ATG SoC

HECTOR: Formal System-Level to RTL Equivalence Checking

Alfred Koelbl, Sergey Berezin, Reily Jacoby,

Jerry Burch, William Nicholls, Carl Pixley

Advanced Technology Group

Synopsys, Inc.

June 2008

Outline

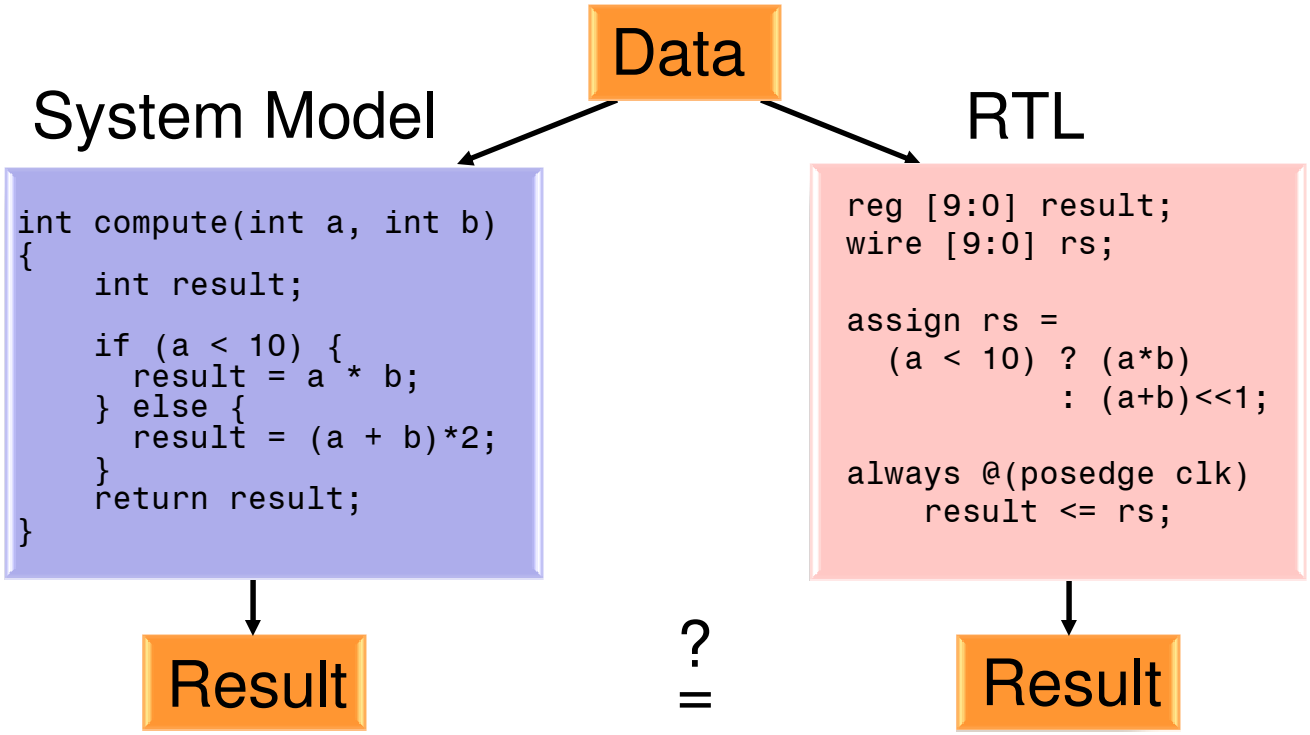
- Motivation
- Architecture of Hector
  - Frontend
  - Notions of equivalence and interface specification
  - Proof procedure
  - Solvers
  - Debugging
- Customer results
- Additional applications
- Conclusion

MOTIVATION

System-level design

- Some reasons for system-level design:
  - Faster verification at the system-level
  - Easier architectural exploration
  - No need to worry about implementation details
  - Productivity gain by using High-Level-Synthesis
- RTL Verification problems:
  - Verification of RTL doesn't get any easier
  - Bugs due to faulty specification
  - Bugs due to wrong implementation

Functional equivalence checking



- **System-level model is a transaction/word level model for the hardware**
- **System and RTL compute same outputs given same inputs**
- **Equivalence checking proves functional equivalence**
- **Timing and internal structure can differ significantly, but the observable results must be the same**

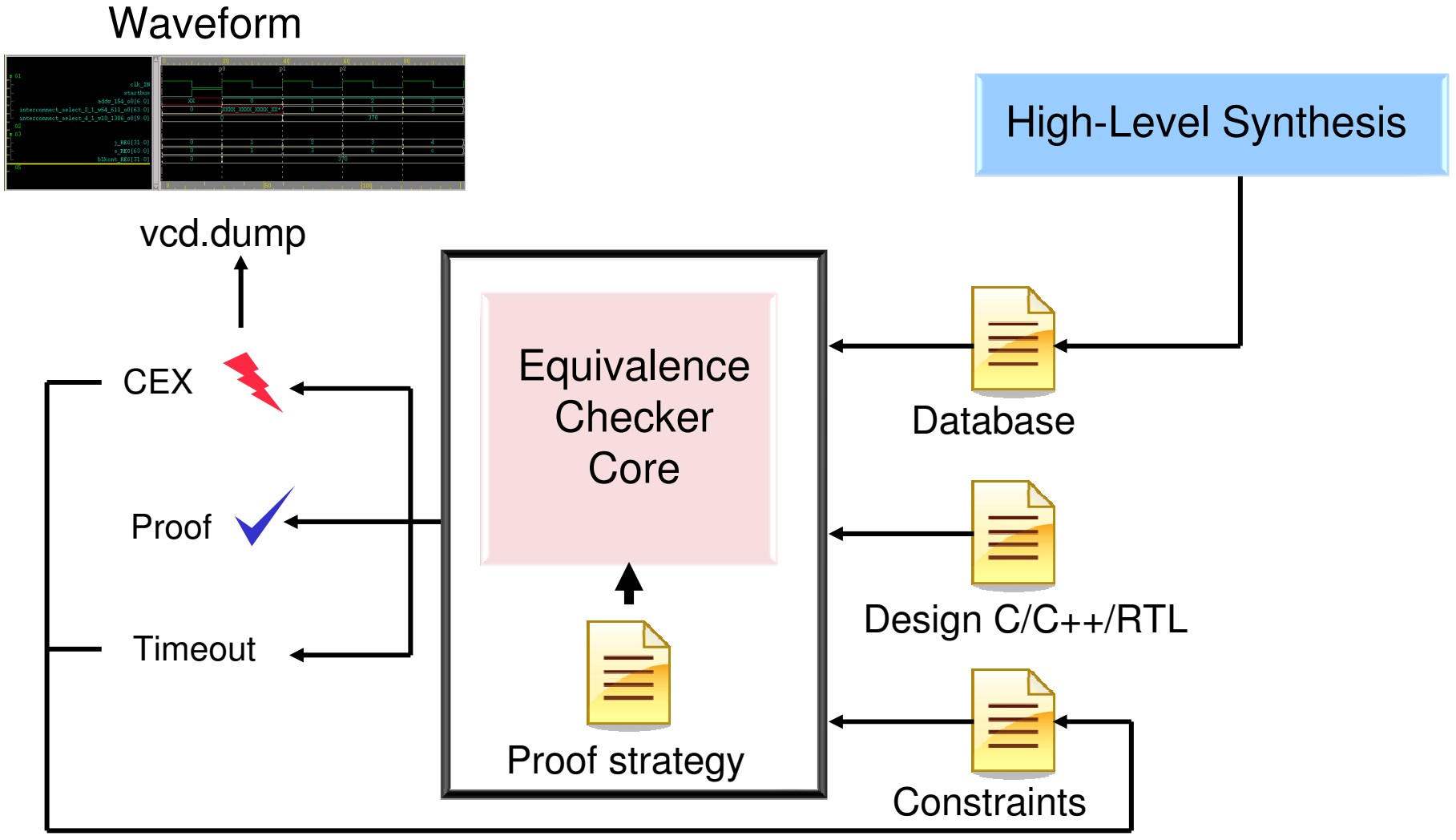
Manual (Ad hoc) Flow

- Architect creates C++ specification
- RTL designer creates RTL implementation
- RTL contains much more implementation details
- Problems:
  - Designs often embedded in own simulation environment, need to specify input/output mapping, notion of equivalence
  - Specification and implementation can be significantly different
  - Constraints are often in designer's head, need to be formalized
  - Input/output differences sometimes difficult to capture in a formal model

