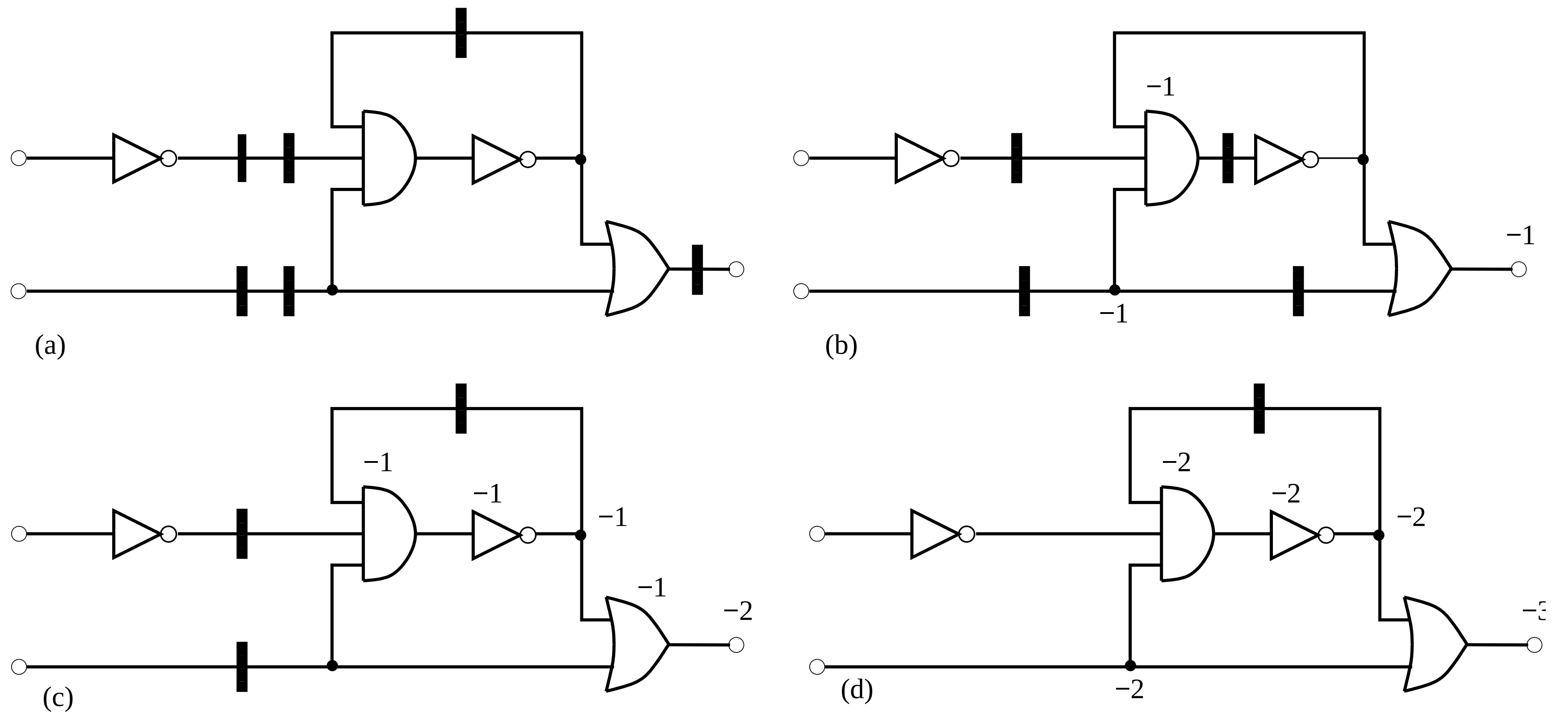
Min-area retiming

- Relocate registers across gates to minimize their count
- Enables dramatic reduction in register count
  - Especially for highly-pipelined designs
- Converts k-th invariants, which hold only *after* time $k-1$, to true invariants

"Transformation-Based Verification Using Generalized Retiming" CAV01



Min-area retiming

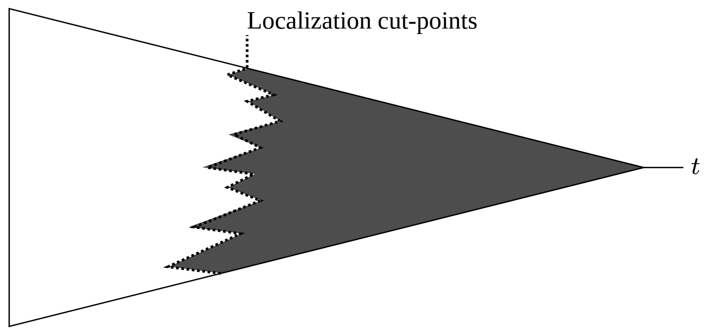


Localization

- Discards unnecessary logic by replacing gates with primary inputs
  - Often yields *dramatic* reductions to netlist size
- Overapproximate abstraction: may yield spurious fails
  - Resolved by *refinement*, adding logic back to the abstraction

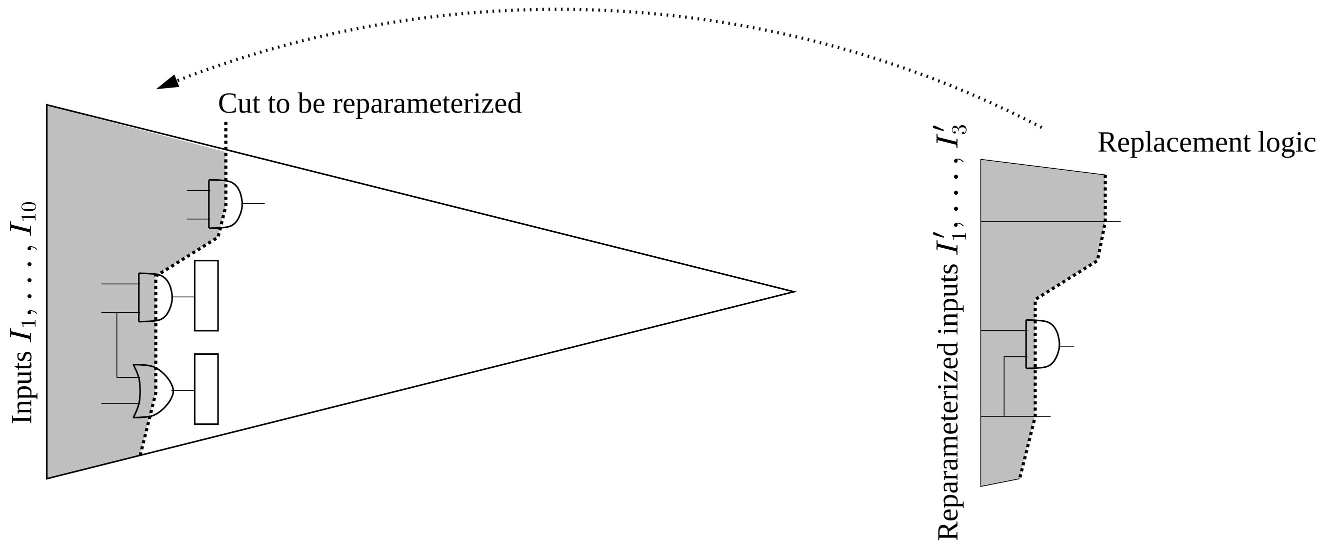
“Counterexample-Guided Abstraction Refinement” CAV00