High-Level Synthesis Flow

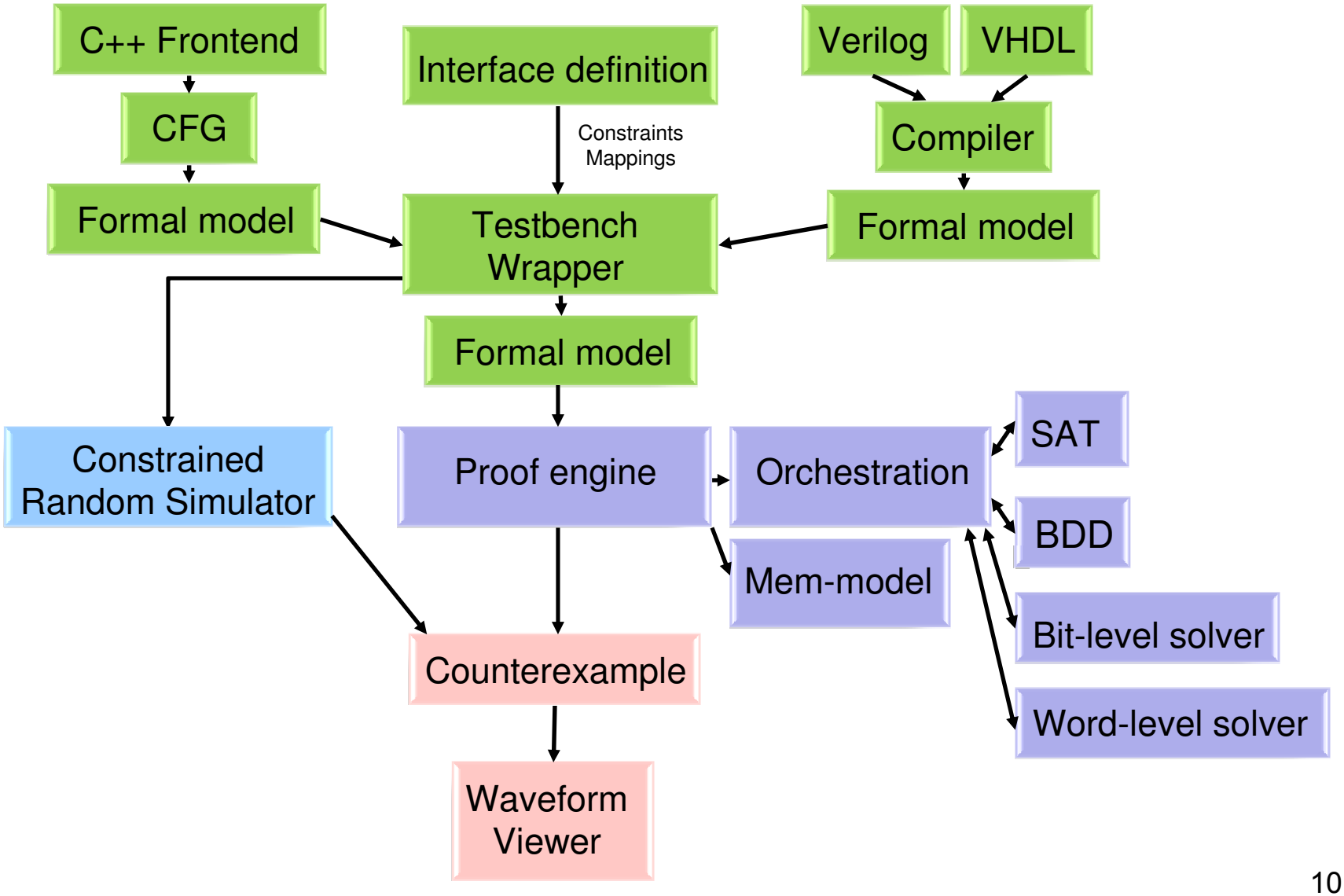
- Equivalence checker proves correctness of produced RTL
- You cannot sell a high level synthesis tool without a verification tool!!!
- Advantages:
  - All information about constraints & interface mappings / latency differences available from the synthesis tool
  - Hints can significantly simplify proof
  - Push-button solution possible
- Problems:
  - Every assumption given as hint must be proven by equivalence checker
  - High-level synthesis tool must be able to produce the information

HL-Synthesis integration

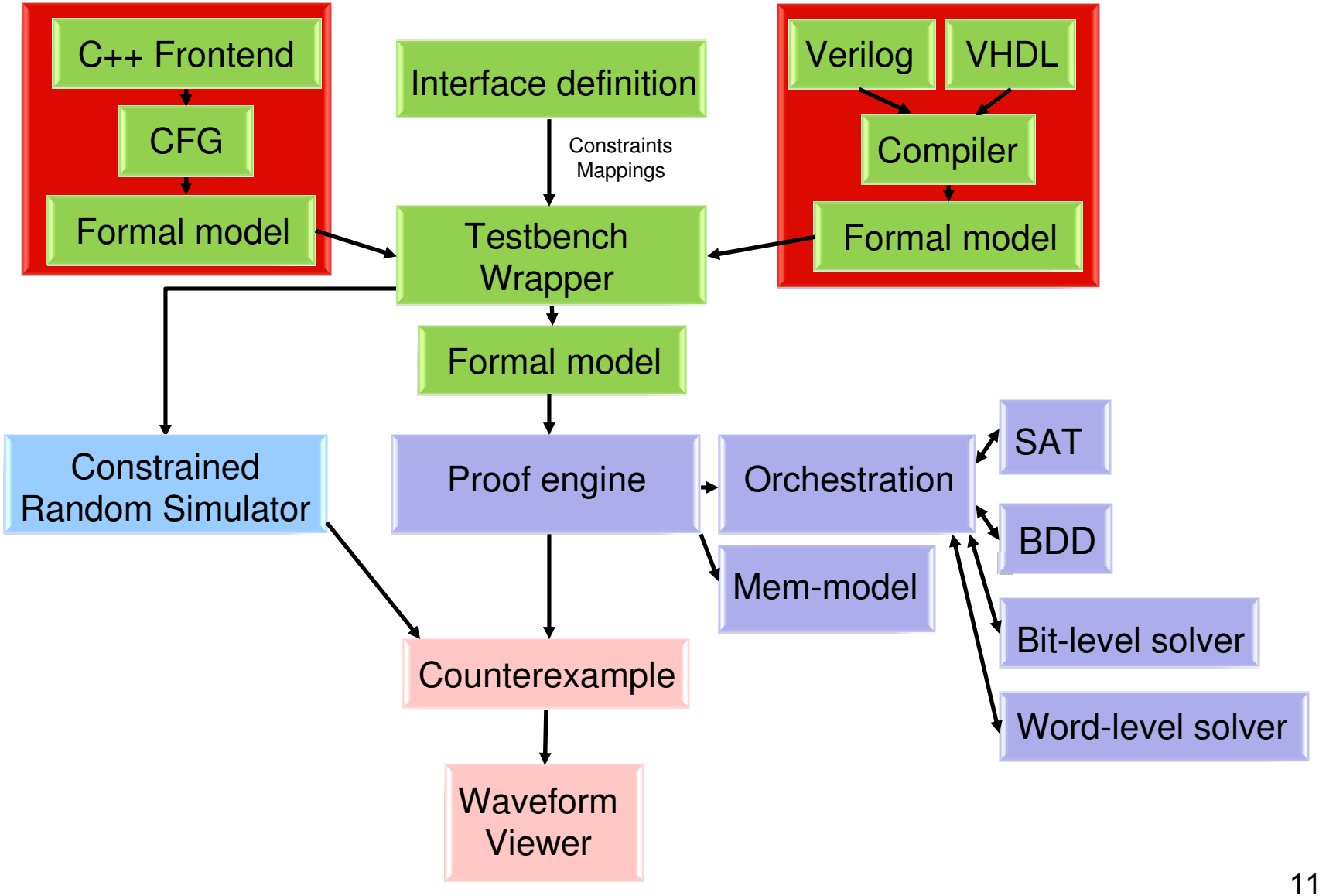


ARCHITECTURE

Components

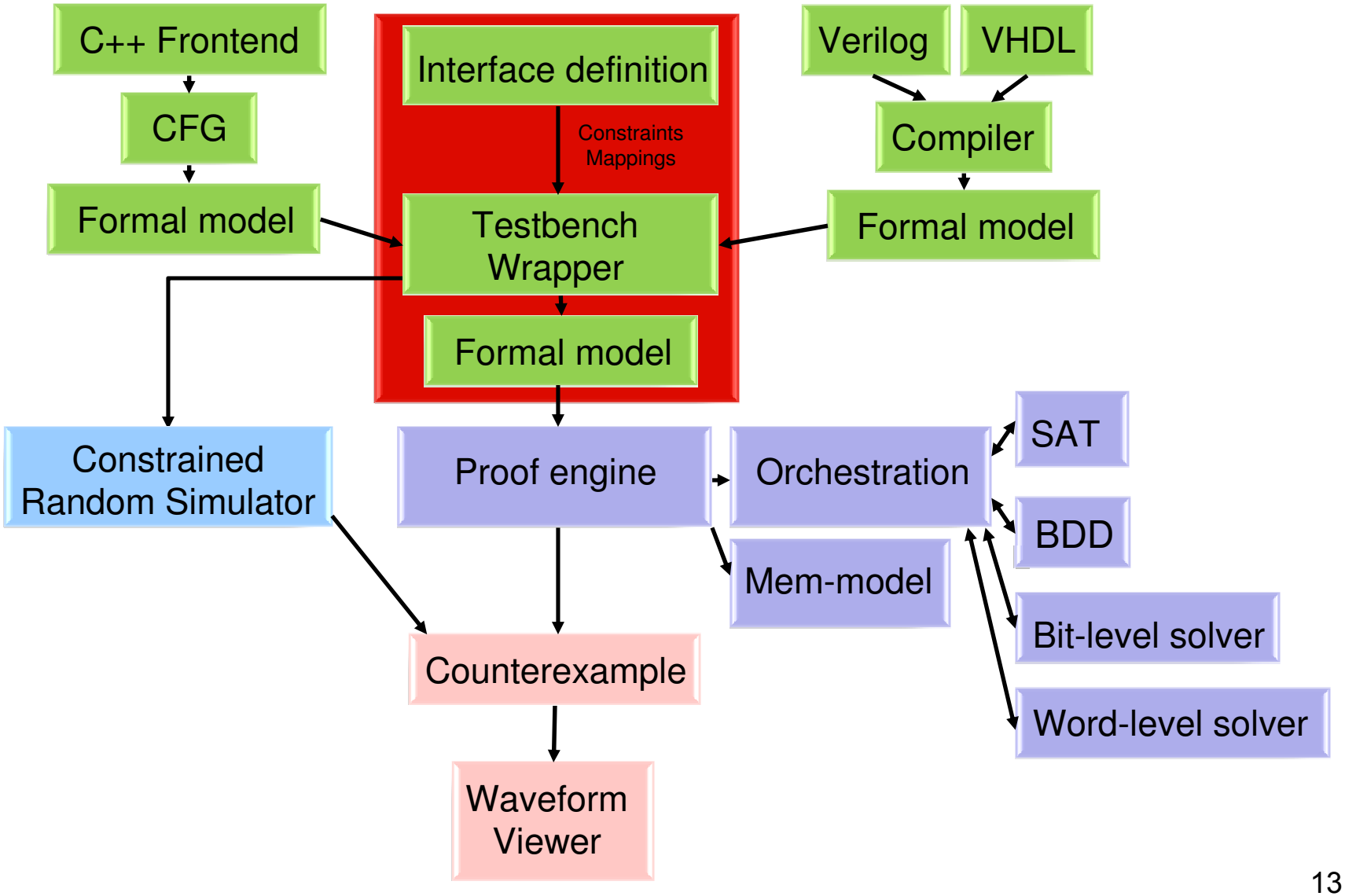

C++ Frontend
CFG
Formal model
Interface definition
Constraints
Mappings
Testbench
Wrapper
Formal model
Verilog
VHDL
Compiler
Formal model
Constrained
Random Simulator
Proof engine
Orchestration
SAT
BDD
Mem-model
Bit-level solver
Word-level solver
Counterexample
Waveform
Viewer

Front-End


C++ Frontend
CFG
Formal model
Interface definition
Constraints
Mappings
Testbench
Wrapper
Verilog
VHDL
Compiler
Formal model
Formal model
Constrained
Random Simulator
Proof engine
Orchestration
SAT
BDD
Mem-model
Bit-level solver
Word-level solver
Counterexample
Waveform
Viewer

INTERFACE SPECIFICATION

Interface specification



Notions of equivalence

- What does equivalence mean for comparing system-level models against RTL ?
  - Depends on how abstract the system-level model is
  - Different customers, different applications
  - Different design styles
  - No definite answer (yet)

- Identify commonly used notions:
  - Combinational equivalence
  - Cycle-accurate equivalence
  - Pipelined equivalence
  - Stream-based equivalence
  - Transaction equivalence
  - … ?

How to deal with different notions ?

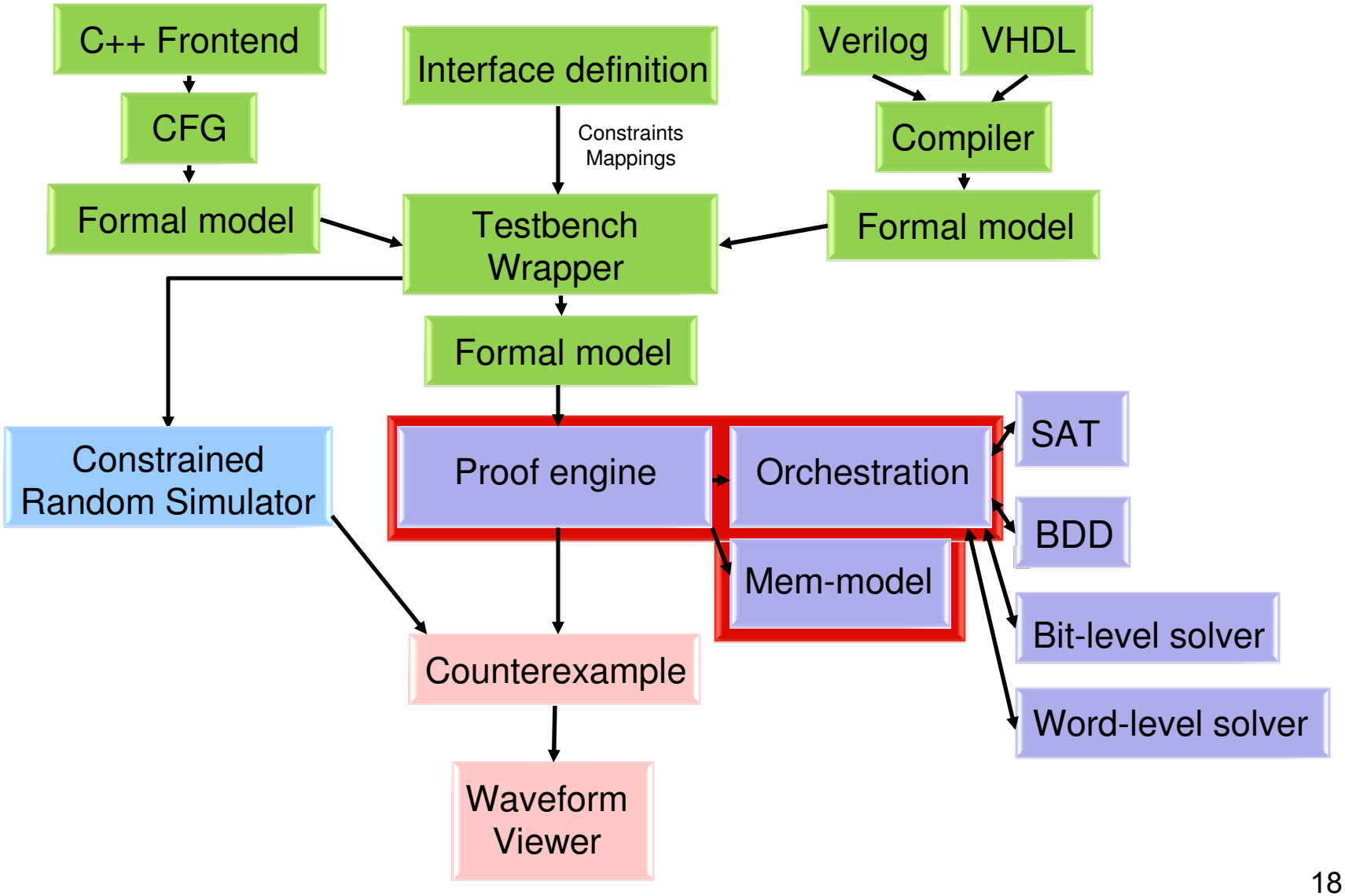
- Idea: Reduction to cycle-accurate equivalence check
- Rule of thumb: If you can build random pattern testbench, checking outputs on the fly, you're safe.

Verification wrapper generation

- User (or synthesis tool) provides the following information:
  - Input/output mapping between C++ and RTL
  - Input constraints
  - Output don’t cares
  - Memories / memory mappings
  - Register mappings
  - Notion of equivalence (optional)
- Verification wrapper is automatically generated
- Reduces problem to cycle-accurate sequential equivalence check

PROOF PROCEDURE

Proof procedure



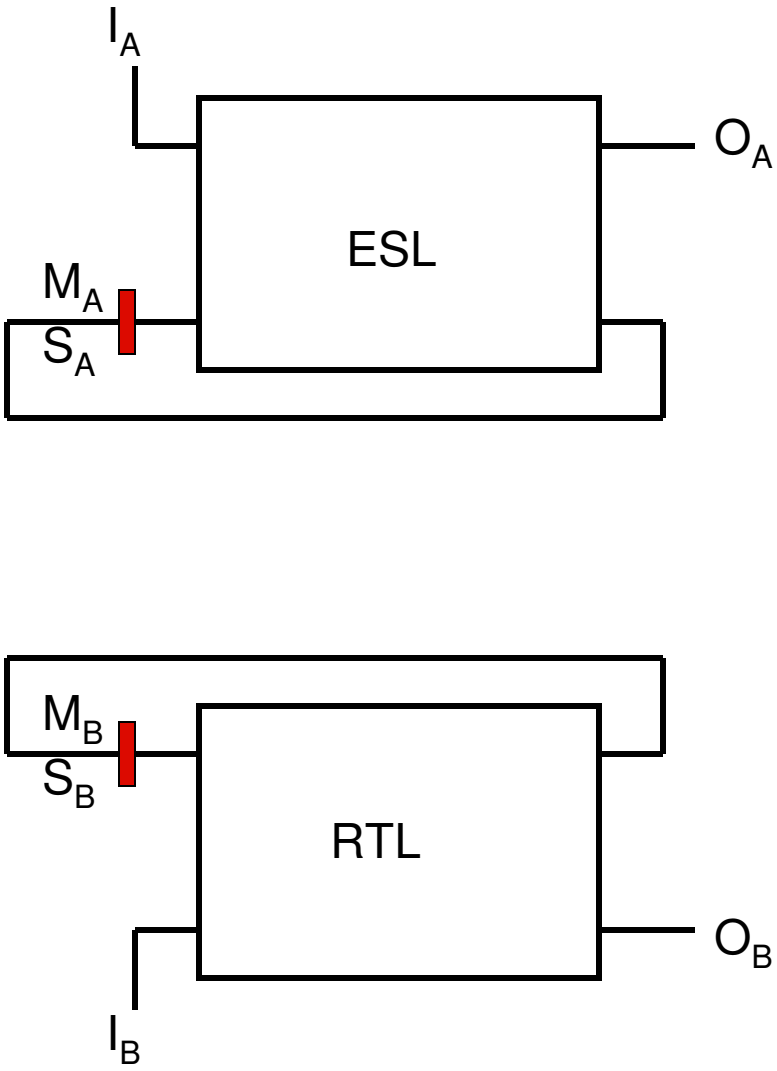
Verification approach

- Constrained Random simulator checks for easily detectable discrepancies
- Bounded formal check for harder discrepancies
- Formal proof (complete):
  - Problem reduced to sequential equivalence checking
  - Reachability analysis would be an approach
  - But: Most system-level designs are arithmetic heavy, reachability infeasible
  - Induction proof
- Proof idea:
  - Implementation and specification perform same computations
  - Not necessarily in the same number of cycles
  - Unroll for the duration of a transaction, prove that symbolic expressions are the same
- Proof engines:
  - Bit-level equivalence checkers (SAT, BDDs)
  - Word-level rewriting engine for arithmetic (COMBAT)
  - Hybrid (word & bit) engine for orchestration
  - PEP's