“Automatic Abstraction without Counterexamples” TACAS04



Input Elimination

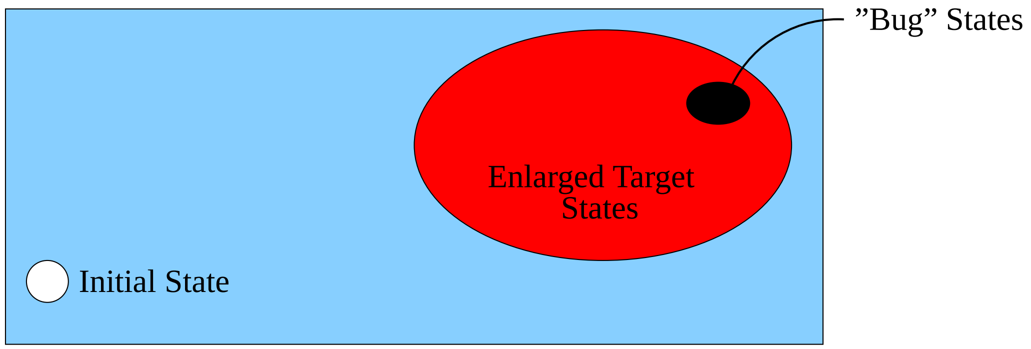
- Identifies a cut in the fanout of original inputs
- Replaces that cut with trace-equivalent logic, with fewer inputs
  - Input elimination helps BDD-based analysis
  - Often reduces gate, register count as well


Cut to be reparameterized
Inputs $I_1, \ldots, I_{10}$
Replacement logic
Reparameterized inputs $I'_1, \ldots, I'_3$

"Maximal Input Reduction of Sequential Netlists via Synergistic Reparameterization and Localization Strategies" CHARME05

Target Enlargement

- Replace property p by states which violate p within k transitions
- Makes falsification easier for semi-formal search
  - Eliminates *probability bottlenecks* during last k transitions
  - *Earlier* falsification is also beneficial
- May also reduce register count; enhance inductivity

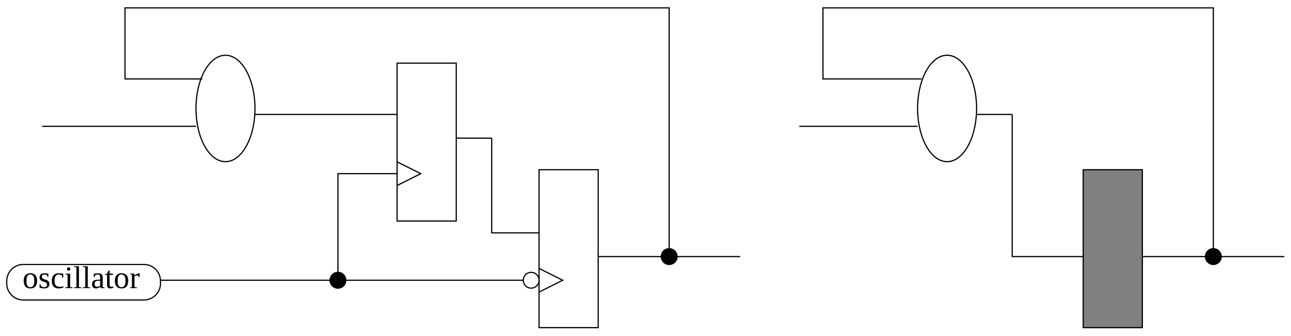


“Property Checking via Structural Analysis” CAV02

“Increasing the Robustness of BMC by Computing Lower Bounds on the Reachable States”, FMCAD04

State-folding Transformations

- Sometimes referred to as *phase abstraction*
  - Unfold next-state functions to reflect k transitions
  - Enables elimination of oscillating clocks
  - Reduces state element count with k-phase latching schemes



"Automatic Generalized Phase Abstraction for Formal Verification" ICCAD05

Property Decomposition

- If two properties are isomorphic, we only need to solve one
  - If one is provably correct, the other must be as well
  - If one falsified, can map counterexample to the other by input substitution

- Can furthermore combine with disjunctive property decomposition

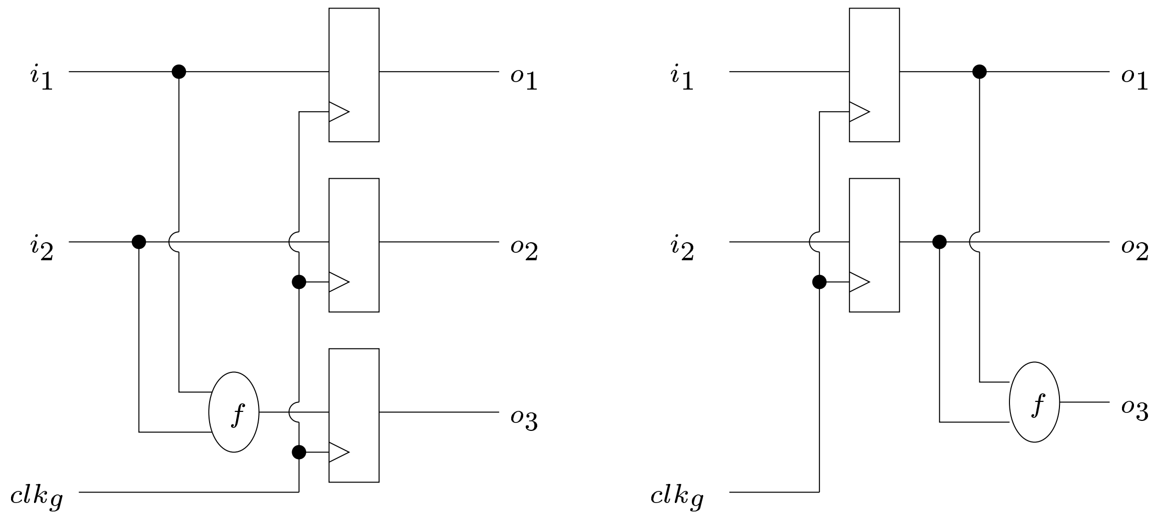
  $a(0,\ldots,n) \not\equiv b(0,\ldots,n)$ rewritten as $\bigvee_{i=0}^{n} a_i \not\equiv b_i$
  - Each bit-slice can be checked independently
  - And, if isomorphic, checking 1 suffices to solve the rest!

- Isomorphisms exist in many designs, though often requires sequential redundancy removal to prune iso-breaking self-test logic

  "Structural Symmetry and Model Checking" CAV98

Sequential Rewriting

- Eliminate registers expressible as deterministic functions of others
  - Based upon approximate reachability
- Complementary to retiming
  - Gated-clock clk_g entails feedback loops freezing registers in place



“Functional dependency for verification reduction” CAV04

Transformation Algorithms assist Proofs

- Reduction in state variables greatly helps enable reachability analysis
- "Tightening" state encoding through redundancy removal, retiming enhances inductivity
  - Inductive proof analyzes 2^N states, minus some that lead to fails
  - Transformations themselves prune some unreachable states
- Reduction alone may solve problems
  - After all, an unreachable property is merely a redundant gate...