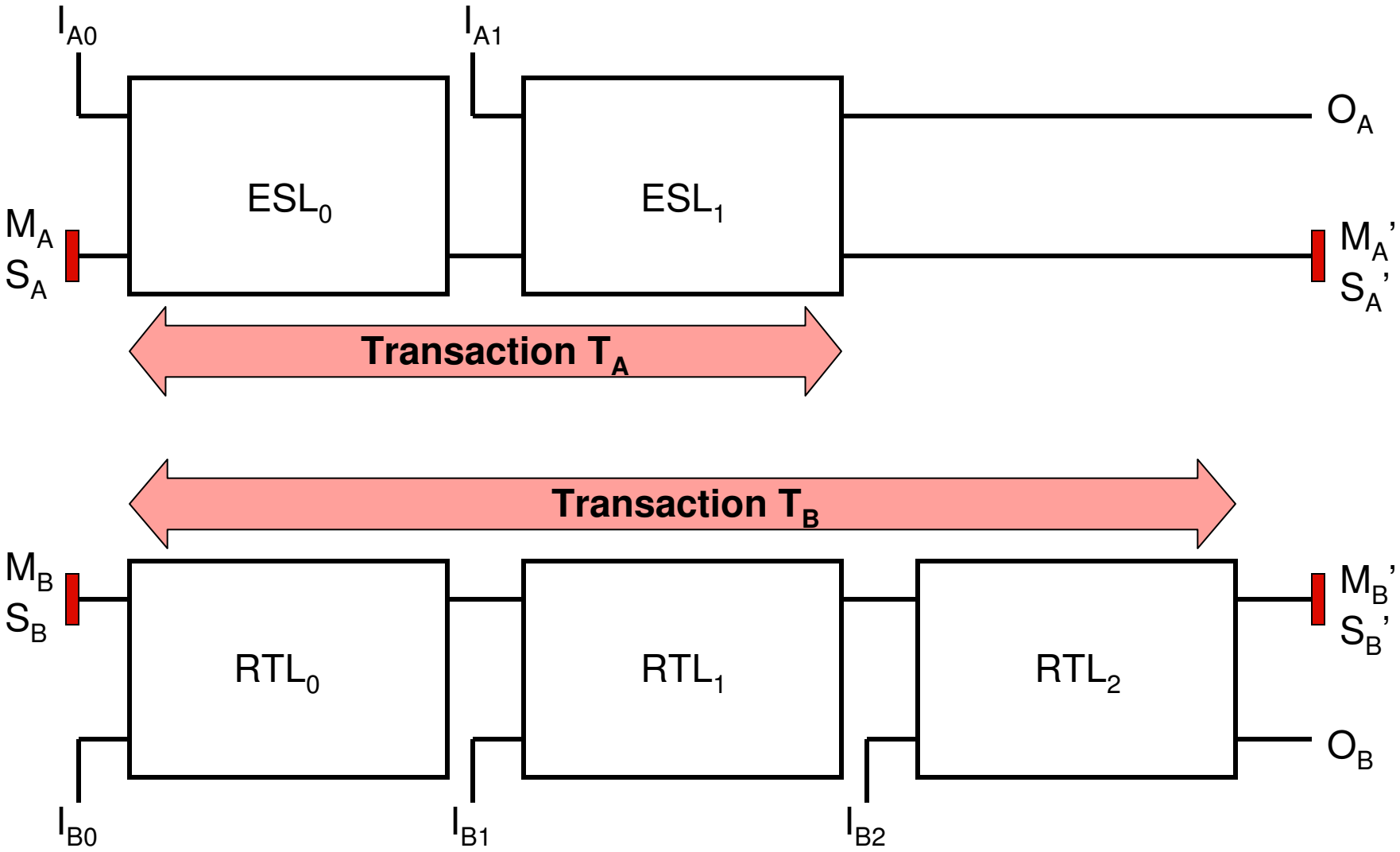
Induction proof

- Transaction equivalence
  - Assume that designs start in valid state (superset of reachable state set)
  - Execute single transaction by unrolling ESL and RTL models for one transaction
  - Check outputs after transaction
  - Check state after transaction
- Proof strategy: Induction
- Needs state invariants
  - Register mappings
  - Memory mappings & memory constraints
  - Additional invariants
- Prove that resulting SAT formula is UNSAT

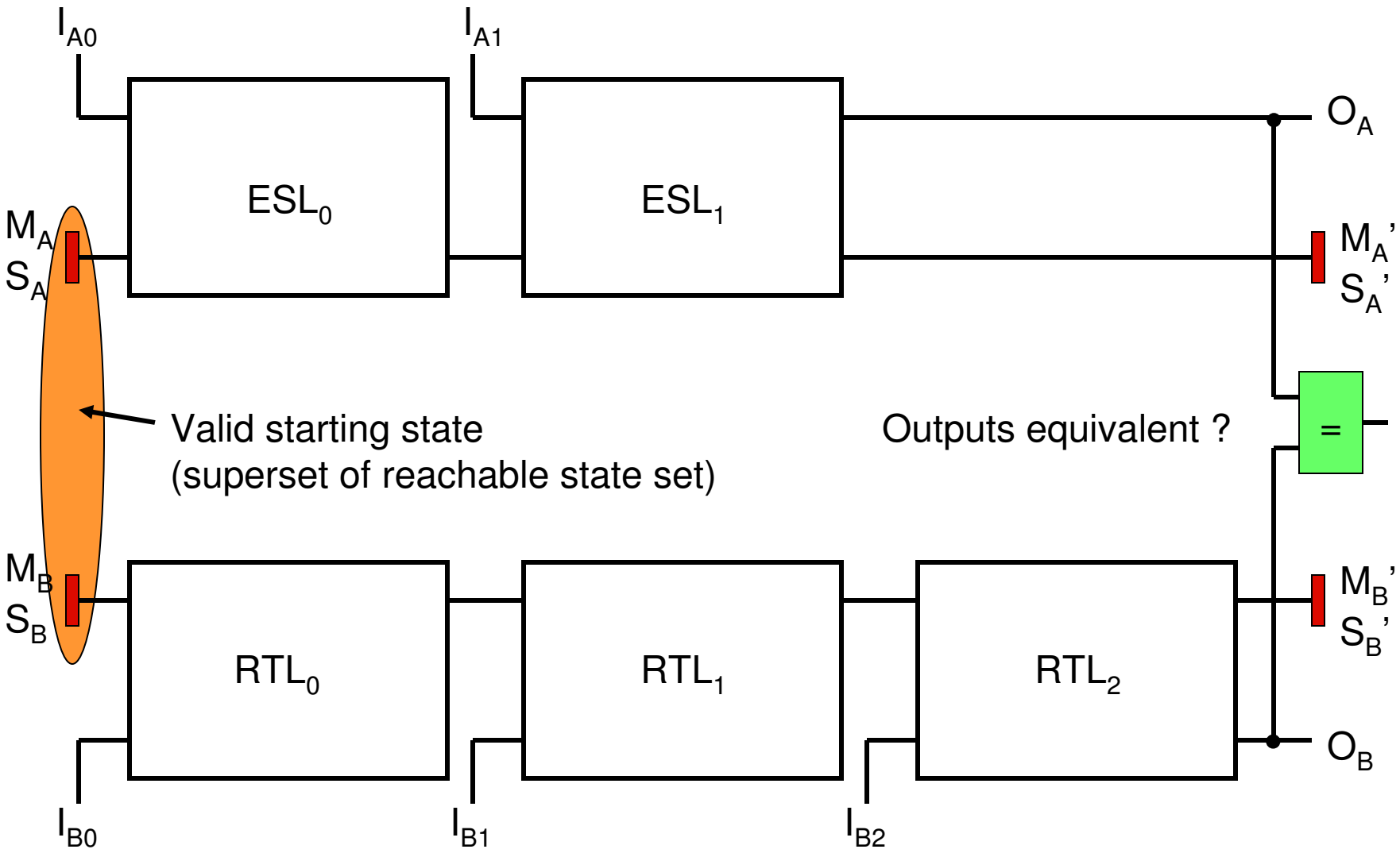
Transaction equivalence



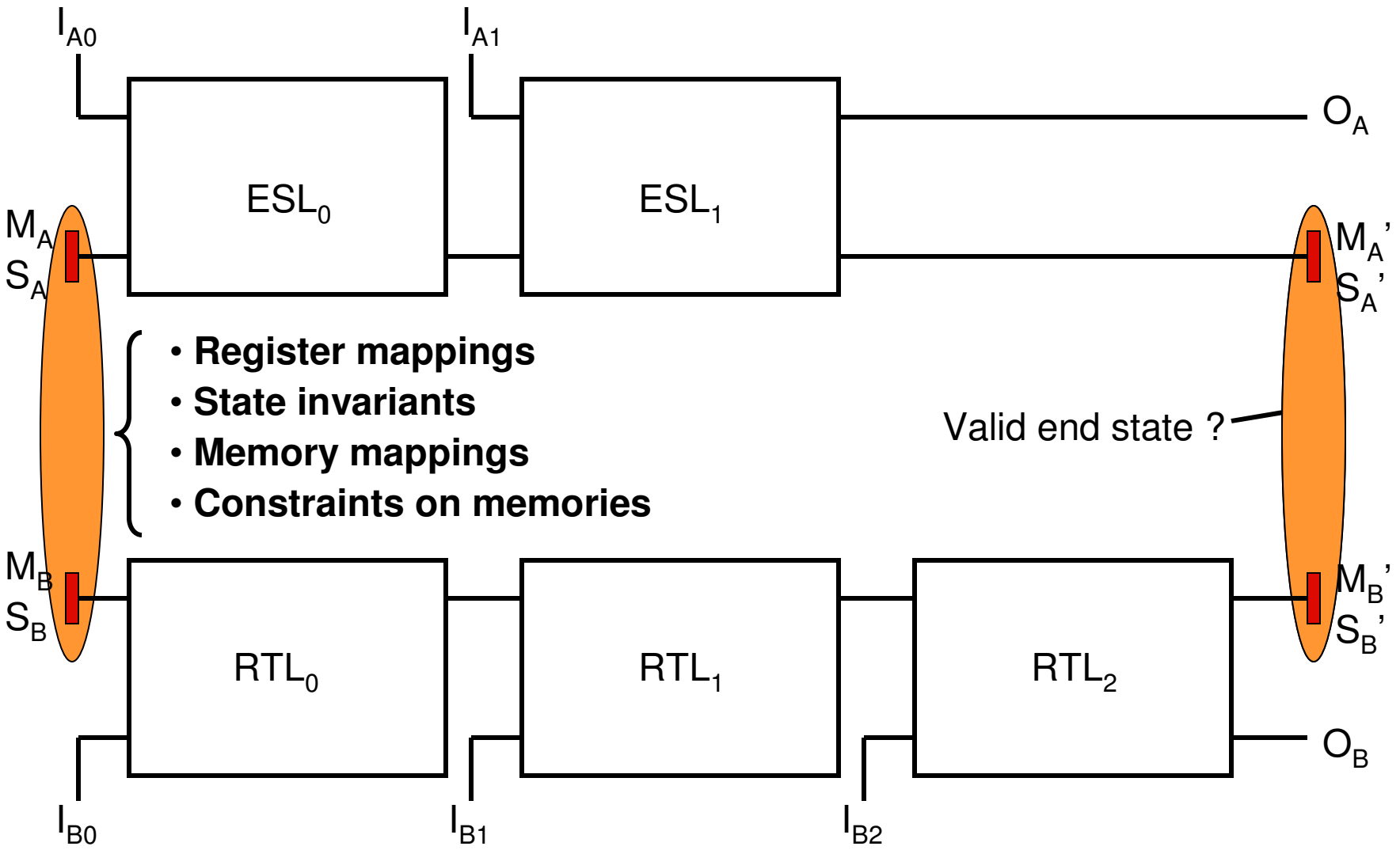
Transaction equivalence



Transaction equivalence



Transaction equivalence



Proof procedure

- **Assumptions**

$$a_0 = MM_0(M_A, M_B) \wedge MM_1(M_A, M_B) \wedge \ldots$$

$$a_1 = c_0(M_A, M_B) \wedge c_1(M_A, M_B) \wedge \ldots$$

$$a_2 = r_0(S_A, S_B) \wedge r_1(S_A, S_B) \wedge \ldots$$

$$a_3 = i_0(M_A, M_B, S_A, S_B) \wedge \ldots$$

- **Proof obligations**

$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow MM_0(M'_A, M'_B) \wedge \ldots$$

$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow c_0(M'_A, M'_B) \wedge \ldots$$

$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow r_0(S'_A, S'_B) \wedge \ldots$$

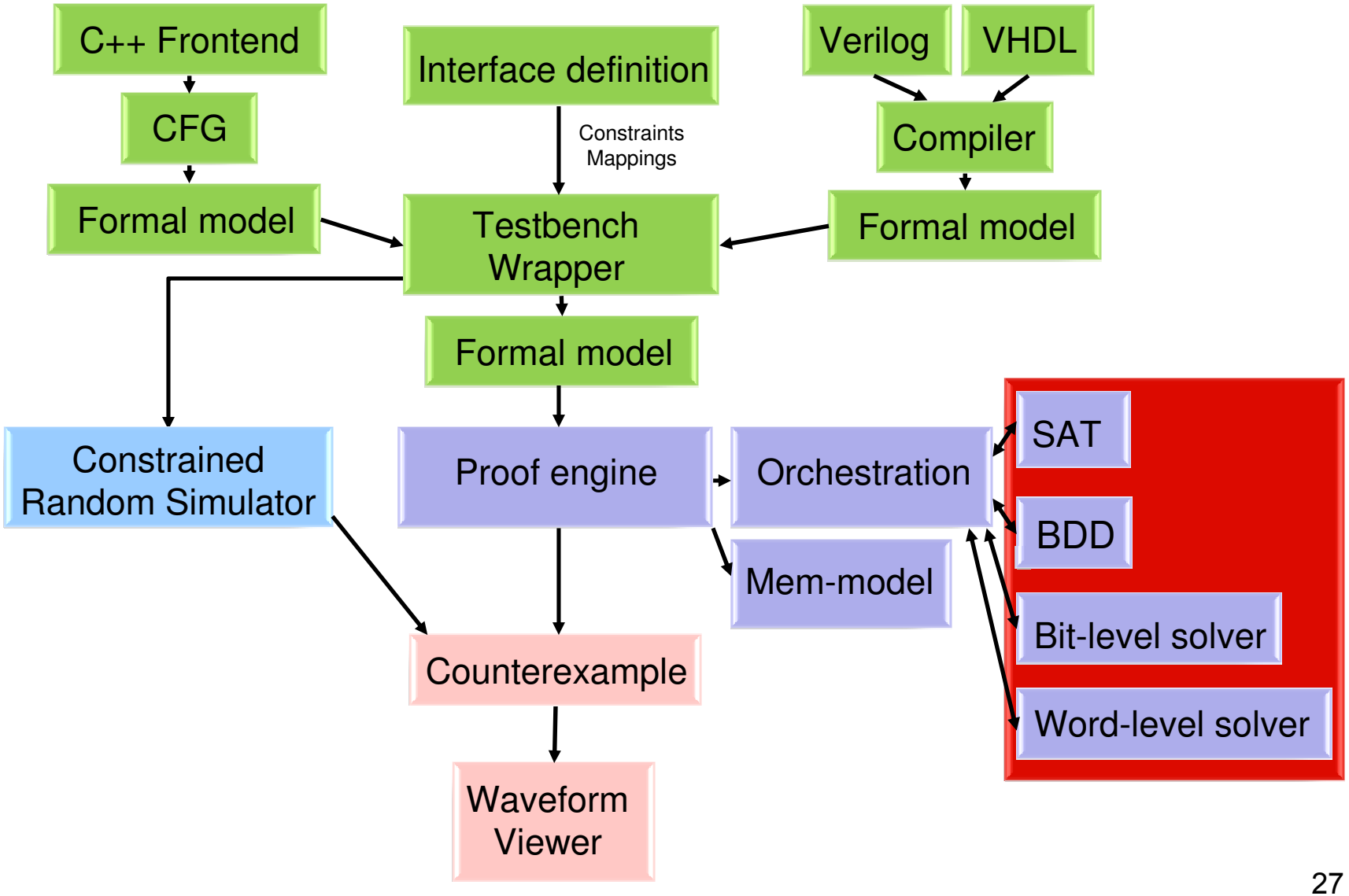
$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow i_0(M'_A, M'_B, S'_A, S'_B) \wedge \ldots$$

$$a_0 \wedge a_1 \wedge a_2 \wedge a_3 \Rightarrow O_A = O_B$$

- **Check model assumptions, e.g., that no array accesses are out-of-bounds**

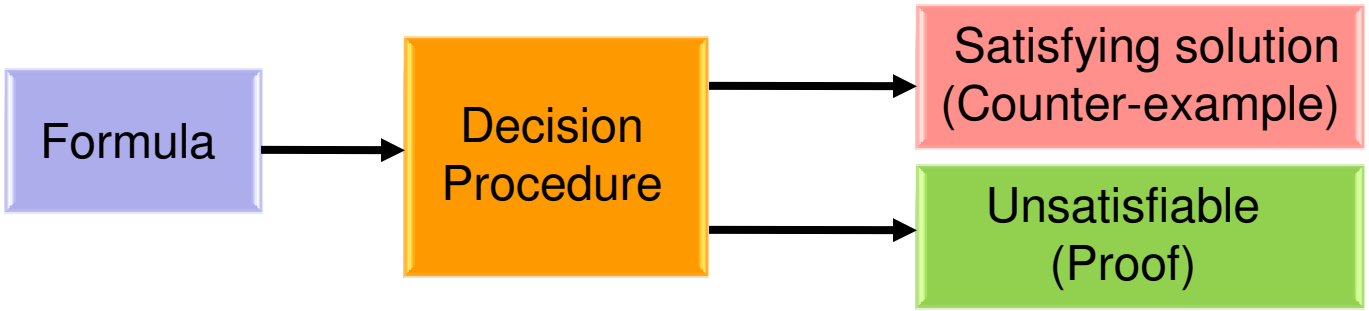
SOLVERS

Solvers


C++ Frontend
CFG
Formal model
Interface definition
Constraints
Mappings
Testbench
Wrapper
Formal model
Verilog
VHDL
Compiler
Formal model
Constrained
Random Simulator
Proof engine
Orchestration
SAT
BDD
Bit-level solver
Word-level solver
Mem-model
Counterexample
Waveform
Viewer

Decision Procedures

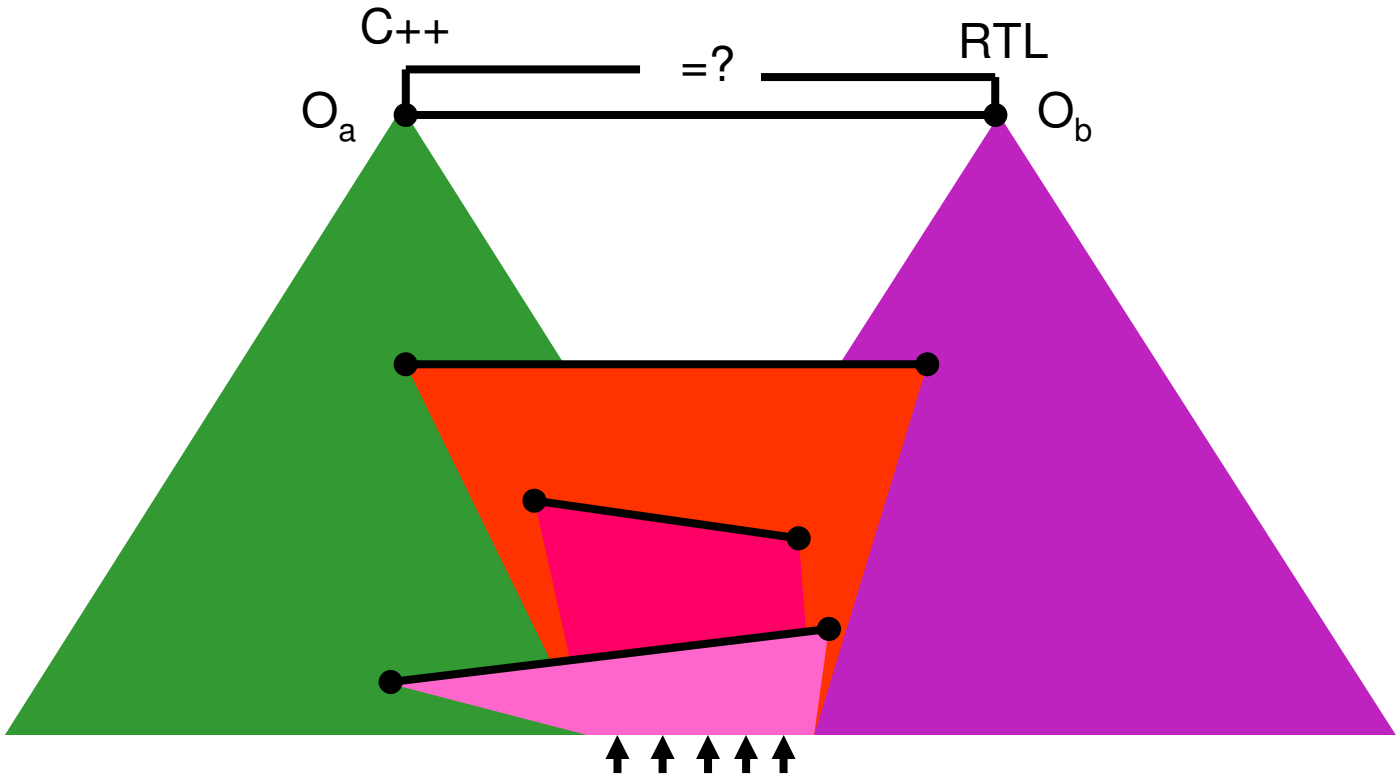
- Core technology for formal reasoning



- Used for intermediate equivalences
- Used for output equivalences
- Word-level solvers
  - Good for equivalent arithmetic
  - Bad for producing counter-examples
- Bit-level solvers
  - Good for falsification
  - Bad for arithmetic