"Transformation-Based Verification Using Generalized Retiming" CAV01

"Exploiting state encoding for invariant generation in induction-based property checking" ASPDAC04

Transformation Algorithms assist Falsification

- The smaller the netlist, the faster algos like sim within SFV run
- A smaller netlist often yields exponential improvements to SAT
- Reducing sequential netlist creates *amplified* improvement to SAT
  - Simplify once, unfold many times
  - **Transforms enable deeper exhaustive search**

"Dynamic T.R. Simplification for Bounded Property Checking" ICCAD04

"Exploiting Suspected Redundancy without Proving It" DAC 2005

Transformation Algorithms assist Falsification

- Reduction of sequential netlist, prior to unfolding, is very useful
- Further reducing of the unfolded netlist is also beneficial
  - Unfolding opens up additional reduction potential
  - We leverage a hybrid SAT solver which integrates rewriting and redundancy removal algos with core SAT search
    - ∗ All reasoning is time-sliced for global optimality

“Robust Bool. Reasoning for Equiv. Checking and Functional Pro. Verif” TCAD02

“Improvements to Combinational Equivalence Checking” ICCAD06

Transformation Algos assist Transformation Algos!

- Certain synthesis-oriented transform synergies have been known more than a decade
  - *Retiming and resynthesis*
  - Resynthesis enables more optimal register placement for retiming
  - Retiming eliminates “bottlenecks” for combinational resynthesis
- Many verification-oriented transforms have been discovered more recently

Transformation Algos assist Transformation Algos!

- Localization introduces cut-points, enabling peripheral retiming
- Phase abstraction eliminates feedback loops which prevent retiming
- Localization and retiming enhance potential for input elimination
- Input elimination simplifies logic for localization and retiming
- Redundancy removal reveals isomorphisms hidden by self-test logic
- Rewriting algos open up greater reduction potential for other rewriting algos

SixthSense

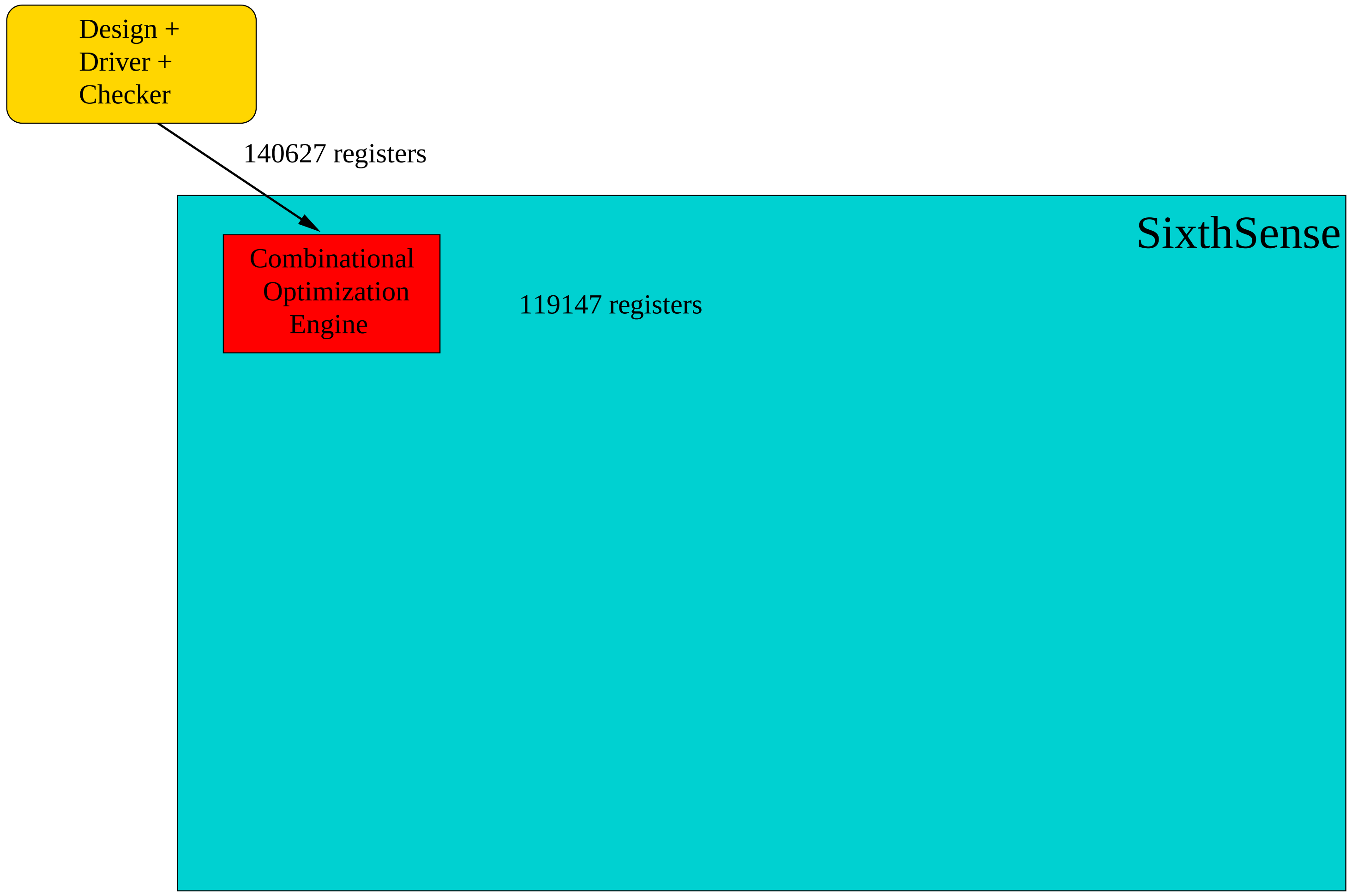
- SixthSense is a system of cooperating algorithms
  - We already introduced several of them:
    - **Transformation** engines
    - **Falsification** engines
    - **Proof** engines

- *Transformation-based verification* framework
  - Modular engine API enables arbitrary sequencing
  - Exploits maximal synergy between various algorithms
    - Incrementally chop problems into simpler sub-problems until tractable for formal reasoning

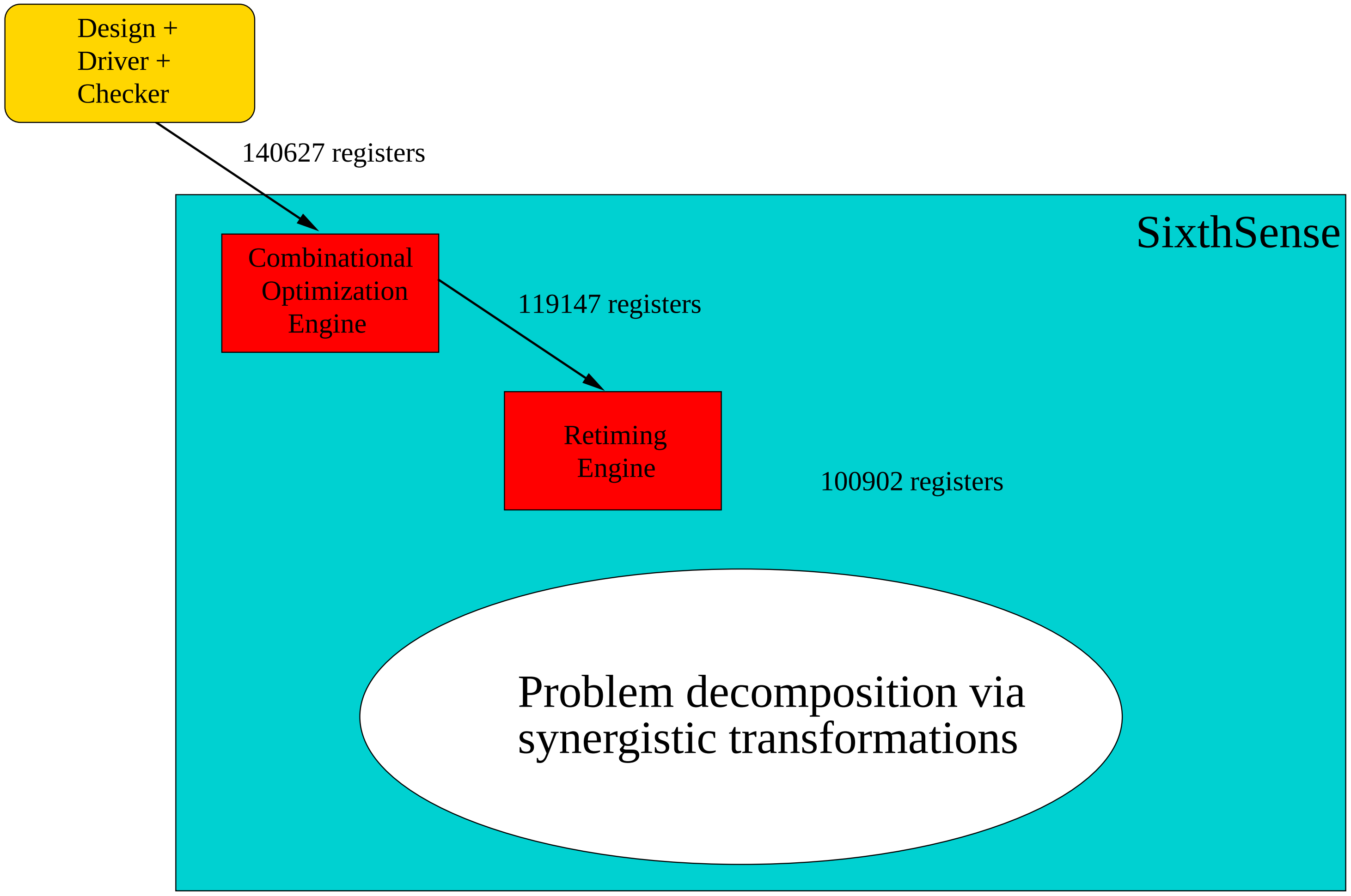
SixthSense

Design +
Driver +
Checker
140627 registers
SixthSense

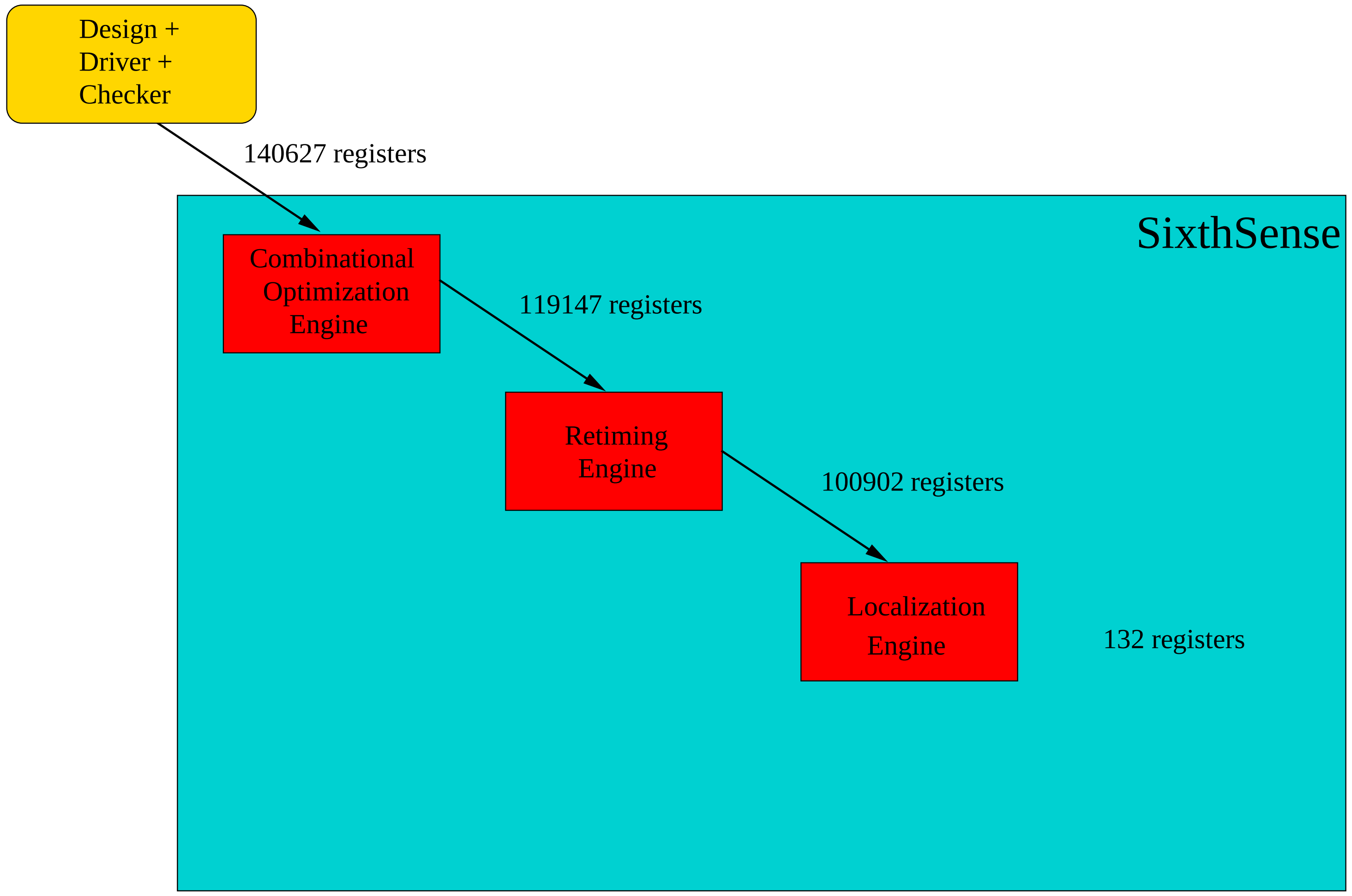
SixthSense


Design +
Driver +
Checker
140627 registers
SixthSense
Combinational Optimization Engine
119147 registers