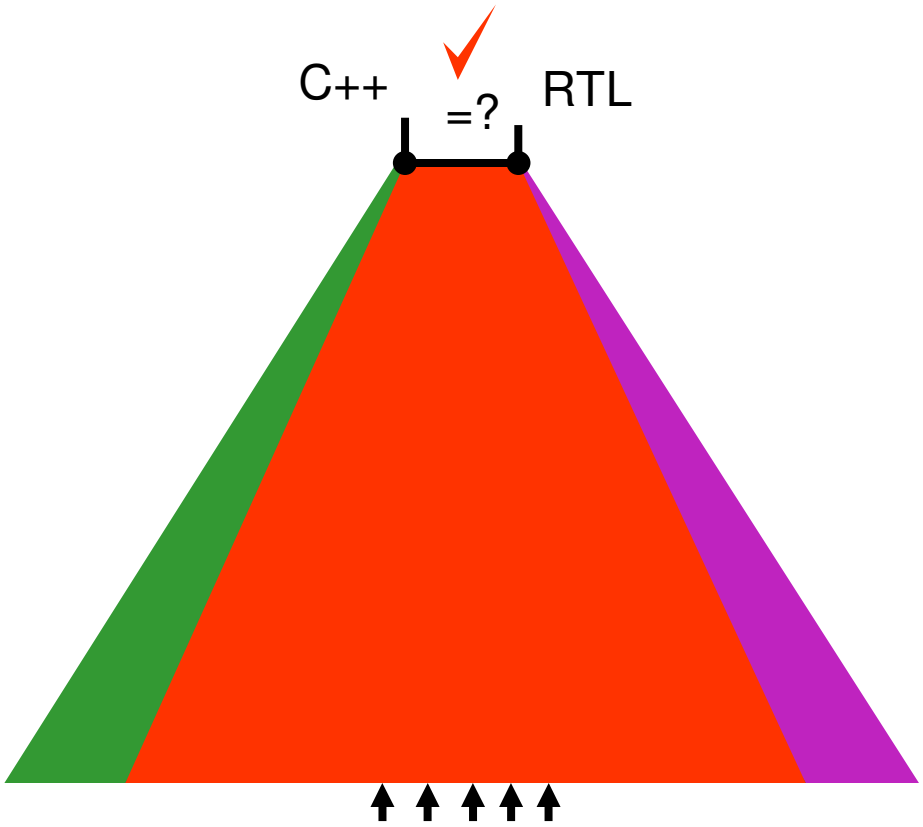
Equivalence check of two DFGs

1. Find potentially equivalent points (PEPs) (e.g. by simulation)
2. Prove them equivalent using bit- and word-level engines
3. Merge equivalent points thereby increasing sharing
4. Prove outputs equivalent



Equivalence check of two DFGs

1. Find potentially equivalent points (PEPs) (e.g. by simulation)
2. Prove them equivalent using bit- and word-level engines
3. Merge equivalent points thereby increasing sharing
4. Prove outputs equivalent



Word-level solvers

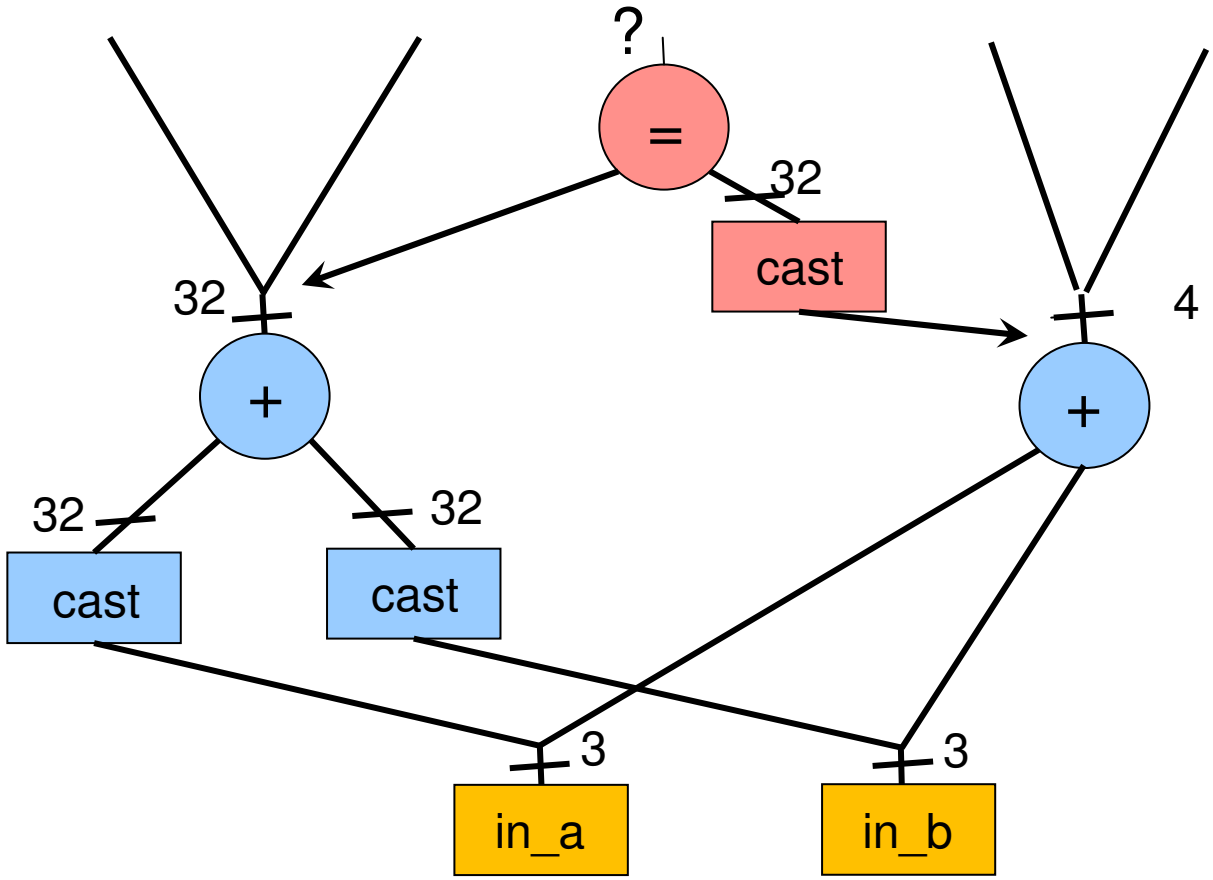
- SMT solvers (SAT module theories)
  - Reason about arithmetic
  - Theories for linear arithmetic, bit-vectors, uninterpreted functions, arrays, real arithmetic
  - Need to be able to deal with finite word-sizes
- Re-writing engines
  - Re-write formulas into normal-form
  - Convergence can be an issue
  - CVCLite from Stanford
- Lessons learned:
  - Only Bit-Vector theory (and maybe theory of arrays if powerful enough) useful
  - Many abstraction techniques are only useful for property checking
  - Few solver techniques specifically target equivalence checking problem

Bit-level solvers

- Construct Boolean circuit based on bit-level representation of operations
- BDDs
  - Canonical representation, very easy to check if formula is unsatisfiable
  - Tendency to memory blowup
  - Good for local intermediate equivalences
  - Good for XOR trees
- SAT
  - Convert circuit to Conjunctive Normal Form (CNF)
  - Branch-and-bound search
  - Efficient optimizations (conflict analysis, non-chronological backtracking)
- ATPG / Circuit-based SAT
  - Branch-and-bound search directly on Boolean circuit

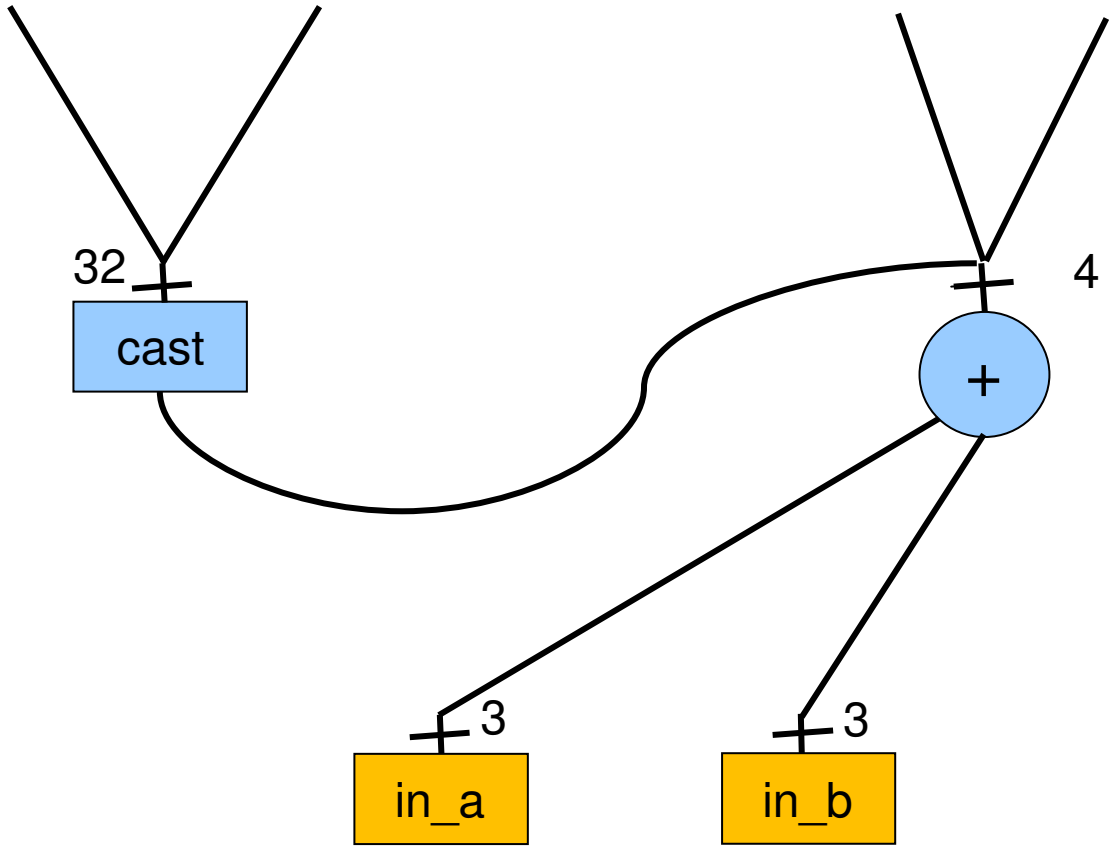
Solver technology

- **Compare word-level graphs modulo zero-extension / sign-extension and merge intermediate equivalent points**