SixthSense

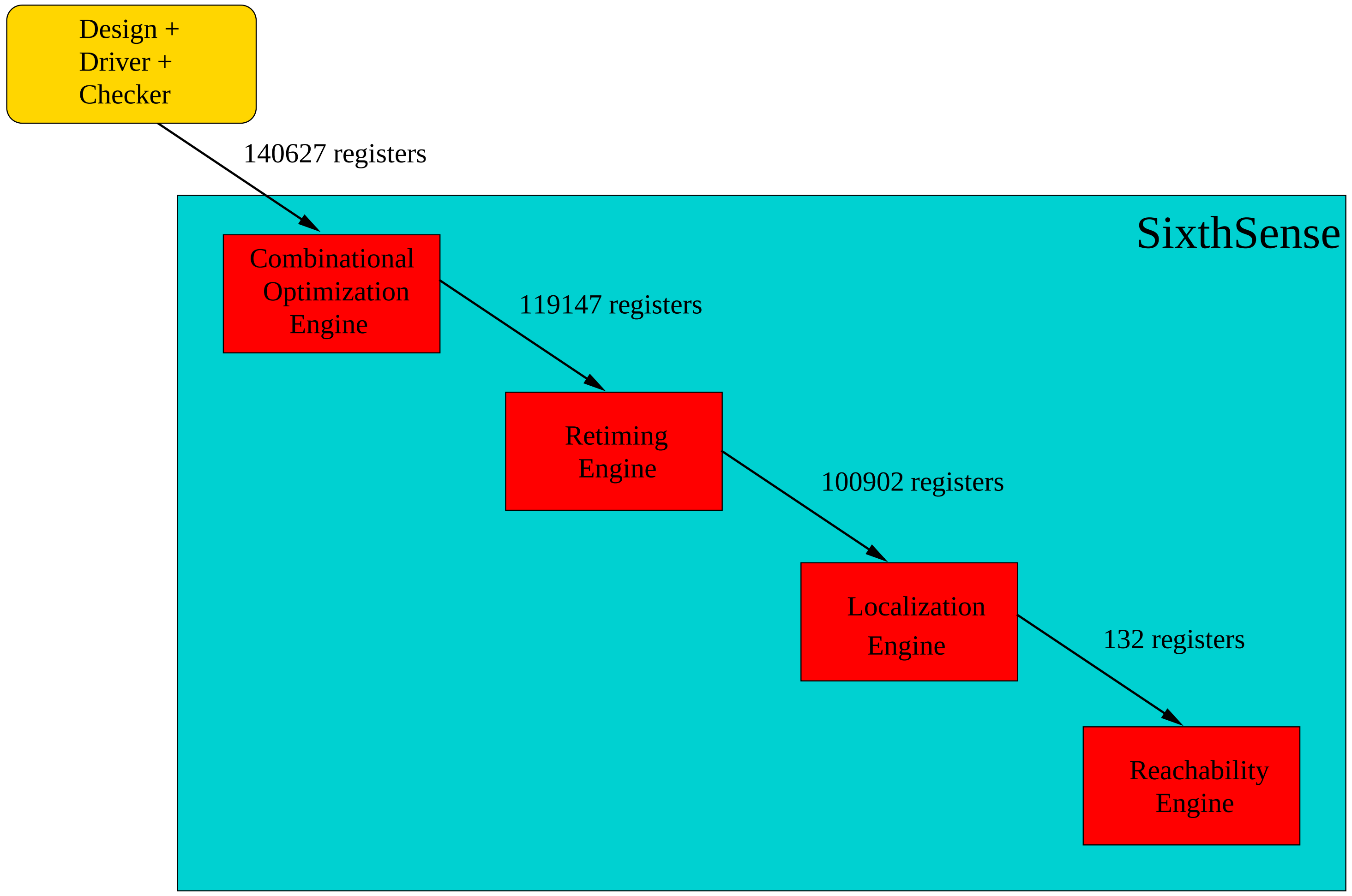

Design +
Driver +
Checker
140627 registers
SixthSense
Combinational Optimization Engine
119147 registers
Retiming Engine
100902 registers
Problem decomposition via synergistic transformations

SixthSense

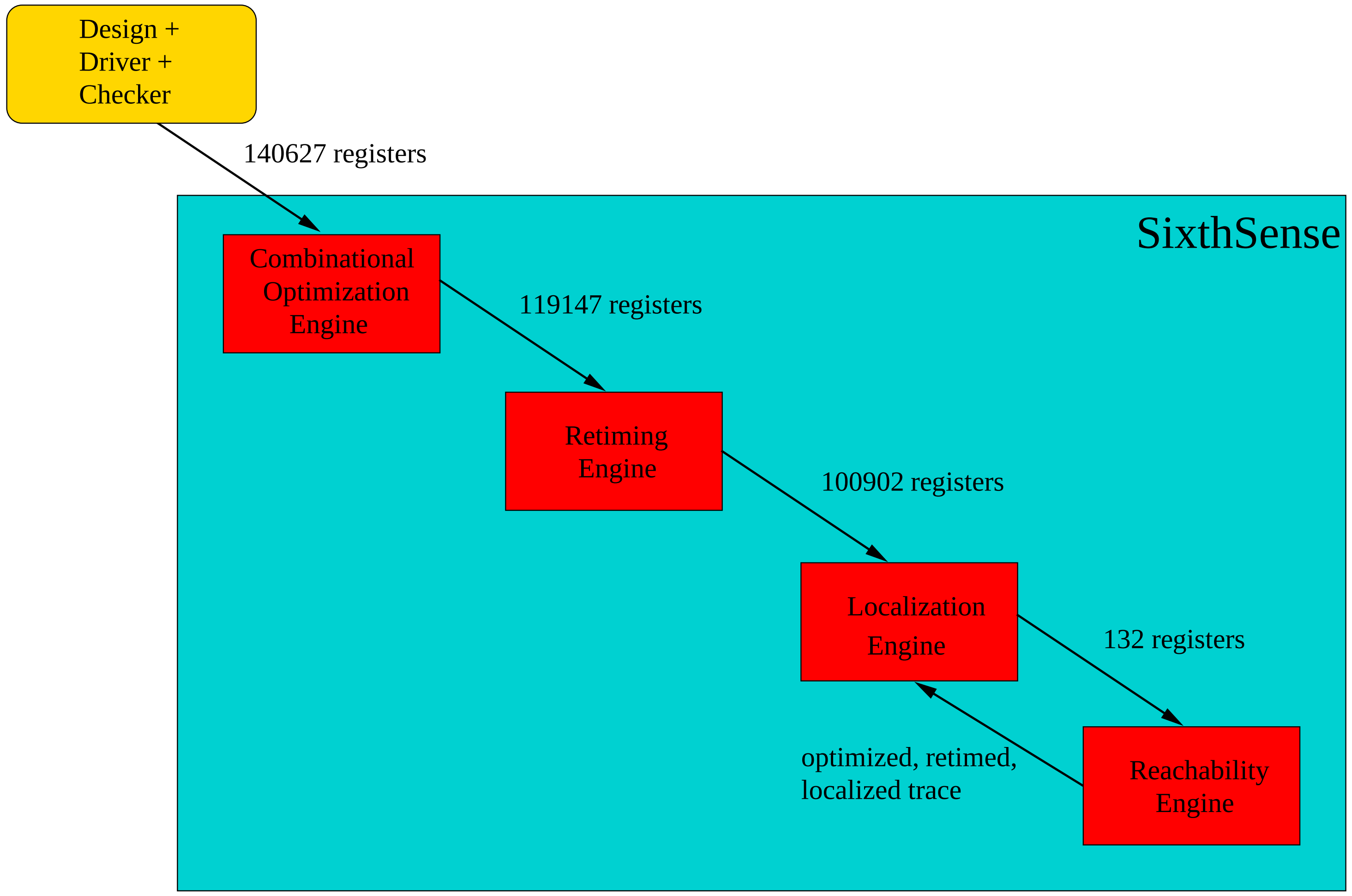


Design + Driver + Checker
140627 registers
SixthSense
Combinational Optimization Engine
119147 registers
Retiming Engine
100902 registers
Localization Engine
132 registers

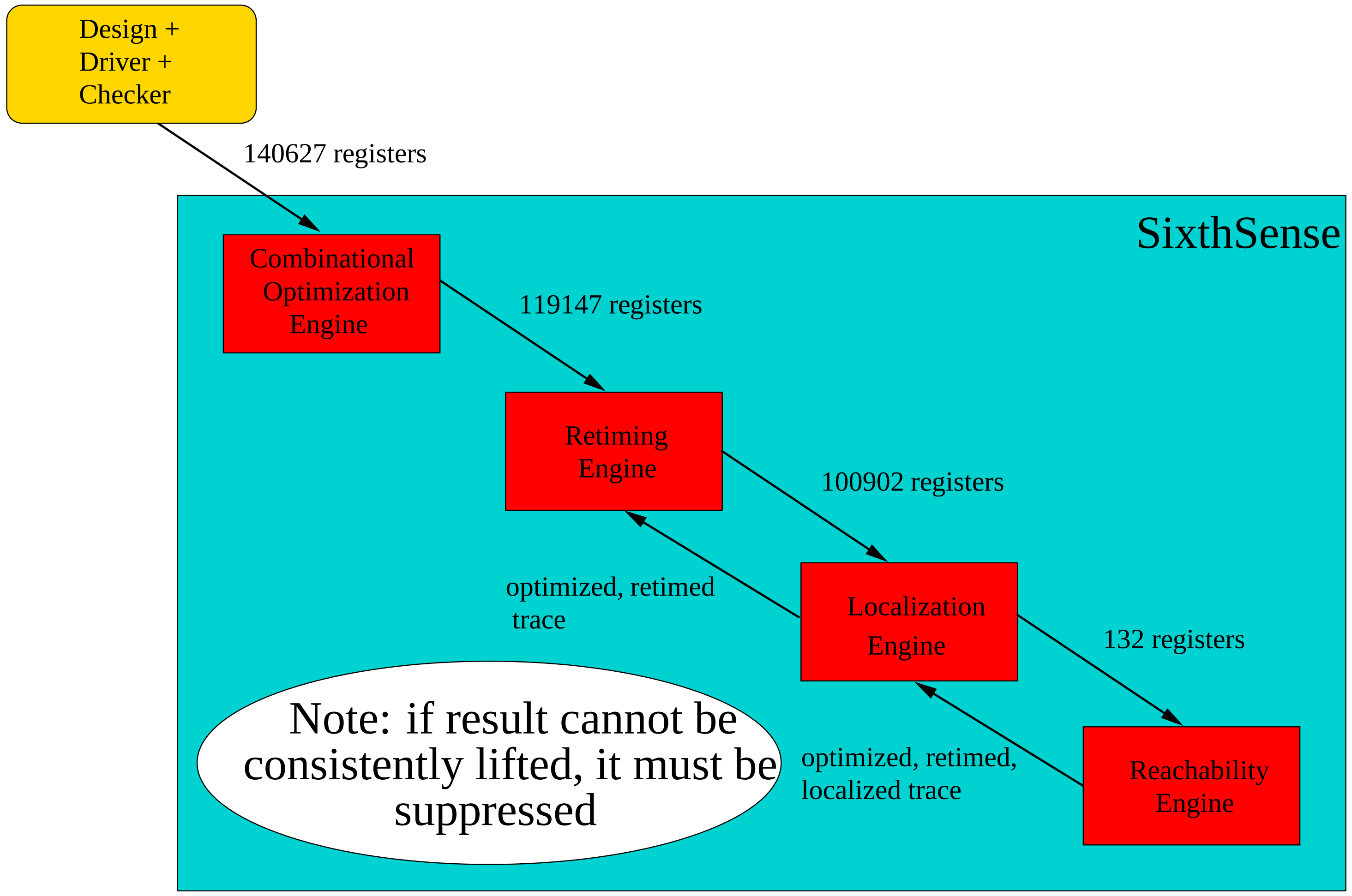
SixthSense


Design +
Driver +
Checker
140627 registers
SixthSense
Combinational Optimization Engine
119147 registers
Retiming Engine
100902 registers
Localization Engine
132 registers
Reachability Engine

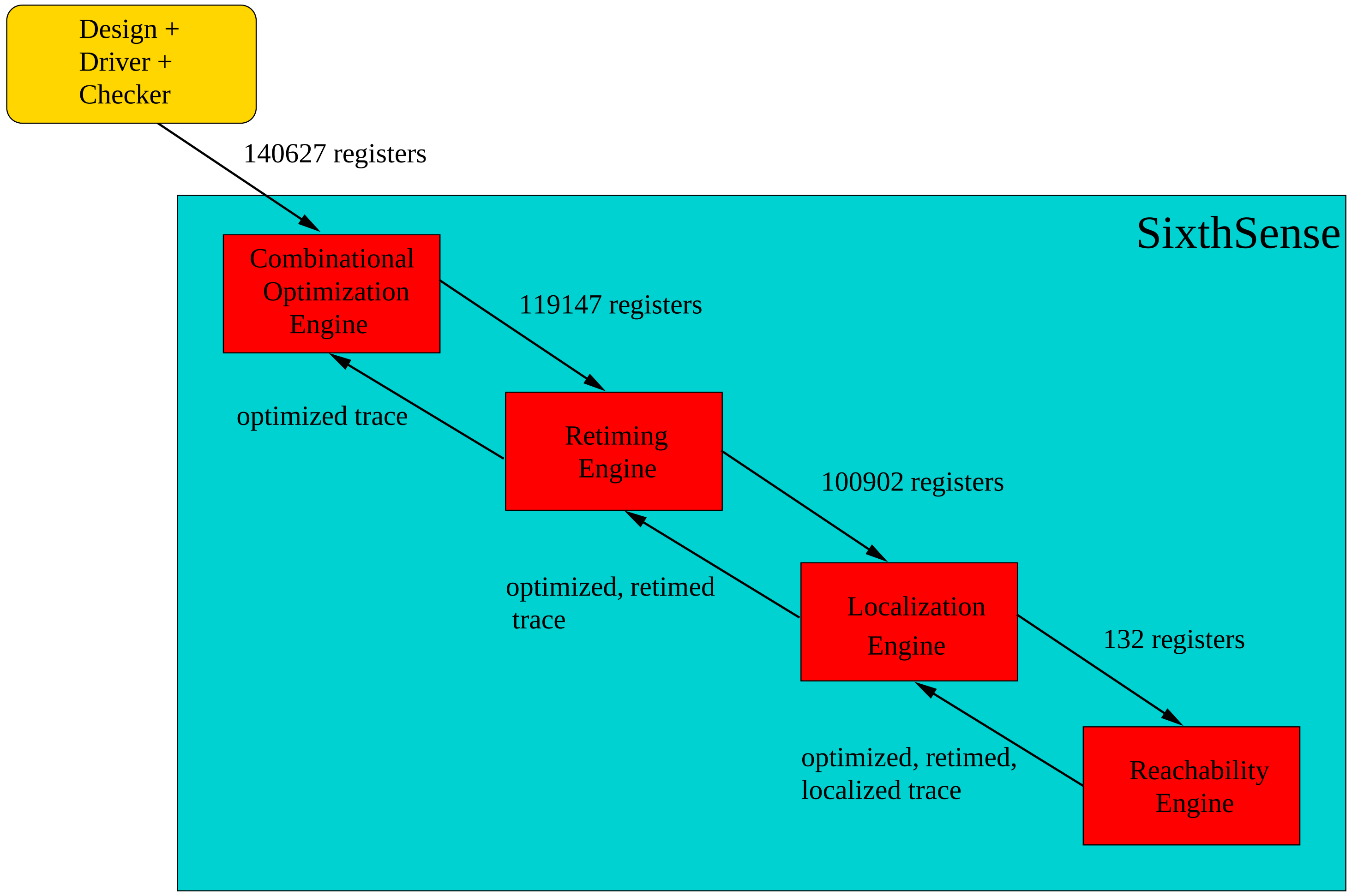
SixthSense


Design +
Driver +
Checker
140627 registers
SixthSense
Combinational Optimization Engine
119147 registers
Retiming Engine
100902 registers
Localization Engine
132 registers
Reachability Engine
optimized, retimed, localized trace