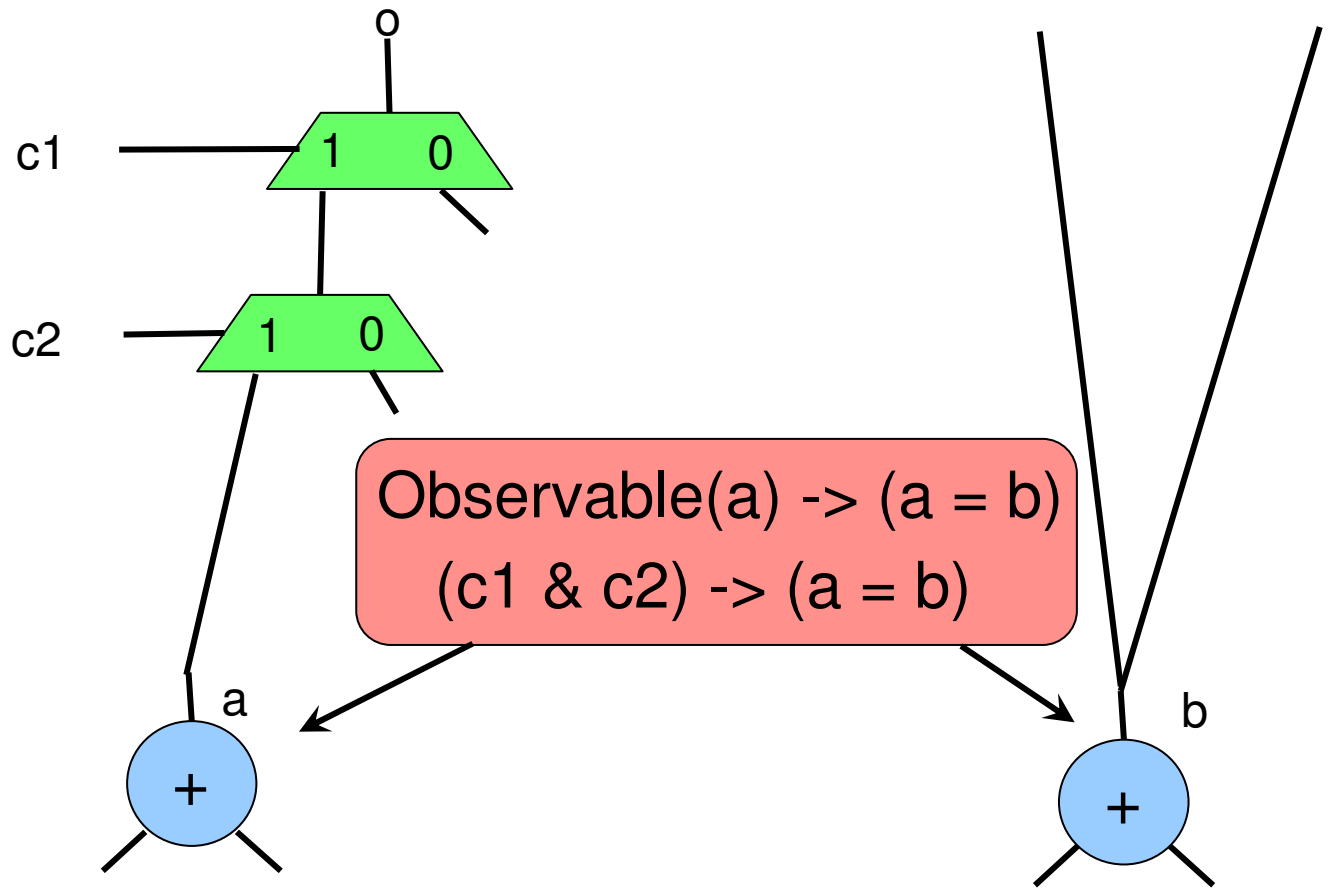
Solver technology

- **Compare word-level graphs modulo zero-extension / sign-extension and merge intermediate equivalent points**



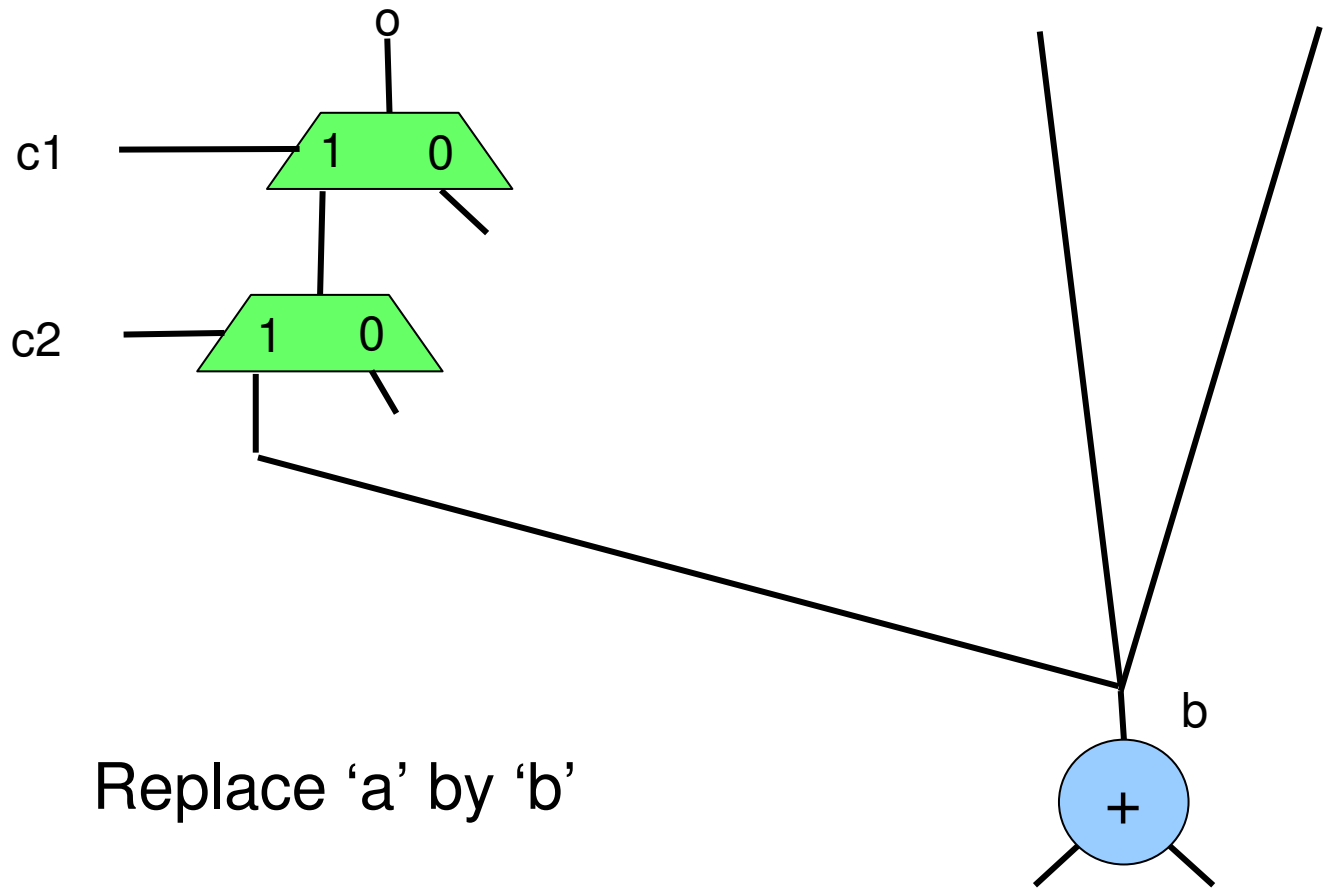
Solver technology

- **Compare word-level graphs modulo observability**



Solver technology

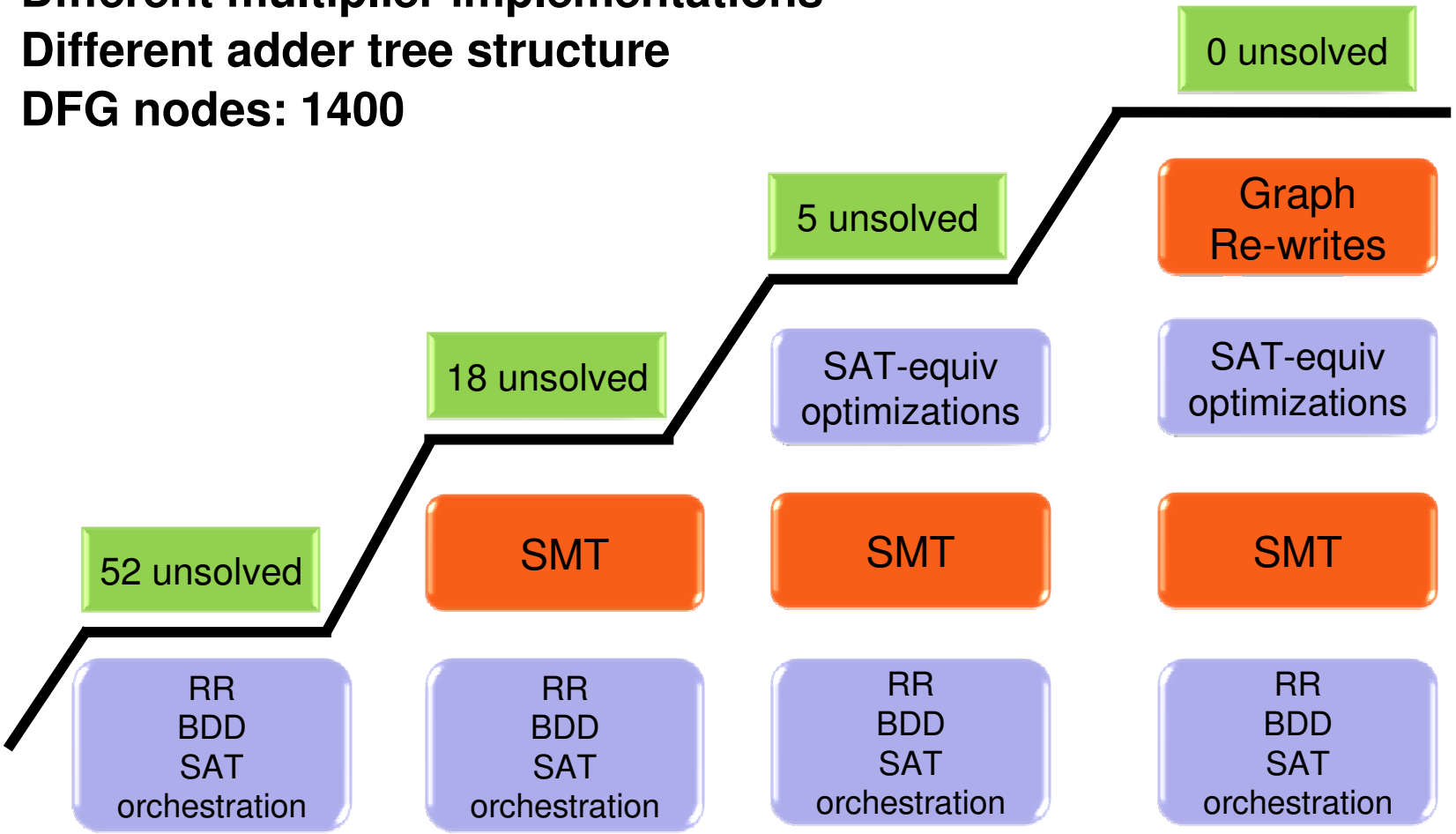
- **Compare word-level graphs modulo observability**