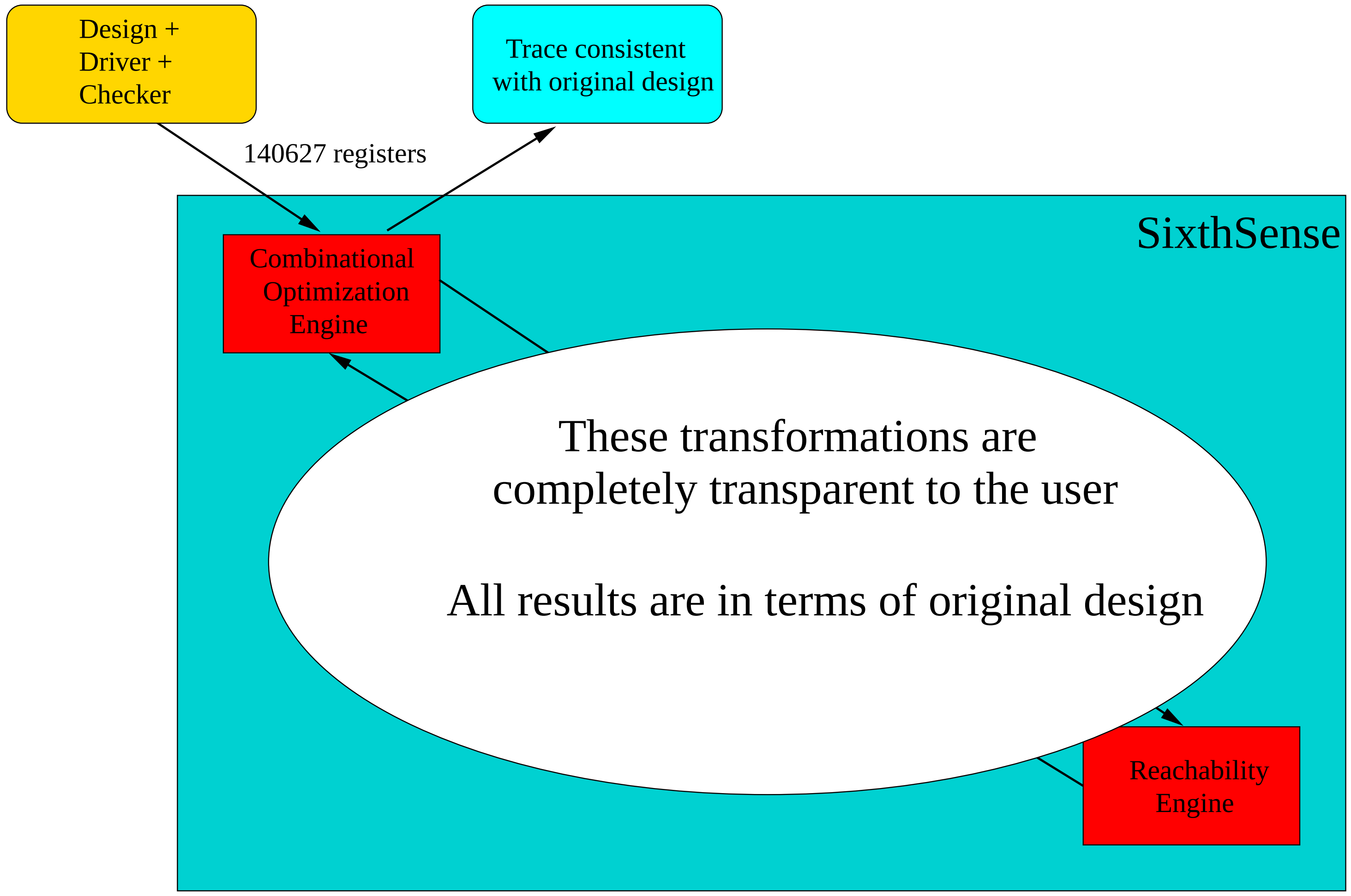
SixthSense


Design +
Driver +
Checker
140627 registers
SixthSense
Combinational Optimization Engine
119147 registers
Retiming Engine
100902 registers
optimized, retimed trace
Localization Engine
132 registers
Note: if result cannot be consistently lifted, it must be suppressed
optimized, retimed, localized trace
Reachability Engine

SixthSense


Design +
Driver +
Checker
140627 registers
SixthSense
Combinational Optimization Engine
119147 registers
optimized trace
Retiming Engine
100902 registers
optimized, retimed trace
Localization Engine
132 registers
optimized, retimed, localized trace
Reachability Engine

SixthSense


Design +
Driver +
Checker
Trace consistent with original design
140627 registers
SixthSense
Combinational Optimization Engine
Reachability Engine

These transformations are completely transparent to the user

All results are in terms of original design

Synergistic Transformations

RING	Initial	COM	EQV	RET	COM	CUT	COM	EQV	BMC 192	
Registers	50988	20768	2320	1932	1930	1930	1930	1841	**FAIL**	
Gates	412804	137588	15434	18114	15691	15467	15086	14194	**24028s**	
Inputs	5313	2730	572	1419	1172	851	850	850	**2.5GB**	
IU	**Initial**	**COM**	**EQV**	**LOC**	**COM**	**MOD**	**LOC**	**CUT**	**EQV**	
Registers	239898	141987	71788	50404	50033	37022	593	582	**PASS**	
Gates	1154650	565513	299243	202935	198512	127826	2356	1905	**2378s**	
Inputs	5548	3020	3018	1775	1775	2905	605	370	**2.9GB**	
MMU	**Initial**	**COM**	**LOC**	**CUT**	**LOC**	**CUT**	**RET**	**COM**	**CUT**	**RCH**
Registers	124297	67117	698	661	499	499	133	131	125	**PASS**
Gates	763475	397461	9901	8916	5601	6605	16381	4645	1300	**1206s**
Inputs	1377	162	1883	809	472	337	1004	287	54	**436MB**

COM comb. optimization / rewriting
RET retiming
MOD state folding
RCH reachability

EQV sequential redundancy removal
CUT input elimination
BMC bounded model checking

Transformation Algo Power

- Redundancy is quite prevalent in high-end designs
  - E.g., lookup queue replication due to placement issues
- Redundancy also arises between design and testbench
  - Reference-model type of verification may be viewed as SEC
- High-performance designs greatly benefit from retiming, state-folding
  - *Though these transforms are useful on virtually all designs*

Why is TBV so Effective?