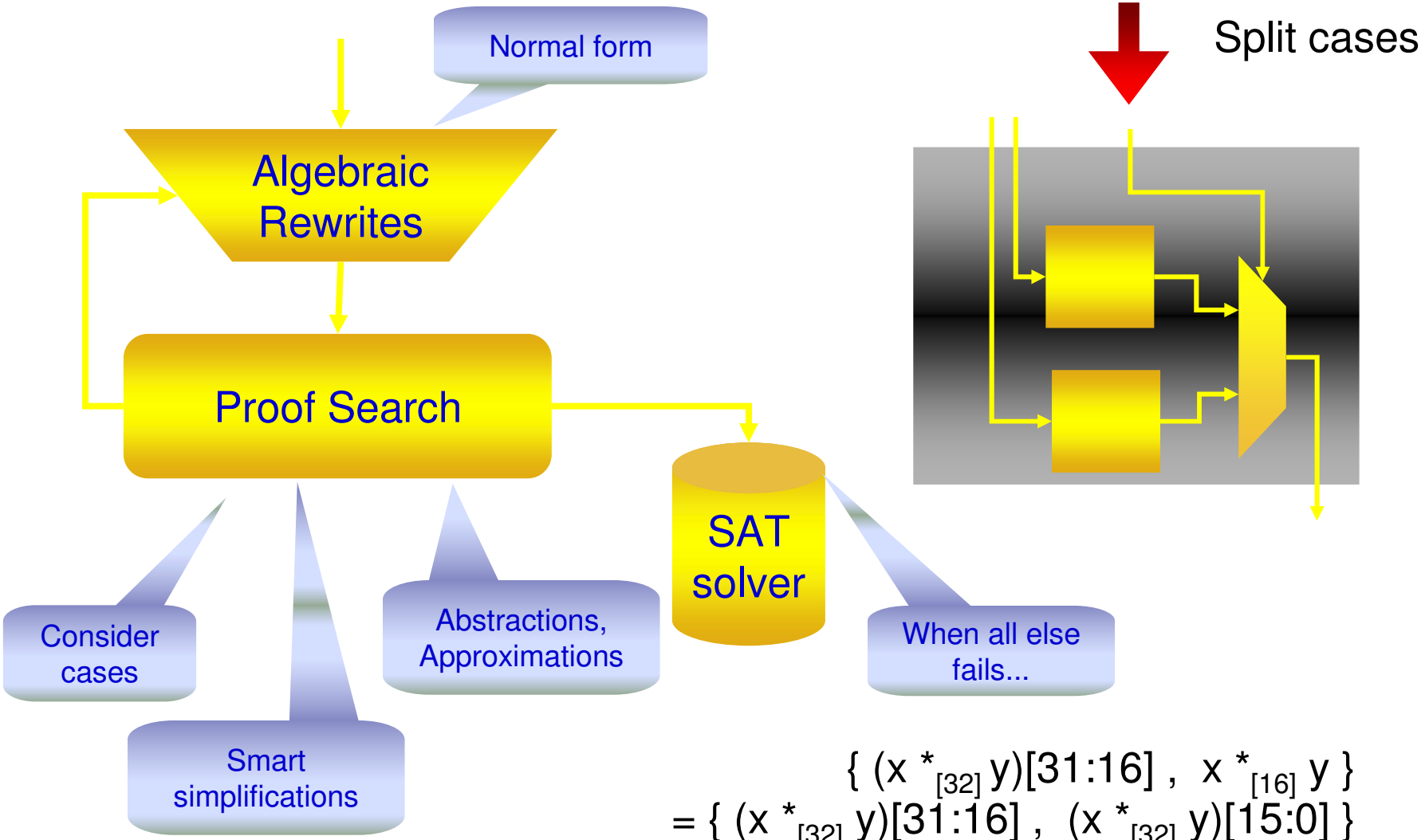
Replace ‘a’ by ‘b’

Effectiveness comes from many techniques

- 68 word (as opposed to bit) outputs
- SL – RTL : different data path architectures
- Different multiplier implementations
- Different adder tree structure
- DFG nodes: 1400



The Algebraic Solver Strategy



$$\{ (x *_{[32]} y)[31:16] ,\ x *_{[16]} y \}$$
$$= \{ (x *_{[32]} y)[31:16] ,\ (x *_{[32]} y)[15:0] \}$$
$$= x *_{[32]} y$$

CUSTOMER EXPERIENCES

COMPANY B

Experience w. Company B

- Ad Hoc (manual) design flow
- All modules are parts of a router design
- Customer wanted free consulting.

- Problems
- Customer did not do block-level verification
- Constraint/counterexample loop
- Manager did not understand the idea of equivalence checking—he thought Hector was a bug finder
- We did the work but eventually the customer could run Hector by herself
- C++ model not entirely complete: one case of two modules in RTL and one in the C++
- Abstracted away the simulation environment manually

Experience w. Company B

- Core algorithms improved greatly during evaluation
- Developed different memory models, e.g., TCAM.

- Successes
  - Were able to conclusively compare all outputs
  - The D5 was not completed by customer

Hector experimental results

Design	# lines of code		# arrays # rams	#disc repan cies	#bugs found	time	final result
	C	RTL					
D1	50	6200	1 / 1	0	0	4min	proven
D2	70	580	1 / 1	0	0	2min	proven
D3	570	1720	1 / 3	9	1 RTL 1 C++	4min	proven
D4	1700	7500	4 / 4	8	1 RTL 1 C++	<1h	proven
D5	4300	6700	31 / 33	>40	4 RTL	43min	62 proven, 15 cex

COMPANY N

Experience w. Company N

- Ad Hoc (manual) design flow
- All modules were from an arithmetic unit: both integer and floating point
- GPU design
- C++ models act as reference models to provide expected/correct output values
- Coverage metrics help but not always reliable
- bugs missed
- Customer was very experienced with formal methods.

Experience w. Company N

- Many mismatches are found
  - Real design bugs were caught
    - mostly corner cases
  - C++ model bugs were found
  - Raised questions on the definition of correct behavior
    - Specification documents clarified/modified
- Some instructions are proven automatically by the tool without any human assistance
- Some instructions are too complex or too large for the tool to handle
- Several techniques for the user to try to assist the tool
- The main theme is divide-and-conquer

Experience w. Company N

- Due to the initial success in finding bugs and proving correctness, the use of high level equivalence checking expands to several designs of company's active GPU development project
  - 10 design blocks, 119 sessions set up and run, 107 proven (some after fixes to bugs found by FV)
    - Includes multiplication logic
- Focused on designs with a high probability of success
  - data transform with simple temporal behavior and input constraints
- A bug was found in a previous project that would have been caught by running this
  - a special case only affects a single input value

Experience w. Company N

- High-level equivalence checking will become part of company's verification plan
  - Demonstrated its value for suitable designs
  - Increase confidence and find difficult bugs more quickly
  - Will not replace other forms of verification, complementary to existing methodology

COMPANY
T

Experience w. Company T

- Designs generated automatically from C++ by Synfora synthesis tool
  - Four designs from four different encryption algorithms + fir filter
  - All four had streams
  - Designs were run entirely automatically!
  - Put in scripting capability to tool
  - Synfora gave Hector hints—all were checked independently
  - Had to support many Synfora features such as streams, bit width pragmas, loop unroll pragmas, memories
  - Hector can now handle loops without unrolling.

Behavioral synthesis result

- Synfora Pico-Extreme synthesized designs
- Encryption designs for GSM/GPRS/UMTS protocols

Design	# lines of code		# arrays # rams	#disc repan cies	#bugs found	time	final result
	C	RTL					
DS1	293	5663	0 / 0	0	0	5min	proven
DS2	579	14015	0 / 0	1	0	17min	proven
DS3	717	11563	2 / 2	2	0	21min	proven
DS4	931	45274	4 / 4	2	0	19min	proven

ADDITIONAL APPLICATIONS

Word/Transaction Level Tools

- Datapath verification in Synopsys's Formality equivalence checker
  - The core solver is the Hector core engine
- Formal front end for SynplicityDSP
- Equivalence checking of Simulation vs. Synthesis models of Synopsys IP
- Model checking at the word level: Bjesse CAV'08, FMCAD'08

Conclusions

- System-level to RTL equivalence checking is a very hard problem
- But… We do it on live commercial designs NOW
- Synthesis is MUCH easier to verify than manual (ad hoc) design flow
- HECTOR is not a product – yet.