- Property checking is PSPACE-complete
  - Casting proof as *redundant gate detection* does not alter this fact
  - Clearly, certain transforms are also PSPACE-complete
- Many transforms require only polynomial resources
  - Retiming, phase abstraction, . . .
- Many others can be applied in resource-bounded manner
  - Redundancy removal, sequential rewriting, input elimination, . . .
  - *Trade reduction potential for efficient run-time*

So, Why is TBV so Effective???

- Different algorithms are better-suited for different problems
  - Feed-forward pipeline can be rendered combinational by retiming
    - NP-complete problem hiding in a PSPACE-complete “wrapper”

- More generally: transforms may eliminate facets of design which constitute bottlenecks to formal algorithms
  - Often a *variety* of logic within one industrial testbench
    - Arithmetic for address generation
    - Queues for data-routing
    - Arbitration logic to select among requests
  - Intuitively, optimal solution may rely upon multiple algos

Why TBV is So Effective

- Optimal solution often requires a time-balance between algorithms
  - Algorithmic *synergy* is key to difficult proofs
  - Like time-sliced integration of redundancy removal and SAT
  - Given complexity of property checking, the proper set of algos often makes the difference between *solvable* and *intractable*
- Transforms have substantially simplified almost every problem we have encountered
  - Though clearly a limit to reduction capability of a given set of transforms
  - Then rely upon a strong proof + falsification engine set

Is it Difficult to Find an Optimal (Enough) Algo Flow?

- Countably infinite number of algo configurations
- For the most difficult problems, may require dozens of nested engines
- Finding the best-tuned config is a very difficult problem
- How can we make this technology as easy to use as sim?
  - Cannot require users to learn about algos
  - Cannot require people to hand-tune configs

Automation in TBV

- SixthSense users ask for assistance tuning configs, if default run does not solve problem
- We could not pre-package a specific config set as effective as our expert hand-tuning
  - Customization of configs necessary for difficult problems
  - Yet - the “algo” used to manually tune configs not too complex...
- Solution: develop *expert system* engine to automate config experimentation, customization
  - We still *teach* rules to the expert system (and vice versa)
  - Though expert system substantially reduced need for hand-tuning

“Scalable Automated Verification via Expert-System Guided Transformations”
FMCAD04

Reusable Testbenches

Increasing Verification ROI

- Human resources are the most expensive aspect of HW design
  - Tightening resources necessary to sustain any business
- Verification is increasingly becoming dominant factor
  - Expensive to have 2 people specify same design (1 sim, 1 formal)
  - Goal: reuse spec for both domains
  - Reuse is cost-effective
  - Also enables sim team, designers to write formal specs

Sim / (S)FV Testbench Reuse

- Requires scaling to unit-level testbenches
  - More meaningful than block-level testbenches
    - Better-documented interfaces to *drive*
    - More encompassing properties to *check*
  - **Verify functionality** vs. **verify blocks**
  - More cost-effective
    - Fewer testbenches necessary to *cover* design
- Refer to Tuesday panel discussion by Roesner

Testbench Reuse vs. Proof Strategy

- Reusable TBs developed for “ease of specification” vs “ease of proof”
  - If proofs vs. SFV is (realizable) goal, is this wasted effort?
- High cost associated with manual decomposition of complex proofs
  - SFV, multi-algo proof solution is very powerful to address
  - Falsification wrings out early bugs quickly
  - Proofs are also often possible even on large testbenches
- When proofs go through, no need for manual decomposition!
- Else, manual decomposition done *after* wringing out bugs with SFV

Scaling to Sim-Sized Testbenches

- Scaling to sim-sized testbenches has many advantages
- However, even with SFV and multi-algo solution, still falls short of sim-sized testbenches
  - Sim can be done at μproc core level, chip level, system level!
  - Sim tricks like logic folding / parallel evaluation, abstract RAM representation, . . . difficult to exploit in unbounded model checking
- Much room for improvement in automated (S)FV technologies
  - **HW verification is not a solved problem!**
- Refer to Thursday presentation by Jacobi

SixthSense Applications

SixthSense Applications

1. Sequential equivalence checking
   - IBM uses *Verity* for CEC
   - SixthSense used for SEC
     - Validate sequential optimizations for timing, power, . . .
     - Check conditional equivalence between design evolutions
     - Quantify, validate late design fixes
   - Saves verification resources; enables greater design optimality
   - Refer to Thursday presentation by Paruthi

SixthSense Applications

2. Designer-level applications
   - Assertion-based verification
   - SEC for exploring optimal design space
   - Leveraging FV feedback for design optimization
     - ∗ Unreachable states; redundancy; power optimization potential
   - Scalability important in these applications
     - ∗ Testbenches created with little thought of “proof strategy”
     - ∗ Users have lesser experience with FV, toolset

- *RuleBase* also used for such applications

SixthSense Applications

3. Block-level verification
   - Traditional FV deployment
   - Dedicated testbenches, often developed by verification team
   - Due to scarce resources, blocks selected by prioritization
     - ∗ Most complex
     - ∗ Recent / late modifications
     - ∗ Hardest to test
     - ∗ In response to *bug curve*
   - Sometimes result from decomposition of unit-level testbenches

- *RuleBase* also used for such applications

SixthSense Applications

4. Unit-level verification
   - Matches the level at which a sim testbench is developed
   - *Synthesizable testbench* used for FV enables optimal reuse
     - ∗ (S)FV, sim, emulation
   - Scalability critical due to size
   - Many proofs are achievable even on unit-level testbench
     - ∗ No need to even reuse in sim, or decompose to block-level!
   - Refer to Monday presentation by Glökler

SixthSense Applications

5. Lab fail recreation: **When a chip misbehaves...**
   - (S)FV increasingly relied upon for such applications
     - ∗ Invariably a corner-case bug to have evaded so much validation
     - ∗ If sim could not find original bug, how much confidence would it yield that the fix is robust?
   - On-chip debug facilities offer partial insight into cause
   - Usually have a good idea of property to check and “buggy region”
   - Scalability critical since often requires a fairly large design slice
     - ∗ SFV useful since bug-hunting vs. proving is “the mission”

Conclusion

Conclusion

- IBM has successfully used model checking for ~ 10 years
  - Nonetheless, sim remains predominant validation methodology
- Open question whether a design as complex as POWER can be fully formally verified
  - Very large and complex design
  - HDL acquires circuit characteristics due to CEC reliance
  - SEC may ultimately help augment this complexity

Conclusion

- SixthSense philosophy: *non-intrusive FV*
  - Scale FV to sim-sized testbenches
    - Integrate SFV algorithms
    - Integrate a variety of synergistic proof + transform algos
  - Ensure high automation, ease of use
- Push for reusable testbenches across sim + FV
  - Greater ROI of specification investment
- Result: substantially wider-spread use of FV

Future Work

- HW verification is not a solved problem

- **MUCH** room for improvement in automated proof algorithms, abstraction / reduction algorithms, semi-formal algorithms
  - Ideal if applicable to bit-level netlists
  - We consider this to be the most critical direction

- Additional insight into methodologies for verifying a wider variety of industrial-complexity designs

- Additional insight into how to effectively leverage, automate light-weight theorem proving for a wider variety of problems